

ŽILINSKÁ UNIVERZITA
V ŽILINE

Practical SQL for Oracle Cloud

Michal Kvet

Karol Matiaško

Štefan Toth

Scientific redactor doc. Ing. Michal Záborský, PhD.

Reviewers prof. Ing. Marcel Harakal', PhD.
doc. Ing. Jarmila Škrinárová, PhD.

Copyright © University of Žilina

© M. Kvet, K. Matiaško, Š. Toth, 2022

ISBN 978-80-554-1880-3

Contents

Preface.....	11
Introduction	11
Acknowledgment.....	11
Oracle Academy	12
Organization of the book	14
Lab 1 – Oracle Cloud Infrastructure (OCI)	15
1.1 SQL Developer connection specification.....	38
1.2 SQL*Plus command-line – SQL Client	41
1.2.1 Alternative 1 – full definition	42
1.2.2 Alternative 2 – connect identifiers.....	44
1.2.3 Capturing activities in SQL	49
1.2.4 Working with Help	50
1.2.5 Working with multiple commands	51
1.2.6 Comments.....	51
1.2.7 Working with procedures and functions.....	52
1.2.8 Connection and session termination	54
1.3 Syntax symbols	55
Lab 2 – Basics of data retrieval	57
2.1 Introduction.....	57
2.2 Projection, selection, column alias	58
2.2.1 Personal_id structure	61
2.2.2 Dual table	61
2.3 Using functions	61
2.3.1 Character string functions	62
ASCII function	62
CONCAT function	62
String character case management (LOWER, UPPER, INITCAP functions)...	63
LENGTH function.....	64
SUBSTR function	64
TRIM function	64
2.3.2 Numeric and Math functions.....	65
ABS function.....	65
CEIL function.....	65
ROUND function	66
FLOOR function	66
TRUNC function	66
MOD function	67
2.3.3 Date and Time functions	67
SYSDATE function.....	68
SYSTIMESTAMP function	68
ADD_MONTHS function	69
EXTRACT function	69
LAST_DAY function.....	70
MONTHS_BETWEEN function.....	70
NEXT_DAY function	71

TRUNC function	72
2.3.4 Conversion functions.....	73
TO_CHAR function	73
TO_DATE function.....	75
TO_NUMBER function	75
TO_TIMESTAMP function	75
2.3.5 Advanced functions.....	76
CASE conversion function.....	76
COALESCE function	77
DECODE function	77
NULLIF function	78
NVL function	78
NVL2 function	78
USER function	79
SYS_CONTEXT function.....	79
2.4 Managing NULL values.....	80
2.5 Comparing strings (equality, operator Like)	81
2.6 Using Order By clause	83
2.7 Table joining	85
2.8 Cartesian product	88
2.9 SETs operations (IN, EXISTS)	90
2.10 Managing duplicate values.....	94
2.11 Table alias	95
2.12 Practice.....	96
Lab 3 – Insert, Update, Delete statements and transactions.....	99
3.1 Introduction.....	99
3.2 Insert statement	99
3.2.1 Insert – values type.....	100
3.2.2 Insert – Select type	101
3.3 Update statement.....	102
3.4 Delete statement	104
3.5 The order of operations	105
3.6 Foreign key definition.....	105
3.7 Changing the primary key value	106
3.8 Transactions	107
3.9 Practice.....	109
3.9.1 Insert statements	109
3.9.2 Update statements.....	110
3.9.3 Delete statements.....	110
Lab 4 – Data modeling	113
4.1 Introduction.....	113
4.1.1 System analysis	113
4.1.2 System design.....	114
4.1.3 Technical design.....	114
4.2 Creating data model	114
4.3 Conceptual modeling	117
4.4 Entity-relational conceptual model	118

4.4.1	Identifying key	119
4.5	Conceptual schema notation in E-R model	119
4.5.1	Linear notation	119
4.6	Type diagram / Occurrence E-R diagram.....	119
4.6.1	Type diagram.....	120
4.6.2	Occurrence E-R diagram	120
4.7	Attributes.....	120
4.7.1	Non-atomic attributes	122
4.7.2	Group attributes	122
4.7.3	Multiple value attributes.....	122
4.8	Relationships and integrity constraints.....	123
4.8.1	Identifying and non-identifying relationship.....	123
4.8.2	Relationship cardinality.....	124
	Cardinality 1:1	124
	Cardinality 1:N.....	125
	Cardinality M:N	125
4.8.3	Decomposition of the M:N relationship cardinality	126
4.8.4	Associative entity	128
4.8.5	Membership types	129
4.8.6	Multiple relationships between same tables	130
4.8.7	Recursive (self) relationships	131
4.9	Data modeling in Toad Modeler tool	131
4.9.1	Environment settings.....	132
4.9.2	Entity management.....	133
4.9.3	User-defined domain	137
4.9.4	Relationship management	141
4.9.5	Generating SQL script.....	142
4.9.6	Executing script on the server	145
4.9.7	Working with directories and files	146
4.10	Practice.....	148
Lab 5 – Create, Alter and Drop commands		151
5.1	Introduction.....	151
5.2	Data types.....	152
5.3	User management.....	153
5.4	Table management	155
5.4.1	Create command.....	156
	Foreign key.....	158
	Domain definition (check constraint).....	159
	Default value	160
	Constraint naming	160
	Create table as Select.....	160
5.4.2	Alter command.....	162
	Add option.....	162
	Modify option.....	163
	Drop option	164
	Table renaming.....	164
5.4.3	Drop command.....	165

Recycle bin	165
5.5 Index	167
5.5.1 ROWID	167
5.5.2 Index management	168
5.5.3 Types of indexes.....	168
B+ tree index type	168
Bitmap index	171
Index organized table	172
5.5.4 Access methods	172
5.6 Practice.....	173
Lab 6 – Data loading	175
6.1 Introduction.....	175
6.2 SQL Loader.....	175
6.3 EXP / IMP utility	184
6.4 Creating import/export using dump files.....	185
6.4.1 Import using data pump.....	185
Object storage.....	186
Bucket	186
Create_credentials procedure	189
Authentication token	190
Data Pump Import Wizard	195
Bucket	205
Object.....	205
6.4.2 ExpDp	207
6.4.3 Useful notes.....	216
Lab 7 – Managing privileges	217
7.1 Introduction.....	217
7.2 Grant command.....	217
7.2.1 System privilege management.....	217
7.2.2 Object privilege management.....	219
7.3 Accessing another schema object.....	220
7.4 Revoke command.....	220
7.5 Grouping privileges to roles	223
7.6 Practice.....	224
Lab 8 – Advanced techniques of data retrieval.....	225
8.1 Introduction.....	225
8.2 Aggregate functions	225
8.3 Fundamentals for Group By clause management.....	227
8.4 Working with aggregate function Count and Group By clause.....	228
8.5 Having clause	233
8.6 Extended versions of table joining	235
8.6.1 INNER JOIN type	236
8.6.2 ON / USING CLAUSE	237
8.6.3 LEFT OUTER JOIN type.....	237
8.6.4 RIGHT OUTER JOIN type.....	238
8.6.5 FULL OUTER JOIN type	238
8.6.6 SEMI JOIN type.....	239

8.6.7	ANTI JOIN type.....	239
8.6.8	NATURAL JOIN type	240
8.7	Relational algebra operations	240
8.7.1	Union operation.....	241
8.7.2	Difference operation.....	244
8.7.3	Intersection operation	245
8.8	Recursive relationships	246
8.9	Using the same table multiple times in the Select statement.....	249
8.10	Practice.....	250
Lab 9 – Procedures, functions and packages		253
9.1	Introduction.....	253
9.2	Code preliminaries	254
9.2.1	Variable definition.....	254
9.2.2	Assignment, NULL	254
9.2.3	Conditional processing	255
	IF condition	255
	Condition CASE.....	259
9.2.4	LOOPS.....	263
	Infinite loop, EXIT condition	263
	WHILE loop type	264
	FOR loop type	264
9.3	PL/SQL anonymous block	265
9.4	Procedure, function	266
9.4.1	Procedure syntax	267
9.4.2	Function syntax	268
9.5	Executing stored method.....	269
9.5.1	Disable procedure.....	270
9.5.2	Enable procedure.....	270
9.5.3	Get_line procedure	271
9.5.4	Get_lines procedure.....	271
9.5.5	New_line procedure	271
9.5.6	Put procedure.....	271
9.5.7	Put_line procedure.....	272
9.6	Calling function from the Select statement	273
9.7	Exception handling.....	275
9.8	Ways of passing parameters	282
9.8.1	Position way of passing parameters.....	282
9.8.2	Passing parameters using names	283
9.8.3	Hybrid passing.....	284
9.9	Differences between anonymous and stored (named) PL/SQL block	284
9.10	Removing procedures and functions	284
9.11	Select statement in PL/SQL	285
9.11.1	SELECT INTO type.....	285
9.11.2	CURSOR.....	286
9.12	Increasing control – access rights.....	292
9.13	Packages.....	296
9.13.1	Package specification syntax.....	297

9.13.2	Package body syntax	298
9.13.3	Overloading.....	301
9.13.4	Initialization block.....	302
9.14	Practice.....	305
Lab 10 – Triggers		307
10.1	Introduction.....	307
10.2	Syntax	308
10.3	Restrictions for trigger definition.....	311
10.4	Triggers turning on and off	311
10.5	Changes monitoring	311
10.6	Default values	314
10.7	Conditions for trigger firing	315
10.8	One trigger – multiple operations.....	318
10.9	Referential integrity management	320
10.10	Changing the value of the primary key	322
10.11	Sequences and triggers.....	323
10.11.1	Sequence syntax.....	323
10.11.2	Sequence and transaction correlation.....	326
10.12	DDL triggers.....	327
10.13	Event triggers.....	329
10.14	Practice	330
Lab 11 – Relational integrity		331
11.1	Introduction.....	331
11.2	Integrity constraints classification.....	331
11.3	Entity integrity	332
11.3.1	Primary key candidate	332
11.3.2	Primary key	332
11.3.3	Alternative key	333
11.3.4	Superkey.....	333
11.4	Referential integrity	333
11.4.1	Referential integrity rule	333
11.4.2	Referential integrity consequences	334
11.4.3	Cascade option example.....	334
11.4.4	Restricted option example	336
11.4.5	Nullified option example.....	337
11.5	User integrity.....	338
11.6	Column integrity	338
11.7	Domain integrity	339
11.8	Integrity constraints controlling and processing.....	339
11.9	Practice.....	339
Lab 12 – Views.....		341
12.1	Introduction.....	341
12.2	Syntax	341
12.3	Exceptions.....	342
12.4	Managing data in views	344
12.5	Attribute name redefinition in views.....	347
12.6	Check option clause	347

12.7	Read only view.....	349
12.8	View based on multiple tables and triggers.....	350
12.9	Triggers associated with views	350
12.10	Summary.....	351
12.11	Practice	351
Lab 13 – Date and Time value management.....		353
13.1	NLS parameters & session format	358
13.1.1	NLS_Language.....	359
13.1.2	NLS_Territory	360
13.1.3	NLS_Date_Language	360
13.1.4	NLS_Date_format	361
13.2	Transformation of the personal_id into the date of birth.....	361
13.3	Get the list of persons who celebrate a birthday today.....	362
13.4	Get the list of students who passed the exam this month	363
13.5	Get the list of students who passed the exam previous last month.....	364
13.6	Get the list of the persons, who will celebrate their birthday next Sunday ..	366
13.7	Get the Date of the second Sunday of the month	368
13.8	Get the list of the persons, who will celebrate their birthday next week	369
13.9	Get the difference between Date values	370
13.10	Get the difference between Date values – a sophisticated solution	370
13.11	YY vs. RR.....	372
13.12	Actual employees.....	373
13.13	Period models and Allen relationships	374
13.14	Unlimited validity definition	377
13.15	Data type Interval management	378
13.15.1	Interval Year to Month data type	378
13.15.2	Interval Day to Second data type	379
13.15.3	Examples – Interval data types	380
13.15.4	Update validity definition based on Interval data value.....	380
Lab 14 – Data dictionary views		383
14.1	Introduction.....	383
14.2	Data dictionary – structure	384
14.3	Querying data dictionary	387
14.3.1	List of tables owned actual user	387
14.3.2	List of table attributes.....	387
14.3.3	Get attribute data type and characteristics	387
14.3.4	Get system identifier and definition of the primary key.....	389
14.3.5	Get system identifier and definition of the foreign key	390
14.3.6	Listing triggers for a particular table	392
14.3.7	Listing developed methods (procedures, functions).....	392
14.3.8	Managing sequences.....	396
14.4	Practice.....	397
Lab 15 – Reports		399
15.1	Overview	399
15.2	Environment settings, background.....	400
15.3	Filtering, sorting	406
15.4	Hidden columns	413

15.5	Binding multiple reports – Master – Child.....	414
15.6	Graph reports.....	422
15.7	Pie graph type reports.....	426
15.8	Line type reports	428
15.9	Three-dimensional (3D) graph types.....	434
15.10	Binding multiple reports of various types.....	436
15.11	Exports.....	438
15.11.1	CSV format	440
15.11.2	Delimited format.....	441
15.11.3	Text format.....	442
15.11.4	Excel format.....	443
15.11.5	XML format.....	445
15.11.6	HTML format.....	446
15.11.7	Exporting to PDF	450
15.12	Script format (Insert)	453
	Summary	455
	References	457
	Abbreviations.....	461
	Index	465
	Appendix A – Model Student	473
	Table PERSONAL_DATA	473
	Table STUDENT.....	476
	Table STUDY_SUBJECTS	479
	Table ST_FIELD	482
	Table SUBJECT	484
	Table TEACHER.....	486
	Table SUBJECT_YEAR	488
	Table ST_PROGRAM.....	490
	Table CONTACT	493
	Appendix B – Model Flight.....	495
	Table L_PERSON	495
	Table L_FLIGHT TICKET	497
	Table L_CLASS	500
	Table L_FLIGHT	502
	Table L_PLANE.....	505
	Table L_EMPLOYEE	507
	Table L_AIRPORT.....	510
	Table L_PLANE_TYPE.....	512
	Table L_COUNTRY	514
	Table L_TOWN.....	515
	Table L_AIR_COMPANY	517
	Appendix C – Model Library	519
	Table K_PERSON.....	519
	Table K_READER	522
	Table K_RENT_BOOKS	524
	Table K_BOOK.....	526
	Table K_TITLE.....	528

Table K_AUTHOR.....	530
Table K_AUTHORS_OF_BOOK.....	532
Appendix D – Syntax.....	535
Appendix E – File management	543

Errata

Authors make every effort to make sure no errors are present in the text. If you find any typo or mistake, that has not been reported, yet, please, let us know. Errata sheets are available here: <https://gofile.me/4voWB/mK3v2SzfU>



Preface

Introduction

We have just prepared the first edition of the book for people to increase practical knowledge and skills in the area of Database systems. The content and labs of the textbook are prepared under our experiences with the education of Database systems at the University of Žilina, Slovakia, supervised by the discussion with the consortium members, experts, and Oracle Academy.

From time to time, we perceive that students, researchers, or practitioners have problems with the correct way of using Database systems during the implementation processes into any information systems.

This book was written for students and practitioners. It is intended as a practical guide for them and other developers to analyze, design, and implement commercial information systems. The language and diagram conventions apply ANSI standards with the strength of the Oracle Autonomous Database used in Oracle Cloud. Toad data modeler is used for data modeling and visual data model preparation.

We suppose that readers can recognize that using database systems and SQL is essential knowledge for the design process, with opportunities for choice and creativity. Nowadays, cloud technology is ubiquitous and provides a robust general solution. Therefore, we have chosen to use the Oracle Cloud environment. Moreover, Free Tier and Always Free option gives you many benefits (autonomous transaction processing database, data warehouse, APEX (tool for data-driven applications definitions), object storage, etc.) free of charge. Such an option is mainly devoted to the testing and development environment but can perfectly fit the self-teaching process. You do not need extra hardware; no installation and administration are necessary.

The text proposes many practical exercises highlighting the problems, solutions, tricks, and improvements to provide a robust, reliable solution and knowledge extension. Each chapter consists of a brief theory overview supervised by the discussion and practical examples.

Acknowledgment

These textbook and e-book versions were prepared during the implementation of the *Cloud cOmputing for Digital Education Innovation (CodeIn)* project – Erasmus+ Strategic Partnerships, Key Action 2, Project Number: 2020-1-HR01-KA226-HE-094713. It is devoted to education using Cloud technology computing. *Oracle Cloud* focuses on the Oracle autonomous databases (transaction processing (ATP), data warehouse (ADW), or JSON).

The proposed book is partially covered by the Erasmus+ project Better Employability for Everyone with APEX (BeeAPEX), supporting the digital transformation of higher education institutions through the development of the digital readiness, resilience, and capacity of educators and students. Grant No.: 2021-1-SI01-KA220-HED-000032218.

Proper knowledge of SQL and PL/SQL is crucial for the data driven application development.



Oracle Academy

Oracle Academy welcomes the publication of this textbook written by experts from the University of Žilina in Slovakia as a significant contribution to the teaching and learning about Oracle technologies, not just at the university but also in the Slovak Republic and the broader international community of educators and learners. It demonstrates how the determination and innovation of lecturers, and the availability of state-of-the-art technologies provide new learning opportunities for students, helping them gain skills and knowledge for their future careers in both local and global IT markets. In this section, we would like to provide a short overview of this program and its free resources that can benefit educators and students using this textbook to achieve their teaching and learning objectives.



About Oracle Academy

As Oracle's global, philanthropic educational program, Oracle Academy advances computing education around the world to increase knowledge, innovation, skills development, and diversity in technology fields. This program engages with thousands of educational institutions and educators in more than 130 countries, helping millions of students become college and career ready.

Oracle Academy provides educators with free teaching resources for computing education including curriculum focused on Java, database, cloud, and project management; Oracle Cloud and Autonomous Database technologies through the Oracle Academy Cloud Program; Oracle APEX low code learning environments; a wide range of software; professional certification resources; and continuing professional development for educators. All teaching and learning resources are designed for degree-granting academic programs of study.

Oracle Academy Cloud Program

The Oracle Academy Cloud Program offers Oracle Academy members exclusive access to the Oracle Cloud Free Tier, a set of services educators and students can continue to use for an unlimited time, even after their graduation, with easy, accelerated signup and no need for a credit card, mobile phone contact information, or any approval delays.

With this free program, Oracle Academy members can teach and learn in the cloud. They can build, learn, and explore the full functionality of Oracle Autonomous Database, the world's only self-driving database, and Oracle Cloud infrastructure for an unlimited time. Plus, they can use free developer tools and get started quickly and learn and practice new technologies with just a one-time classroom setup, saving hundreds of hours of time over years of teaching.

A simple signup process enables member educators and students to also access Compute Virtual Machine or VM, object storage, data egress, and other essential developer building blocks. Educators easily can provision student accounts, and classes are up and running in minutes in a cloud environment without the need to download, install, patch, or maintain software or databases.

New services continually are added to the Always Free Services, and in addition, at the time of publication of this textbook, member educators and their students also receive US\$300 of free credits for one year to prototype applications, run machine learning models

in notebooks, or try software from Oracle Cloud Marketplace. These credits can be spent without providing any credit card details.

Teachers and students enjoy always free access to tools including Oracle Application Express (APEX) for low code Web application development, SQL Developer Web for working with Oracle Autonomous Databases, SQL Notebooks for Machine Learning, Oracle REST Data Services for web interfaces, and Oracle Instant Client for the most popular programming languages. Other cloud technologies include Linux, AI/ML, and digital assistants; students can develop in SQL, NoSQL, APEX, Java, Node.js, Python, PHP, and Ruby.

Supported by Oracle Academy's comprehensive curriculum and hands-on labs, educators and students can teach, build, learn, explore, and develop in the cloud.

In the global classroom, educators and students can take advantage of Oracle Cloud technology for teaching and learning — anytime, anywhere. The cloud is always available, in and out of the classroom, using only an Internet browser, through the Oracle Academy Cloud Program.

Students must be the age of legal majority in their country of residence to access a cloud account.

Faculty can learn more and sign up at <https://academy.oracle.com/cloud>.

Oracle Cloud Infrastructure Foundations I Curriculum

Oracle Cloud Infrastructure (OCI) leads cloud computing with a deep and broad platform of cloud services that enables customers to build and run a wide range of applications in a scalable, secure, highly available and high-performance environment.

The new Oracle Academy Cloud Infrastructure Foundations I curriculum helps students build foundational knowledge of cloud computing by focusing on OCI concepts and terminology through lesson slides, corresponding videos and demonstrations, hands-on labs, and midterm and final exams. Throughout the course, learners gain an understanding of the core infrastructure of cloud, how it works with databases, and information on security, administration, monitoring, and management.

This curriculum is currently available in English only. The recommended total course time, which includes instruction, self-study, videos, and assessment is 90 hours.

Access the full course description under the Cloud Curriculum section of the Oracle Academy website, academy.oracle.com.

Join Oracle Academy

Oracle Academy requires Institutional membership for those institutions and their educators who wish to take advantage of our wide range of free teaching and learning resources. Membership requires completing an agreement and is free.

Members access our free resources through the Oracle Academy Member Hub, a state-of-the-art learning management system. They simply log in on the home page of <https://academy.oracle.com> —

It is easy to join Oracle Academy — and it is free.



Organization of the book

The textbook itself is divided into nine parts organized into fifteen chapters. Besides the Introduction where we described the aims, prerequisites, and necessary environment for the correct work with the Database system, extended by the Oracle academy membership registration and web access structure, the following parts can be recognized:

- Data Manipulation Language (DML),
- Data modeling,
- Data Definition Language (DDL) and the data loading process,
- Advanced SELECT statements,
- PL/SQL introduction,
- Data integrity (DI),
- Data dictionary and additional SQL extension,
- Data reports.

Each chapter contains a short description of the theory, examples, and tasks to evaluate the received knowledge.

During lab 1, we enclosed the documentation:

- for the Oracle Cloud registration,
- advantages of using the Oracle Cloud Always Free option,
- resource categories available for you,
- basic environment navigation,
- first examples of the verification of the functionality.

In the sections highlighting the DML statements, the first attempts with SELECT statements are included with a detailed description of the INSERT, DELETE and UPDATE statements.

The next part is represented by the fifth chapter characterizing the independent part with all necessary knowledge about data modeling.

The part about DDL includes details about Data Definition Language, statement syntax and categorization, description of the available data types, Data Access statements (DAS, Data Control Language (DCL)), and about importing and exporting data to and from the database.

Advanced SELECT statements include a description of the aggregate functions and their management, Group By clause management fundamentals, and table joining options.

Lab. 9 covers the procedural extension of the SQL language. It deals with the procedures, functions, and packages. It also deals with Select statement management in blocks, exception handling, and details about work with methods, and cursors. Lab. 10 deals with the triggers associating the code with the operations fired automatically.

The part about data integrity offers the rules to keep the database in a correct and consistent state. This part contains information about working with views and their influence on the data integrity, as well.

The last part extends the practical knowledge and skills related to working with temporal data types and reports covered by the data dictionary. In the end, we added the Appendix with three models for the practices and the verification of embedded examples and tasks.

We believe that this textbook will be a helpful document for gaining theoretical and practical knowledge of modern database systems.

Lab 1 – Oracle Cloud Infrastructure (OCI)

This lab will drive you through the cloud management principles using the architecture and product types. It will discuss the registration process followed by the terminology summary and database provisioning. Access to the database is done by the SQL Developer application (web or desktop version) or SQL Client.

Connection specification is made of the host, port, service name or SID. It can be specified by the full connection, or stored connect identifiers can be used, referenced by the TNS_ADMIN variable, commonly stored as an environment variable. Whereas the whole communication between the cloud repository and client is secured, encryption keys must be properly used and stored in the Oracle Wallet.

Reader will understand the basic primitives showing him a simple query, procedure, and function execution (deeper discussion is in the lab. 9). There is also discussion related to the table structure and data types of the attributes.

The complex code should use comments for the consecutive reference, evaluation, and upgrade, respectively, which can be either one-line or multi-line types. Each data operation is part of the transaction. If the data were changed, it is necessary to navigate the database system to the approval or operation reject. Reaching Exit automatically approves the active transaction (Commit).

Finally, there is a summary of the syntax notation. Note that the principles and syntax can be found in the documentation, but the Help command can also be used. Individual activities can be recorded using the Spool command.

Oracle Cloud Infrastructure (OCI) uses the Infrastructure as a Service (IaaS) principles to extend the original on-premise systems with high-performance computing power running in a cloud environment. The main advantage is the elasticity, so the system can dynamically reflect current workload, processing demands, and user activity. It uses *Oracle autonomous services*, an integrated security layer, robust functionality, and optimization techniques. OCI brings many benefits to your performance and processing by autonomous services, easy migration, costs reduction, or performance enhancements.

OCI is a residual and exclusive location of the *Oracle Autonomous Database*. It is self-administering, self-repairing, or self-patching. Leveraging machine learning to automate routine tasks, Autonomous Database delivers higher performance, better security, and improved operational efficiency, and frees up more time to focus on building enterprise applications (<https://www.oracle.com/cloud/>).

This chapter will navigate you through the process of the *Oracle Cloud account creation*, registration, up to the connection possibilities using the SQL developer installed either locally or by using web sources. Before we start, let me summarize products available in OCI:

- Oracle analytics using built-in machine learning and artificial intelligence to propose a robust solution for the company and offer better decision-making opportunities. It covers Oracle Analytics Cloud, Oracle Big Data Service, Oracle Big Data SQL Cloud Service, Oracle Data Science, Oracle Cloud Infrastructure Data Flow, and many more.
- Application development environment pointing to the data-driven application development simplifying the whole process. It covers API Gateways, Blockchain,

Data Science, Digital Assistants, Java functionality, Events Services, Mobile Hubs or Oracle MySQL Database Service, Oracle MySQL Database Service (and many more). Two solutions should be emphasized – *Oracle Application Express (APEX)* and *Visual Builder*. These tools provide you with a complex environment to create web or mobile-based applications based on the SQL, PL/SQL, or JavaScript functionality. Thus, by using these tools, implementation is far easier ensured by the rapid development. The solution can be created from evening to morning.

- Applied Software Technologies like AI, Blockchain, machine learning, data science, digital assistants, etc.
- Compute – scalability reflecting the workload to ensure performance.
- Database – Autonomous Transaction Processing, Autonomous Data Warehouse, Autonomous JSON Database, Database Cloud Service (Bare Metal / Virtual Machine), Exadata Cloud Service, ...
- Integration (API Gateway, Application Integration, Oracle GoldenGate, Oracle Data Integrator, Oracle Cloud Infrastructure Data Integration, SOA Cloud Service).
- Observability and Management (Logging, Monitoring, Notifications, Resource Manager, ...).
- Networking and Connectivity (DNS, E-mail delivery, FastConnect, Health Checks, Load Balancing, Virtual Cloud Network, ...).
- Security, Identity, and Compliance.
- Storage (Archive Storage, Block Volumes, Data Transfer, File Storage, Local NVMe SSD, Object Storage, Storage Gateway).

Oracle Cloud technology is widespread across the whole world, divided into commercial and government types. In Europe, clouds are located in multiple cities, like *Amsterdam*, *London*, *Frankfurt*, *Zürich*, or *Newport*. New region centers opened in 2021 are in *Sweden*, *France*, and *Italy*. Oracle is constantly expanding and opening new data centers within its Oracle Public Cloud. Fig. 1.1 shows the reference of January 2022. More about the current state can be tracked using the <https://www.oracle.com/cloud/> web address.



Fig. 1.1: Cloud regions (source: Oracle Cloud presentation, © Oracle)

Oracle Cloud Infrastructure is prepared chiefly for the commercial and government environment to use all the benefits. It is paid based on resource consumption. Thus, it can

lower the total costs and demands of the organization by shifting the environment and administration to Oracle.

OCI has launched a significant project to offer cloud services to the students, as well. *Oracle Cloud Always Free* version is provided to the students or for the testing environment suitability. Services are time-unlimited with the following resource limitations:

- 2 Autonomous Databases limited by the 1 OCPU and 20GB of disc storage for each,
- Compute Virtual Machines (VMs),
- 2 Block Volume Storage – 100 GB in total,
- 10 GB object storage,
- 10 GB archive storage.

Offered free sources provide more than twice the capacity compared to *Amazon Web Services (AWS)* (source: <https://www.oracle.com/cloud/free/>).

For this book, we will use *Oracle Cloud Always Free* option, which is implemented inside the *Oracle Cloud Free Tier*, which provides you 30-day Free Trial. The 300\$ free credits limit the trial version, access to the wide range of Oracle Cloud services during the trial period (containing Databases, Analytics, Compute and Container Engine for Kubernetes), up to 8 instances across proposed services, and up to 5TB of storage.

So much for an introduction. Let's get started creating an account and registering. We will use the Oracle Cloud Infrastructure for studying SQL and procedural language PL/SQL, so we will primarily use the Transaction Processing Database type.

Oracle Cloud Always Free option is available at the following web address: <https://www.oracle.com/cloud/free/>



Fig. 1.2: The QR code of the Oracle Cloud website

We will drive you through the whole process. To start the registration process, please click on the “Start for free” button (fig. 1.3):

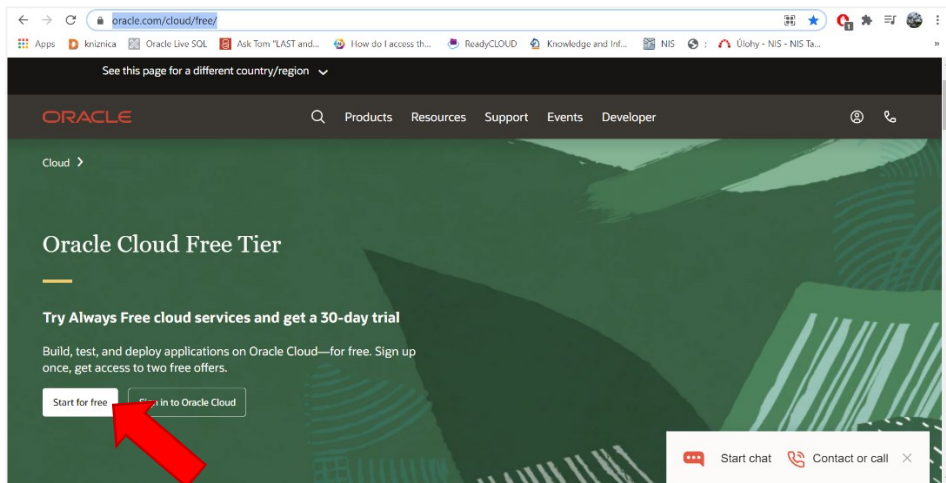


Fig. 1.3: Oracle Cloud – Free Tier, step 1

In the next screen (fig. 1.4), provided resources are summarized in the left part. The right part consists of your registration information, namely: country, first name, surname, and contact e-mail. Please fill in the required inputs.

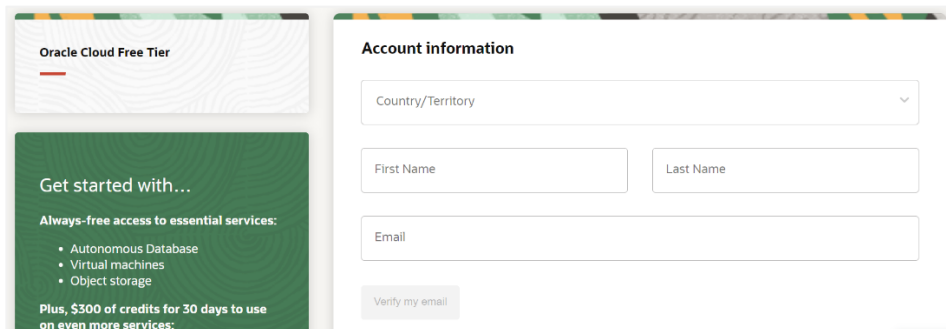


Fig. 1.4: Oracle Cloud – Free Tier, step 2

Then, an e-mail confirmation will be sent to you.



Fig. 1.5: Oracle Cloud – Free Tier, step 3

You have 2 minutes to verify the e-mail account by clicking on the link you receive there.

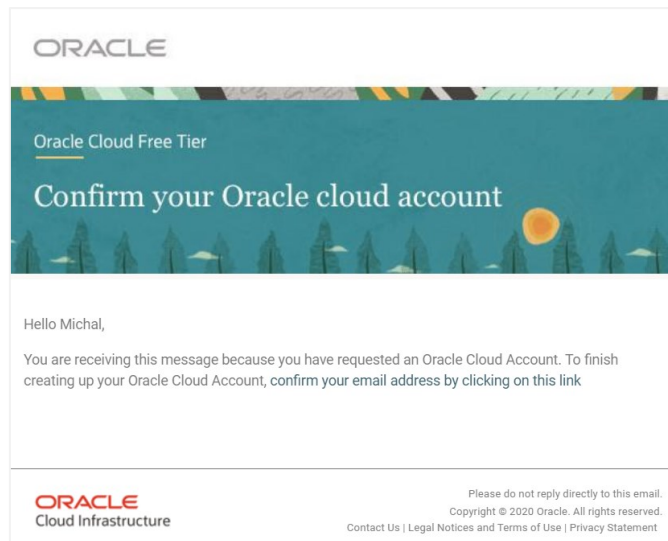


Fig. 1.6: Oracle Cloud – Free Tier, step 4

Then, you will be navigated to the web address, where the password, company, and home region should be specified. Note that the password should be strong enough. It must contain a minimum of 8 characters. At least one of them should be lowercase, uppercase, and special character (except spaces, ~, <, >, or \). The password cannot contain your first name, surname, or e-mail address for security reasons. The size limit of the password is 40 characters.

Cloud account name will be generated and provided to you, so please remember such value (it can be changed for any available unique value).

The *Home region* defines the geographic location of the *Oracle Cloud* provided to you. There, the resources will be created and allocated. Note that it is not possible to change it after the registration process. For Central Europe, two choices are recommended – *Netherlands Northwest (Amsterdam)* or *Germany Central (Frankfurt)*. I will use the Frankfurt location. However, the selection is total, up to you.

The image shows a registration form for an Oracle Cloud Free Tier account. It contains three input fields: "Company Name" with the value "Zilinska univerzita v Ziline", "Cloud Account Name" with the value "kvetmichal40", and "Home Region" with the value "Germany Central (Frankfurt)". The "Company Name" field is marked as "Optional". Below the form, there is a note: "Your home region is the geographic location where your account and identity resources will be created. It is not changeable after sign-up. See Regions for service availability."

Fig. 1.7: Oracle Cloud – Free Tier, step 5

By clicking on the *See Regions* link (<https://www.oracle.com/cloud/data-regions/#emea>), it is possible to get more specific information about the available sources in each destination. Currently, all *Always Free Cloud Services* are accessible at each location. For these labs, *Oracle Autonomous Transaction Processing*, *Oracle Autonomous Data Warehouse*, and *Oracle Cloud Infrastructure Object Storage* will be inevitable, supervised by the *Oracle Application Express (APEX)* and *SQL Developer Web*.

Then, the address is required. The account is verified by using a mobile phone. Follow the instructions and provide the system required codes. The system requires your bank account. Do not worry; no charges will be applied. It is used for the possibility to transfer the *Always Free account* to the more complex charged, anytime based on your requirement.

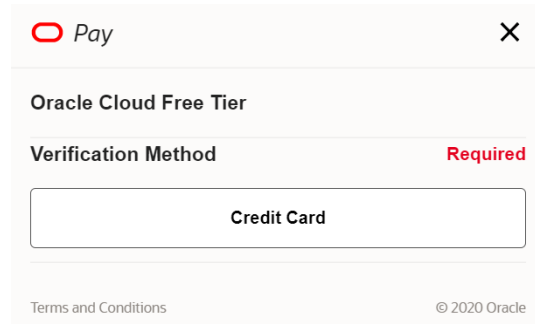
The screenshot shows a modal window titled "Pay" with a close button (X) in the top right corner. Below the title bar, the text "Oracle Cloud Free Tier" is displayed. Underneath, there is a section labeled "Verification Method" with a red "Required" status indicator to its right. A button labeled "Credit Card" is centered within a rectangular frame. At the bottom of the modal, there are two links: "Terms and Conditions" on the left and "© 2020 Oracle" on the right.

Fig. 1.8: Oracle Cloud – Free Tier, step 6

Finally, read the agreement carefully, confirm it and register your account by clicking on the “Start my free trial” button (fig. 1.9):

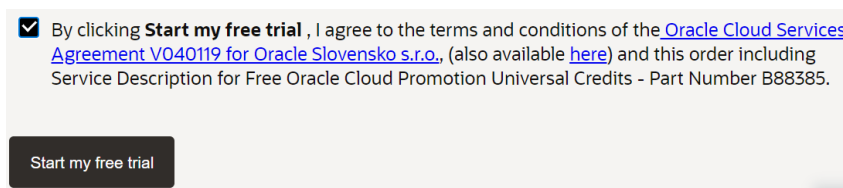
The screenshot displays a checkbox that is checked, followed by the text: "By clicking **Start my free trial**, I agree to the terms and conditions of the [Oracle Cloud Services Agreement V040119 for Oracle Slovensko s.r.o.](#), (also available [here](#)) and this order including Service Description for Free Oracle Cloud Promotion Universal Credits - Part Number B88385." Below this text is a dark button labeled "Start my free trial".

Fig. 1.9: Oracle Cloud – Free Tier, step 7

Now, your account is created so that you can enjoy the robustness of the cloud services. Please, logon to the system (<https://www.oracle.com/cloud/free/>):



Fig. 1.10: The QR code of Oracle Cloud – Always Free login page

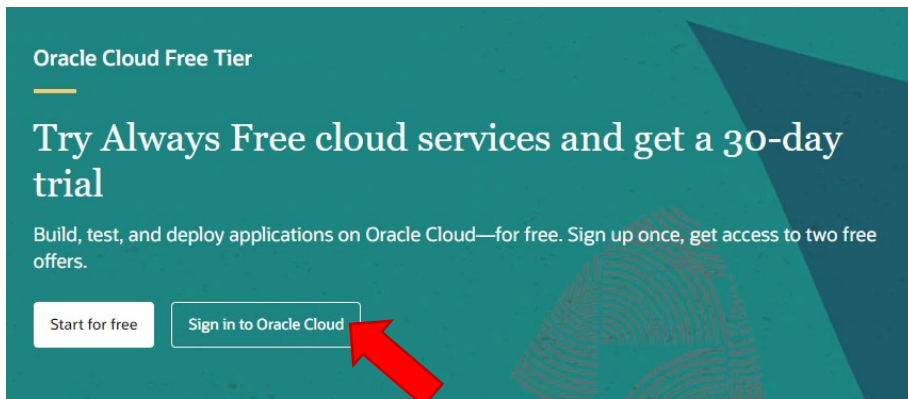


Fig. 1.11: Oracle Cloud – Free Tier, step 8

Specify your Cloud Account Name – it has been generated during the registration process. In my case, the Cloud Name is “kvetmichal”.

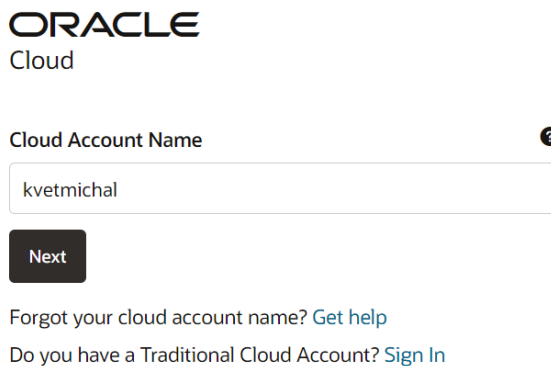
The image shows the Oracle Cloud sign-in form. At the top is the "ORACLE Cloud" logo. Below it is a label "Cloud Account Name" with a help icon. A text input field contains the value "kvetmichal". Below the input field is a "Next" button. At the bottom, there are two links: "Forgot your cloud account name? Get help" and "Do you have a Traditional Cloud Account? Sign In".

Fig. 1.12: Oracle Cloud – Free Tier, step 9

Click on the “Next” button and write your username and password specified during the registration, as well. The username is the same as the e-mail account address (fig. 1.13).

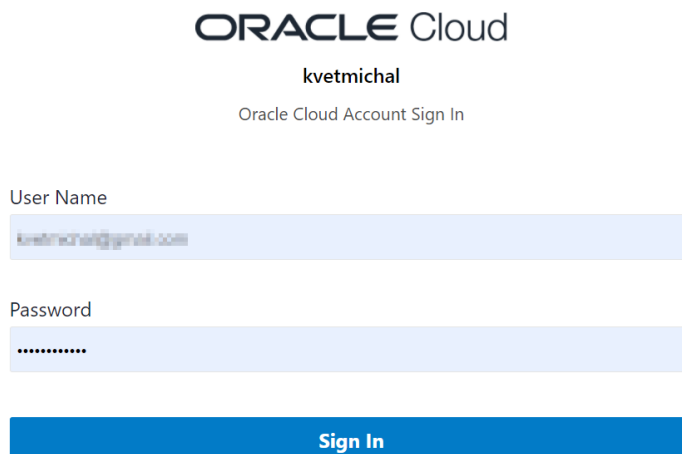
The image shows the Oracle Cloud sign-in form with the "Next" button clicked. The "ORACLE Cloud" logo is at the top. Below it, the text "kvetmichal" is displayed, followed by "Oracle Cloud Account Sign In". There are two input fields: "User Name" with the value "kvetmichal@gmail.com" and "Password" with masked characters "*****". At the bottom is a blue "Sign In" button.

Fig. 1.13: Oracle Cloud – Free Tier, step 10

Click on the “Sign In”. You are now connected to the Cloud. You will be navigated to the main dashboard screen. At the top of the screen, resources, services, or documentation can be searched. Then, the geographical location of the used Cloud is specified (in my case, it is *Germany Central (Frankfurt)*). Then, some notifications can be present. Language can be set to your mother language (Slovak or Czech language is also available). For this book, we will use the English language.

The last button reflects your profile information.

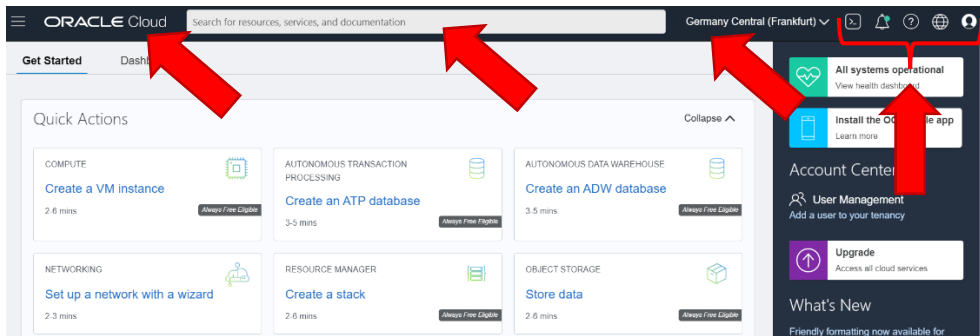


Fig. 1.14: Oracle Cloud – Home Screen

Home screen dashboard can always be obtained by clicking on the Oracle Cloud logo button.

Profile info contains your identification, tenancy, user settings, etc.

Profile

[oracleidentitycloudservice/identitycloudservice.com](#)

Tenancy: kvetmichal

[User Settings](#)

[Sign Out](#)

Fig. 1.15: Profile

Note that the web address always consists of the region specification.

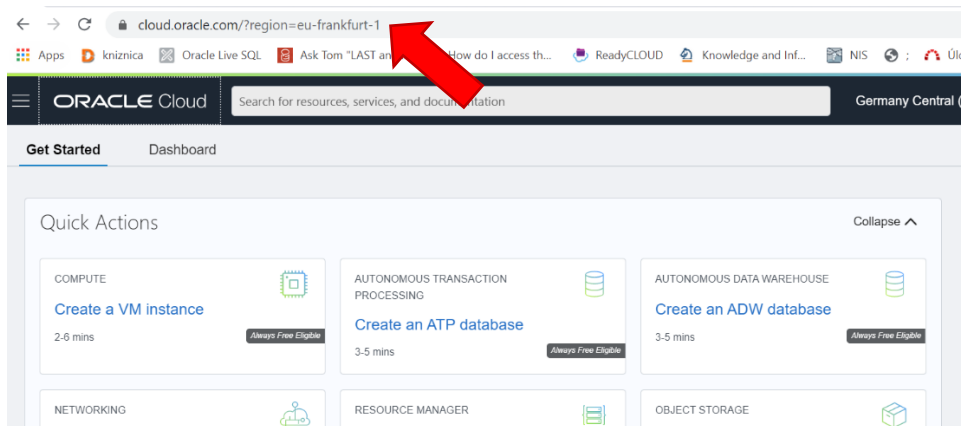


Fig. 1.16: Web address – region

Note that many sources, recommendations, examples, and discussions are available in *Free training from Oracle University* (<https://cloud.oracle.com/?tile=get-started-oracle-university®ion=eu-frankfurt-1>). In addition, there are also *key concepts and terminology* (<https://cloud.oracle.com/?tile=key-concepts-terminology®ion=eu-frankfurt-1>), *introduction to APEX* (<https://cloud.oracle.com/?tile=intro-to-apex®ion=eu-frankfurt-1>) or *Resource Manager* (<https://cloud.oracle.com/?tile=intro-resource-manager®ion=eu-frankfurt-1>). Such sources can be located in the bottom part of the *Home screen*.

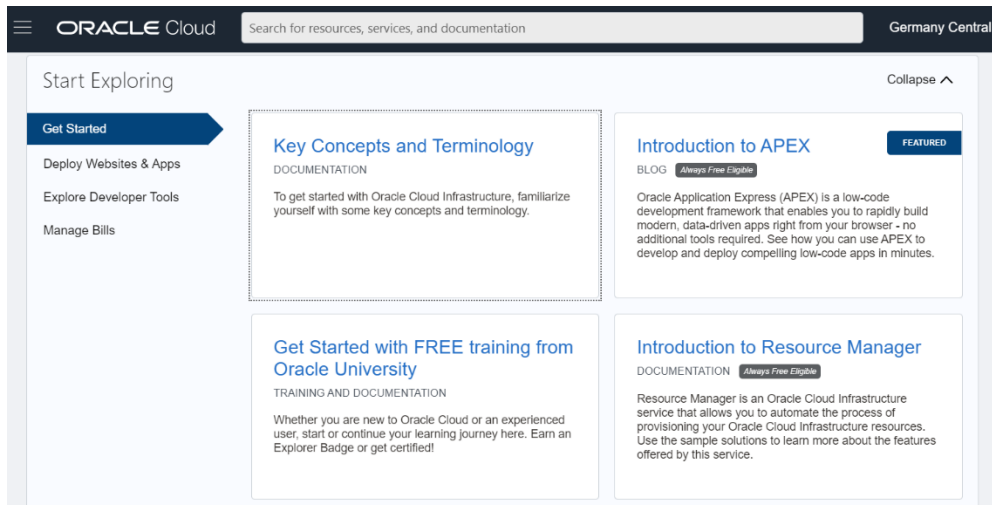


Fig. 1.17: Exploring sources

Before we start with the database creation itself, it is necessary to list some used terminology (note that the whole terminology can be found directly in the specified location, we will name just the most important ones relevant for this book):

Region, availability domain

The *region* is a geographical location from which the resources are provided (e.g., virtual cloud network). Each region consists of at least one availability domain (supervising, e.g., compute instance). Each *availability domain* is independent, isolated from other domains, fault-tolerant. Thus, configuring multiple availability domains can ensure high availability and failure resistance.

Realm

The *realm* is a logical collection of regions. Each *realm* is isolated does not share any data with any other. The *tenancy* is associated with just one realm and has access to the region set belonging to the realm.

Console

Cloud console is a web source application providing access and management of the *OCI*.

Tenancy

The *tenancy* is a specific cloud repository, usually devoted to the organization or company providing secure and isolated storage and processing partition. You can manage, create and associate cloud resources and services across the tenancy.

Compartment

The *compartment* comprises cloud resources (instances, virtual cloud networks, etc.) with specific privileges and quotas. It is rather a logical unit as a physical container.

Note that Oracle provides you a tenancy after the registration, which is a root compartment holding and managing all provided cloud resources. Then, you can create a resource categorization tree. Each resource is associated with the compartment by definition. The core principle is based on granting users only resources inevitable for their work, no more.

Virtual Cloud Network (VCN)

VCN is a virtualized network of the conventional network, including subnets, routers, gateways, etc. It is located within one *region* and can spread multiple *availability domains*.

Instance

Instance is a compute host running in the cloud. Its main advantage is flexibility. You can utilize the sources (physical hardware) on-demand to ensure performance, high availability, robustness, to pass your set security rules.

Image

Image is a specific template covering the operating system and other software installed. In addition, Oracle provides you with several virtual hard drives applicable to the cloud system, like Oracle Linux, CentOS, Ubuntu, or Windows Server. The list and specification can be found in the documentation:

<https://docs.oracle.com/en-us/iaas/Content/Compute/References/images.htm>.

Storage management

Storage management is an inevitable part of data processing and resisting and accessing. **Block volume** is defined as a virtual hard drive providing persistent data storage space. Their principles are similar to the hard drives in ordinary computers. It is possible to attach or detach it on demand, even to another instance, without any data or application loss. **Object storage** is a storage repository architecture available and accessible via web interface anywhere. Physical data can have any structure and any type. The size is limited to 50 GB per file. *Object storage* is a standard repository for backups or large data objects, which are not commonly changed very often. The **bucket** is a lower architectural definition. It is denoted as a logical container within the *Object storage*. Several buckets can be present in any *Object storage*. The amount of the data in size and count aspect is unlimited.

Now, it is time to create the database and enjoy cloud resources.

Database creation

Please, connect to the *Home screen* of the cloud. There is a list of technologies and resources available to you in the *Quick actions* menu (fig.1.18).

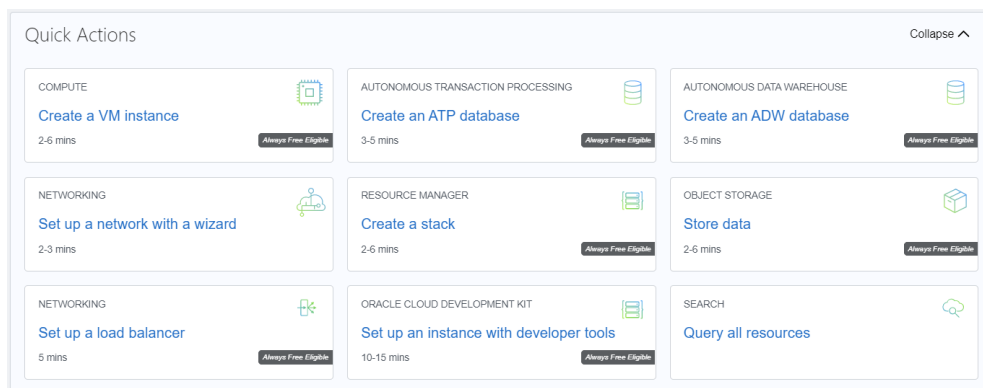


Fig. 1.18: Quick actions

Each name delimits the element, category, estimated time consumption for the creation and mark, whether such resource is available in the *Always Free option*, or the specific licensing is necessary. In this phase, we will create two *Autonomous Transaction Processing (ATP)* databases. One will be used for the *Student* model. The second one will be used later to deal with the *Library*. So click on the “Create an ATP database” button:

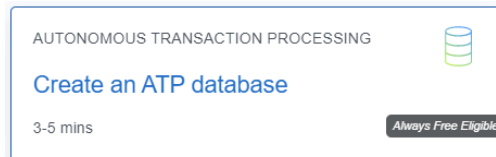


Fig. 1.19: Autonomous transaction processing

You will be navigated to the new window with the database parameter specification. You have to define a *compartment* (you have just one, so the pre-selected variant is suitable). Then, there is an input for the *Display Name* – user-friendly database name for easy identification and *Database Name* (it can contain only letters and numbers. The first character should be a letter. Note that the maximum size is 14 characters).

Provide basic information for the Autonomous Database

Compartment

kvetmichal (root)

Display name

database for STUDENT mode|

A user-friendly name to help you easily identify the resource.

Database name

studentDB

The name must contain only letters and numbers, starting with a letter. Maximum of 14 characters.

Fig. 1.20: Autonomous database definition

Then, the *Workload Type* should be selected. It depends on future usage. *Data Warehouse (DW)* is suitable for complex evaluation and analytics, where the main emphasis is taken on the data retrieval process. The amount of data stored inside is enormous. *Update* operations are not present or are rare. In *DW*, data are loaded in batches, and processing such a procedural is not time crucial.

In contrast, the system contains several indexes, and also denormalized tuples and values can be present to improve data retrieval efficiency, which is mainly highlighted. *Transaction Processing type* is suitable for online short-running queries and transactions. It is based on the data normalization and high concurrency of the processing. Built-in *JSON type representation* is mainly associated with the document *API* or storage management in a *JSON format* style. *JSON type* is now available in the *Free Tier*, as well. *APEX workload type* is characterized by the storage for the APEX application development – data-driven system creation and deployment. For this study, we will select the *Transaction Processing type*.

Then, select the deployment type, which can be either shared or dedicated. We will use *Shared Exadata infrastructure*, which is free in our option (see Fig. 1.21).

Then, configure the database (*OCPU* and *storage capacity*), all options are pre-selected. As already stated, the *Always Free* option is limited to one *OCPU* and 20 GB of storage (valid at the time of writing this book). If paid option is chosen, the *Autoscaling option* can be used, as well, by altering the system sources up to three times of the provisioned cores and resources if the workload demands rise.

Choose a workload type

Data Warehouse Built for decision support and data warehouse workloads. Fast queries over large volumes of data.	Transaction Processing Built for transactional workloads. High concurrency for short-running queries and transactions. ✓	JSON Built for JSON-centric application development. Developer-friendly document APIs and native JSON storage.	APEX Built for Oracle APEX application development. Creation and deployment of low-code applications, with database included.
----------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------

Choose a deployment type

Shared Infrastructure Run Autonomous Database on shared Exadata infrastructure. ✓	Dedicated Infrastructure Run Autonomous Database on dedicated Exadata infrastructure.
---------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------

Fig. 1.21: Database parameter definition

When writing this book, the current database version is *21c*. However, feel free to use the newest one. It is also possible to choose at least one older version due to compatibility reasons. Make sure that the *Always Free* selection is chosen.

Configure the database

Always Free ⓘ
☒ Show only Always Free configuration options

ⓘ If your Always Free Autonomous Database has no activity for 7 consecutive days, the database will be automatically stopped. Your data will be preserved, and you can restart the database to continue using it. If the database remains stopped for 3 months, it will be reclaimed. [Learn more](#).

Choose database version
 21c

ⓘ Free Tier Autonomous Databases using Oracle Database 21c cannot currently be upgraded to paid instances.

OCPU count *Read-Only*
 1
Always Free Autonomous databases can utilize up to 1 core. The CPU core count cannot be adjusted.

Storage (TB) *Read-Only*
 0.02
Always Free Autonomous databases can utilize up to 0.02 TB (20 GB) of storage. The storage size cannot be adjusted.

☐ Auto scaling
Allows system to use up to three times the provisioned number of cores as the workload increases. [Learn more](#)

Fig. 1.22: Database version selection

The next step is to set administrator credentials. For the *Always Free* option, the *username* is *ADMIN*, and it cannot be edited. Please, specify the password twice. Note that the password must contain at least 12 characters (by not more than 30). It must have at least one uppercase, one lowercase, and one number inside for security reasons. Password cannot contain double quotes (“) or the word “admin” (or username generally).

Create administrator credentials ⓘ

Username *Read-Only*
 ADMIN
ADMIN username cannot be edited.

Password

Confirm password

Fig. 1.23: Database administrator credentials

Follow the instructions and specify network access, which can be generally limited to the IP address range. In our case, select the general option “*Allow secure access from everywhere*”.

Choose network access

Access Type

Secure access from everywhere
Restrict access to specified IP addresses and VCNs. ✓

Private endpoint access only
Restrict access to a private endpoint within an OCI VCN.

☐ Configure access control rules ⓘ

Fig. 1.24: Network access definition

Finally, choose the “*License Included*” option, whereas you do not have your own licensing.

Choose a license type

Bring Your Own License (BYOL)
Bring my organization's Oracle Database software licenses to the Database service. [Learn more.](#)

License Included
Subscribe to new Oracle Database software licenses and the Database service.

Fig. 1.25: Licensing

At the bottom, you can optionally *Show Advanced Options* to define tags allowing you to organize and track resources in the tenancy. Such an element is not attractive for us.

End the specification process by clicking on the “*Create Autonomous Database*” button. Now, it is almost done. Just wait a few minutes until the database and resources are provisioned.

Create Autonomous Database [Cancel](#)

Fig. 1.26: Create autonomous database

Now, you should see the following screen. The abbreviation of the database type is present in the left part – *ATP* representing *Autonomous Transaction Processing*. The current status is below the signature, now shown in orange color (fig. 1.27) representing the provisioning process.

ORACLE Cloud Search for resources, services, and documentation Germany Central (Frankfurt) > ⓘ ? ⓘ ⓘ ⓘ ⓘ

database for STUDENT model Always Free

DB Connection Performance Hub Service Console Scale Up/Down More Actions ▾

Autonomous Database Information Tools Tags

General Information

Database Name: studentDB
Workload Type: Transaction Processing
Compartment: kvt3 (root)
OCID: ...lata5a [Show](#) [Copy](#)
Created: Tue, Jun 29, 2021, 05:29:37 UTC
CPU Count: 1
Auto Scaling: Disabled ⓘ
Storage: 20 GB
License Type: License included
Database Version: 21c
Lifecycle State: Provisioning
Instance Type: Free [Upgrade to Paid](#) ⓘ
Mode: Read/Write [Edit](#)

Infrastructure

Dedicated Infrastructure: No

Autonomous Data Guard ⓘ

Status: Disabled ⓘ

Backup

Last Automatic Backup: No active backups exist for this database.
Manual Backup Store: Not Configured

Network

Access Type: Allow secure access from everywhere
Access Control List: Disabled [Edit](#)

ATP

PROVISIONING

Fig. 1.27: Provisioning

Provisioning is a process of creating and associating resources. In the right part, the summary is present consisting of the *database name*, *workload type*, *compartment*, and *database system parameters*.

Wait approximately 3-5 minutes until the database is available by replacing the status with the “Available”, denoted by the green color (fig. 1.28). Now, the system is available for management and processing.

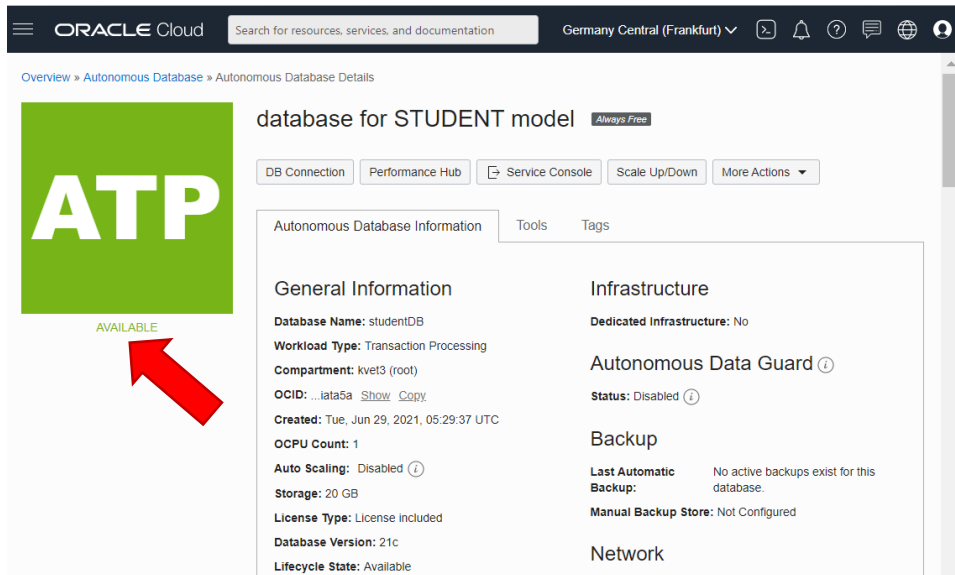


Fig. 1.28: Database availability status

The above screen shows the home screen for the created *StudentDB* database. The bottom part contains usage resource statistics and metrics – *CPU Utilization*, *Storage Utilization*, *Sessions*, *Execute Count*, *Running statements*, and *Queued Statements*. Returned results can be filtered out based on the time intervals. Now, the graphs are empty, whereas no activity has been done.

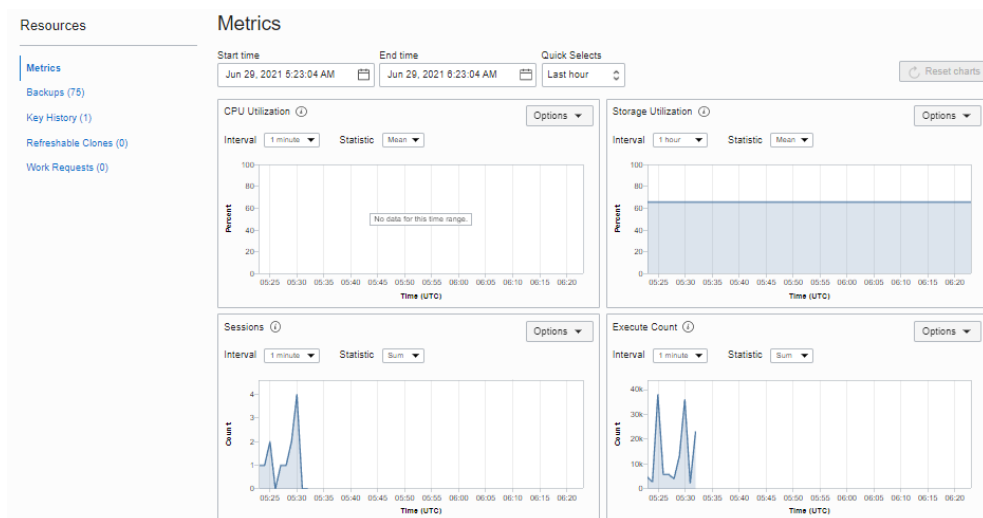


Fig. 1.29: Resource statistics

So, let's return to the database *Home screen*. In the upper part, several buttons and tabs are present, which will be consecutively described.

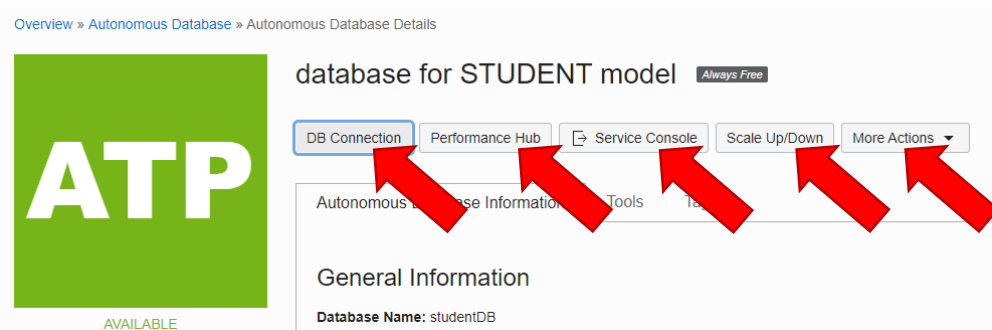


Fig. 1.30: Database home screen

DB Connection provides you the client credentials and connection information to connect to the cloud database. In addition, it will offer you the zipped file consisting of the *Client Credentials (Wallet)* in an encrypted manner. We will use it to connect the *SQL developer client* environment launched locally in the client computer.

Performance Hub consists of extended statistics monitoring activity like average active sessions, *SQL monitoring*, *Automatic Database Diagnostic Monitor (ADDM)*, *Workload*, *Blocking sessions*, etc.

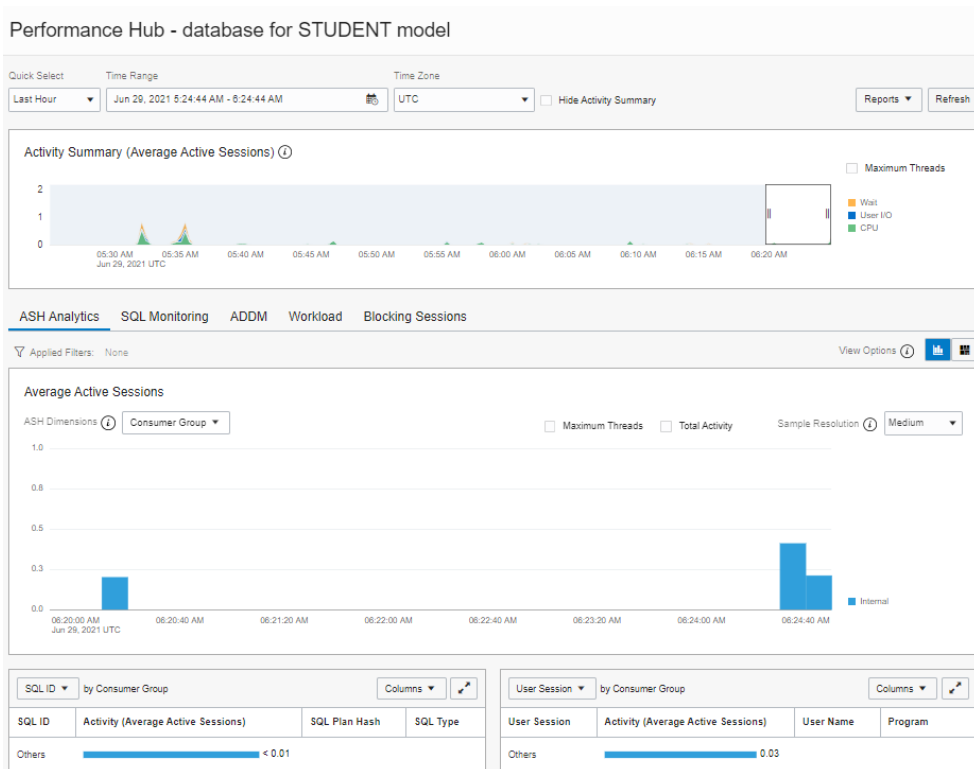


Fig. 1.31: Performance hub

Service Console provides you with the statistics needed for the administration. It focuses on the activity and administration parameter definition. The *service console* is always associated with the defined database. At the *Administration level*, the *wallet* can be downloaded, *resource management* rules can be set, and *administrator password* can be set. There, *Oracle Machine Learning users accounts* can be specified and managed. The form to provide feedback to *Oracle* is present there, intending to give you the best services and experience. The development part of the *Service Console* allows you to download *Oracle Instant Client*, download *SODA drivers*, access the *Oracle APEX*, *SQL Developer Web*, *RESTful services*, etc.

Oracle Instant Client is a set of tools, libraries, and SDKs for building and connecting applications. *SQL*Plus* tool is present there, and *import* and *export* functionality is executed either in the client-side or cloud instance. Oracle strongly recommends using *Data Pump functionality* for data import and export, which is done on the server-side, so there are no additional demands on the internet connection. Thus, a large data amount can be processed. Even if the connection fails during the execution, whereas the whole process is done in the cloud environment, the client just supervises the activity and progress. Libraries provide a layer for the API interface of various languages – *PHP*, *Python*, *Node.js* and access for *OCI*, *OCCL*, *JDBC*, *ODBC*, and *Pro*C* applications.

We will use *Oracle Instant Client* for the access and import activity.

Simple *Oracle Document Access (SODA)* is a set of *APIs* for managing *JSON documents* in the Oracle database. Drivers are available for *Java*, *C*, *PL/SQL*, *Python*, *Node.js*, and *REST*.

Oracle APEX is a low code application tool providing the environment for the data-driven web application definition. The created application can be directly deployed in the cloud environment. Access is then done via the web browser. It is optimized for desktop and mobile systems, as well.

SQL Developer Web provides a web-based interface for the object and data definition and management, as well as the administration of the *Oracle Autonomous Database*. We will use *SQL Developer Web* and a desktop variant of the product installed on the client-side.

Scale Up/Down [Help](#)

Information To access all Autonomous Database features, upgrade the Autonomous Database instance to paid.

[Upgrade Instance to Paid](#)

OCPU count Read-Only
1
Always Free Autonomous databases can utilize up to 1 core. The CPU core count cannot be adjusted.

Storage (TB) Read-Only
0.02
Always Free Autonomous databases can utilize up to 0.02 TB (20 GB) of storage. The storage size cannot be adjusted.

☐ **Auto Scaling**
Allows system to use up to three times the number of cores specified by the OCPU count as the workload increases. [Learn more](#)

[Close](#)

Fig. 1.32: Scalability definition

Scale-Up/Down allows you to react dynamically to the processing demands and workload and optimize processing unit amount and storage demands to provide robust

and performance-resistant solutions. *Automatic scalability* can be applied there, as well. In that case, the number of *CPUs* can change dynamically up to *3 times* the defined limit. In the *Always Free option*, particular settings cannot be done, and only *1 OCPU* and *20 GB* of storage can be used free.

The *More Actions* combo box button groups several activities, which can be done for the defined database, like *Start*, *Stop*, *Restart*, *Clone*, *Rename*, or *Terminate*. *ATP* version can be converted to the *Autonomous JSON database*. Reflecting the licensing, the license type can be enhanced, or the management can be extended by shifting the option to the paid type.

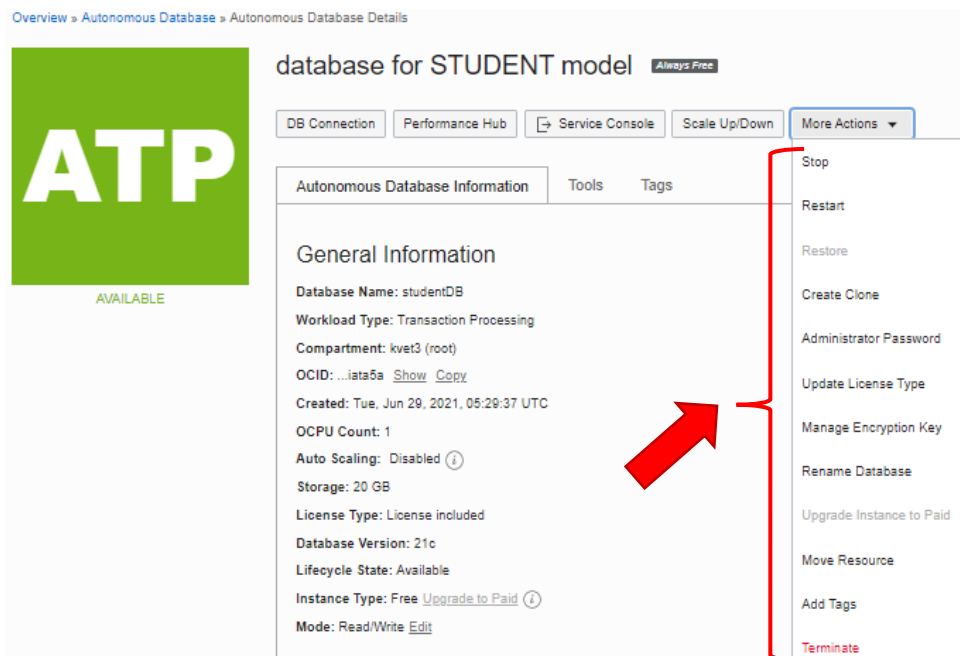


Fig. 1.33: Available action list

Now, let's describe the available tabs in the database *Home* screen. *Autonomous Database Information* consists of the instance and database summary. *Tools* tab provides you connection to the accessible application directly in the web browser. Namely, *SQL Developer Web* is a suitable solution for the data and object definition and enhancements and for administering the database and instance itself. *Oracle Application Express (APEX)* can be launched from such a repository, too. There are also *Oracle Machine Learning (ML) User Administration* tools and *SODA Drivers* modules.

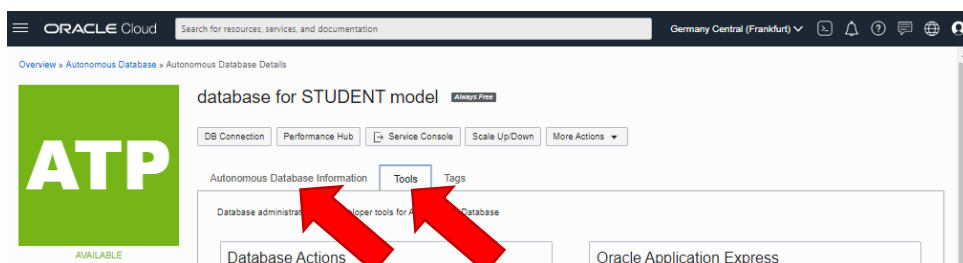


Fig. 1.34: Main database administration tab

Now, choose the *Tools* tab and start *SQL Developer Web* by clicking on the *Open Database Actions* button of the *Database Actions* type.

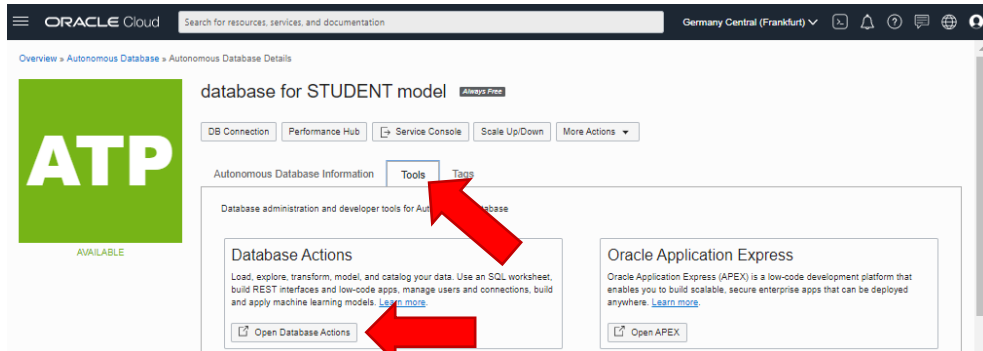


Fig. 1.35: Launching *SQL Developer Web* (1)

A new browser tab will be opened by requesting the username and password. In our case, we will specify administrator user, set during the database definition and applied in the provisioning process.

 This is a login form titled 'ORACLE Database Actions'. It features a 'Username' input field and a blue 'Next' button below it.

Fig. 1.36: Launching *SQL Developer Web* (2)

Thus, the username will be “*admin*” and use the password specified in the database definition.

 This is the same login form as in Fig. 1.36, but now the 'Username' field is filled with 'ADMIN'. The 'Password' field is also present with masked characters. A blue 'Sign in' button is at the bottom.

Fig. 1.37: Launching *SQL Developer Web* (3)

Note, that the standard user is not allowed to access *Oracle Developer Web*. The privileges can be maintained by the following script executed by the *admin*. Parameter *p_schema* and *p_url_mapping_pattern* references the *username* of the particular user, in the following case, the name is “*Michal*”.

```

begin
  ords_admin.enable_schema
  (p_enabled => TRUE,
   p_schema => 'MICHAL', -- username for the grant
   p_url_mapping_type => 'BASE_PATH',
   p_url_mapping_pattern => 'michal',
   p_auto_rest_auth => NULL
  );
  commit;
end;
/

```

Click on the *Sign in*. Now, the *SQL Developer Web* environment is provided. There are four categories there:

- *Development* – consisting of the *SQL definition* environment, *Data modeler*, *JSON*, *REST*, and *APEX*.
- *Data tools* – consisting of the tools for the data loading, catalog to understand object dependencies, data insights, and business models.
- *Administration* – user and privilege management.
- *Monitoring* – performance analysis, statistics.

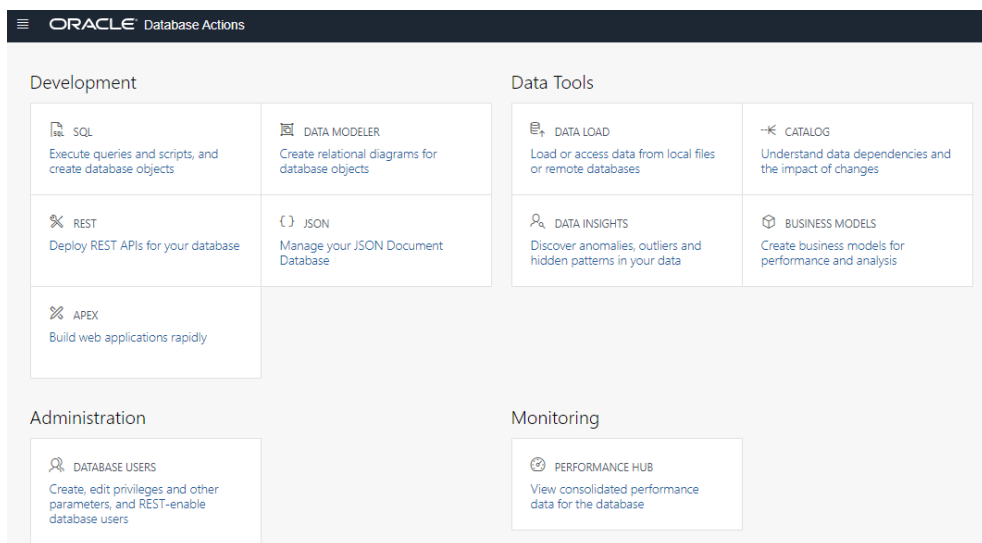


Fig. 1.38: *SQL Developer Web – main screen*

Click on the *SQL*. Such an environment allows you to specify and execute *SQL* commands.

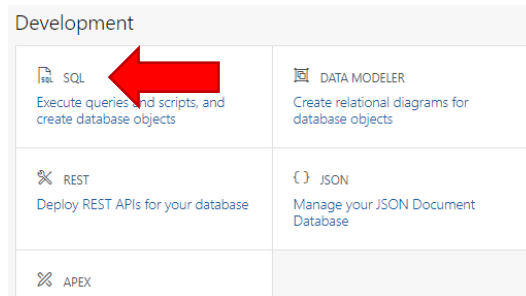


Fig. 1.39: SQL definition module

The environment consists of three parts. The left part reflects individual *objects'* navigation and allows you to search for structure and stored types. The upper part is used for the *SQL statements definition, result set*, or the *information summary* is then in the bottom part.

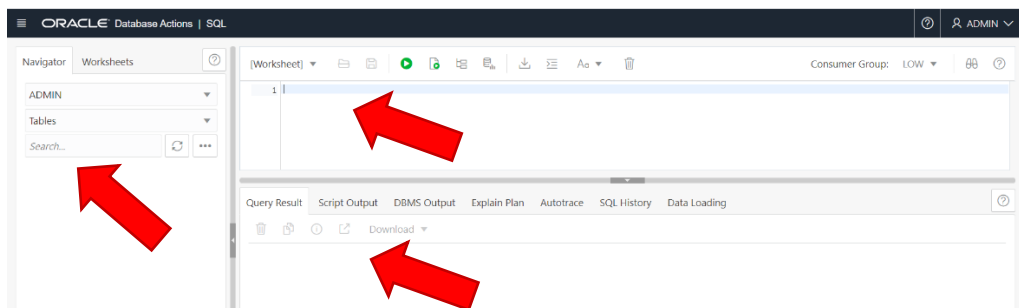



Fig. 1.40: SQL definition environment

Let's write your first *SQL statement* obtaining the current server date:

```
select sysdate from dual;
```

Sysdate is a function call providing you the server date and time. Note that it gives *Date* data type. In *DBS Oracle*, it always consists of the *Date* and *Time* elements, as well. So be aware of it when querying!

The *dual* table is a specific table present in the Oracle database. It has only one attribute called *Dummy*, and the value is *X*. Its owner is the user *SYS*, and each user can select data from it. It cannot be modified, and it is used for obtaining function results. Thus, it produces the result just in one row.

Execute the query by clicking on the “Run statement” button  or by pressing the shortcut *CTRL+Enter*. The produced result set is above the script in a separate part.

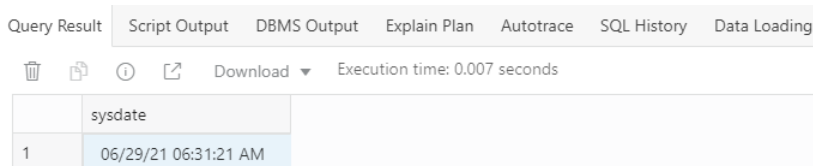



Fig. 1.41: Query result

Alternatively, the whole script can be executed by clicking the “Run script”  or pressing *F5*.

```
select sysdate from dual;  
  
select user from dual;
```

The script result:

```
SYSDATE  
-----  
2021-06-29T06:32:22Z  
  
Elapsed: 00:00:00.003  
1 rows selected.  
USER  
-----  
ADMIN  
  
Elapsed: 00:00:00.007  
1 rows selected.
```

Note that the function call *User* provides you the login of the currently connected session user. In our case, we are connected as user “*Admin*”.

SQL Developer, either in the web or desktop version, proposes a robust, user-friendly environment for the *SQL* or procedural language (*PL/SQL* definition), administration, etc. Moreover, it allows you to create a data model, either manually or reverse engineering, to analyze existing systems and data dictionary.

SQL Developer can be launched locally in the desktop environment. In that case, you can download the tool from the official site of *Oracle*. When writing this material, the newest version is *SQL Developer 21.4.3*. However, feel free to download the most up to date here: <https://www.oracle.com/tools/downloads/sqldev-downloads.html>.



Fig. 1.42: *SQL Developer installation link*

There are several versions for individual operating systems, so choose the best suitable based on your environment. In the case of using *Windows*, I recommend downloading the version including the *Java Development Kit (JDK)*; otherwise, you have to install it manually. *SQL Developer* is written in *Java* and does not need to be installed. Just download the archive, extract it and store it in the file system. There is a file “*SQL developer*”, by which the application is launched. So, download it and run.

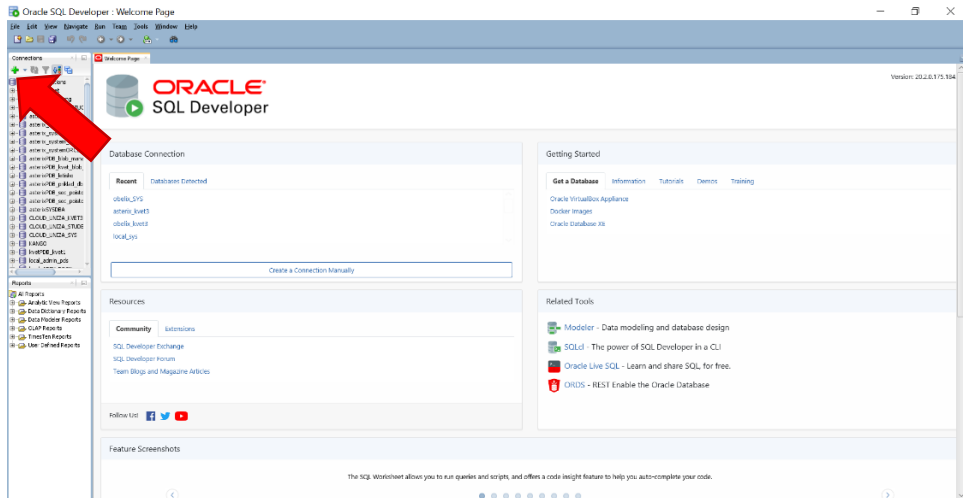


Fig. 1.43: SQL developer environment

Before we start, connecting to the cloud instance is necessary to access the sources and database itself.

Click on the “+” button in the left part (in the *Connection list*). The new window is launched, requesting you to provide connection details. As already stated, credentials to the cloud instance can be obtained by downloading the wallet from the *Cloud console* or *Home screen*. Return to the database *Home screen* of the cloud environment. In the left panel button list, select the *Oracle Database* and *Autonomous Transaction Processing*.

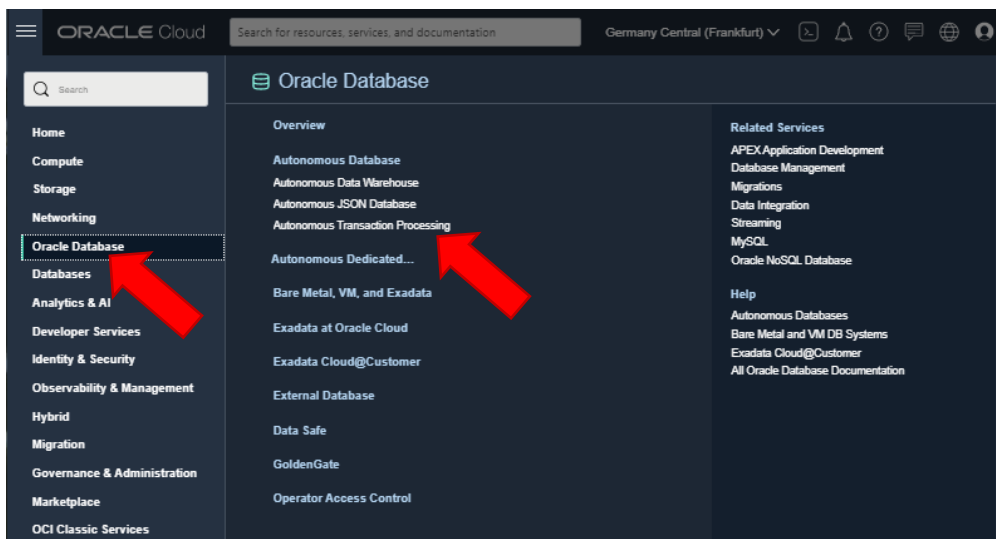


Fig. 1.44: Cloud menu

Choose the relevant database to be connected (in our case, we have just one database to be reached). Click on its description (*Display name*).

Create Autonomous Database							
Display Name	State	Dedicated	OCPU	Storage (TB)	Workload Type	Autonomous Data Guard	Created
database for STUDENT model <small>Always Free</small>	● Available	No	1	0.02 TB	Transaction Processing	—	Tue, Jun 29, 2021, 05:29:37 UTC

Fig. 1.45: List of available databases

Then click on the *DB Connection* button and follow the instructions.

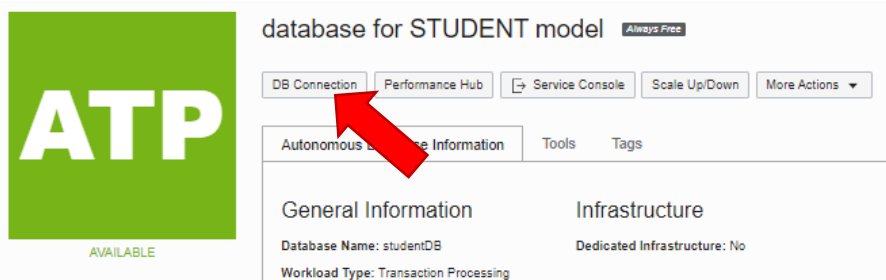


Fig. 1.46: Obtaining DB Connection

Set the *Wallet type* to *Instance wallet* and just click on the *Download Wallet*. *Instance wallet* is used for the specific database connection details definition, whereas the other option (*Regional wallet*) covers all databases and is used for administration purposes.

Database Connection [Help](#)

You will need the client credentials and connection information to connect to your database. The client credentials include the wallet.

Download Client Credentials (Wallet)

To download your client credentials, select the type of wallet, then click Download Wallet. You will be asked to create a password for the wallet.

Wallet Type ⓘ

Instance Wallet

Wallet last rotated: -

Fig. 1.47: Download wallet

A connection always uses a secure type. This file will be necessary to be associated with the *SQL Developer* desktop connection details. Whereas some database clients will require a wallet and password to your database, please specify the password twice and download it.

The name of the downloaded file archive contains two-part – keyword “*wallet*” and identification of your database name. In my case, the name is “*Wallet_studentDB*” with a .zip extension. You do not need to extract such a file. The whole repository is associated with the connection directly. However, looking at the storage internally, the downloaded wallet consists of several files. I will just mention the most relevant for this book:

- *Tnsnames.ora* – connection details – protocols, hosts, ports, etc. Note that the cloud connection is enhanced by the five categories reflecting the importance (low, medium, high, tp, and urgent).
- *Sqlnet.ora* – wallet location and encryption types.
- *Ewallet* – encryption wallet details.

1.1 SQL Developer connection specification

Return to the *SQL Developer* and define new connection parameters.

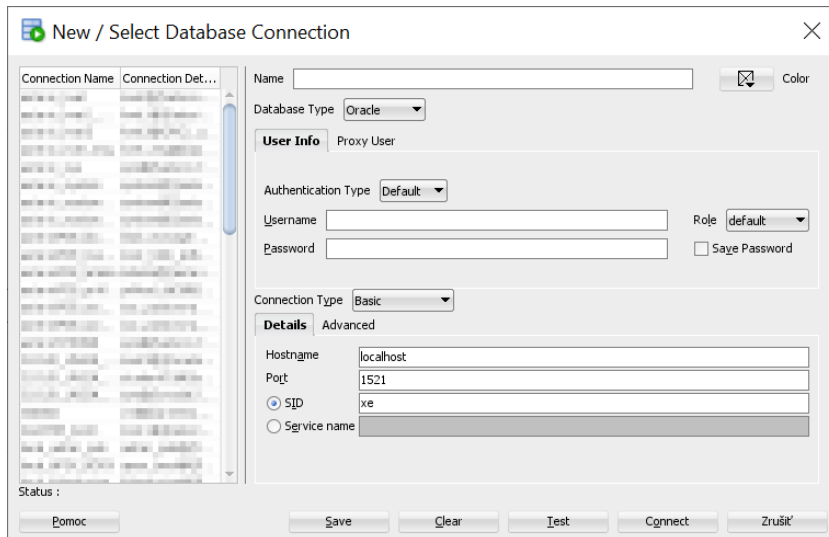


Fig. 1.48: *SQL Developer connection definition (1)*

Connection name is necessary to be specified, by which it can be easily located in the *Connection list*. The name is up to you. *Database Type* is *Oracle*. Optionally, you can install several drivers to connect to *MySQL*, *Postgres*, etc. Let's leave the *Authentication Type* to *Default*. *Username* is the login of the user associated with the database. In our case, we will use user “*admin*”. Define the password specified during the database definition and provisioning. Optionally, you can store the password by using the checkbox. Let remain the *Role* to the *Default* value (you do not have granted particular privilege group, like *SYSDBA*, *SYSOPER*, etc.). *Connection Type* selection should be “*Cloud Wallet*”. In the *Detail* tab, navigate to the folder where the *Oracle Cloud wallet* is stored and save it.

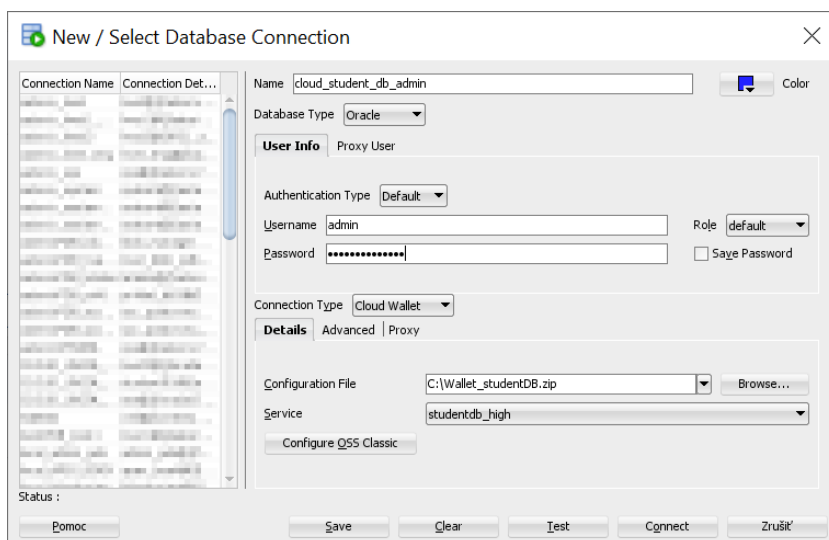


Fig. 1.49: *SQL Developer connection definition (2)*

Test the connection by clicking on the *Test* button. In the left part, you should see the status “*Success*”. Otherwise, the exception will be raised navigating you to the issue. Solve the problem and try again. If the status is “*Success*”, click on *Connect* button. The connection definition will be saved, and you will be routed to the *SQL Developer* environment allowing you to access the defined database using the connection.

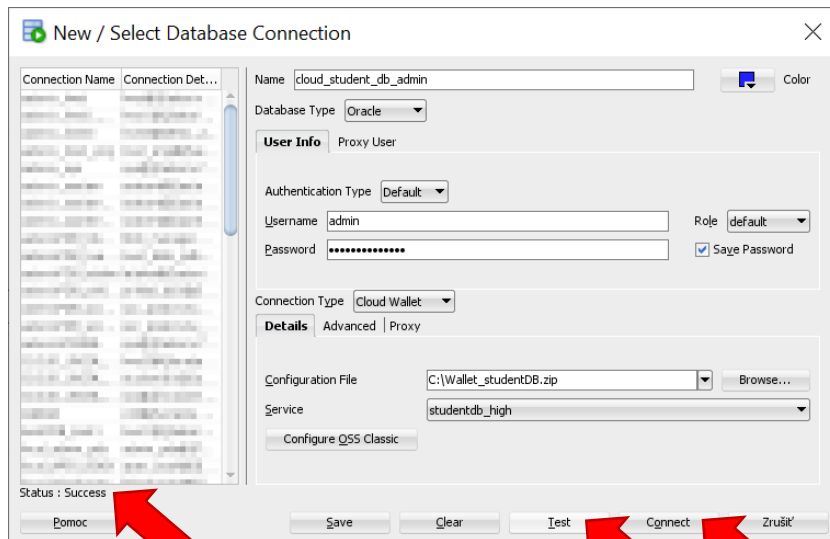




Fig. 1.50: *SQL Developer connection definition (3)*

The currently selected connection is visible in the right part of the screen, as well. It is helpful in the case of using multiple connections in parallel. The service combo box shows the list of connections loaded from the *TNSNAMES.ORA* is stored inside the wallet. There are five types in general. *Low*, *medium* and *high* are primarily used for the data warehouses, whereas *tp* and *tpurgent* focus on the transaction database service connection. The differences are based on sources and parallelism used.

By default, the *SQL Developer* environment consists of two parts. The left part consists of the *Navigator*, a list of connections, and other types made visible (in case of the following figure, *Reports* are listed, as well, we will drive you through the reporting possibilities later in the next chapter). The right part is used for the *SQL* and *PL/SQL* code specifications. After the execution, such part is divided, the upper part remains original, the bottom part shows the results (similarly to the *SQL Developer Web*).

Let's write the following query and execute it. The command can be executed by clicking on the “*Run statement*” button  or by using *CTRL+Enter* shortcut.

```
select sysdate from dual;
```

The whole (selected) script is executed by clicking on the “*Run script*”  button or invoking execution using the *F5* shortcut.

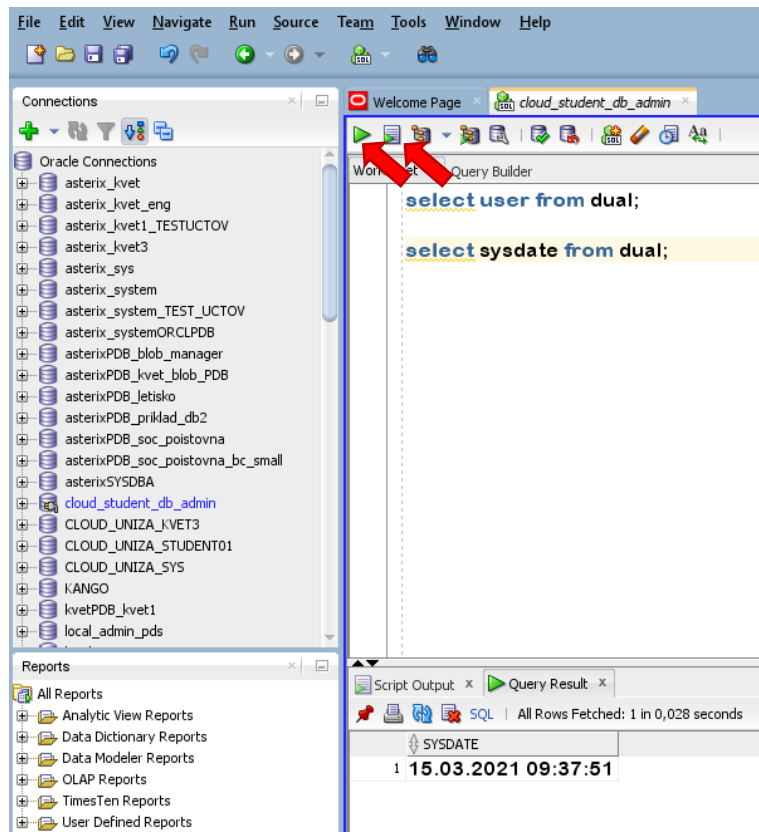


Fig. 1.51: SQL Developer environment

Do not be confused due to the different formats produced for the *Date* value. It depends on the server or *SQL developer* session selection, respectively.

Create a new table named *TAB* containing only one numeric attribute *ID*:

```
Create table TAB(id integer);
```

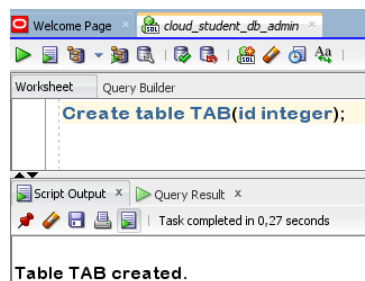


Fig. 1.52: Table definition

Note that the font and size can be altered in the following menu context:
Tools => Preferences => Code Editor => Fonts:

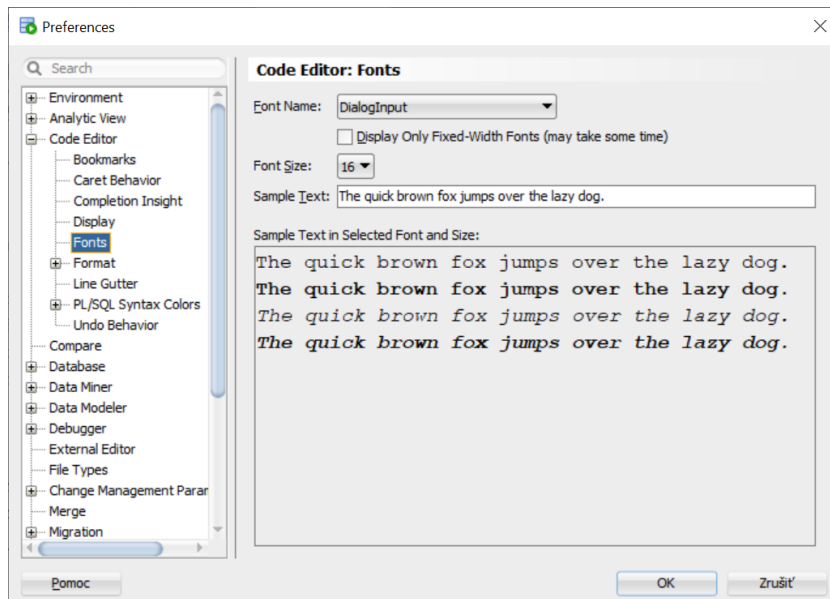


Fig. 1.53: Environment font specification

1.2 SQL*Plus command-line – SQL Client

Connection to the Cloud instance can be made by the *console tool* of the *Instant Client* or *sqlplus* application itself. *Oracle Instant Client* can be downloaded from the following website: <https://www.oracle.com/database/technologies/instant-client/downloads.html>



Fig. 1.54: The QR code of the Oracle Instant Client installation repository

Please, select the appropriate operating system version. It is available for *Windows*, *Linux*, *macOS*, *Solaris*, *HP*, or *AIX*. Then, choose the newest version and download three packages based on the name:

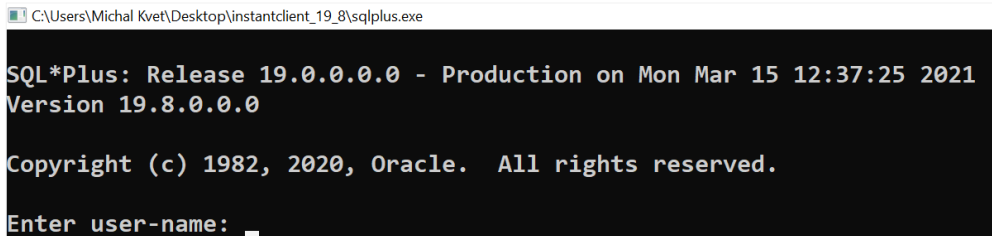
- *Basic Package*,
- *SQL*Plus Package*,
- *Tools Package* (including *SQL Loader*, *Import*, *Export* functionality, and *Data Pump* tools).

Each file consists of one archive file with the *.zip* extension (for OS Windows). Extract all downloaded files and copy the content to one common repository consisting of all files. Inside the destination folder, several tools can be found, like *Adrci*, *Exp*, *Expdp*, *Imp*, *Impdp*, *Sqldr* or *Sqlplus*, and some others, however, the listed tools will be used. As stated,

connection to the database cloud instance can be made by using the *sqlplus* command-line tool. After launching it, username and connection details are necessary to be specified.

Username consists of two parts – login and connection details defined either by the *full (entire) connect string* or by *connect identifier* pointing to the stored connect definition:

```
login@connect_string
login@connect_identifier
```



```
C:\Users\Michal Kvet\Desktop\instantclient_19_8\sqlplus.exe

SQL*Plus: Release 19.0.0.0.0 - Production on Mon Mar 15 12:37:25 2021
Version 19.8.0.0.0

Copyright (c) 1982, 2020, Oracle. All rights reserved.

Enter user-name: _
```

Fig. 1.55: SQL*Plus command-line connection specification

The login value is clear. It has been specified during the database definition or during new user creation, respectively. For us, we have identified only one user called “*admin*”. Connection details can be, in principle, specified by two alternatives, which will be consecutively described.

The first alternative is associated with the *full definition*. The second alternative is based on the already existing connection string reference.

1.2.1 Alternative 1 – full definition

To provide the ability to connect via full definition, *Oracle wallet* will be used. It consists of several files inside the archive. We will need the file *TNSNAMES.ORA* containing the connect string list and *EWALLET* with the encryption keys.

Structure of the *TNSNAMES.ORA* file takes the identifiers of the connection followed by the inner definition (host, port, service name, and connection parameters and security aspects).



```
tnsnames.ora - Notepad
File Edit Format View Help

studentdb_high = (description= (retry_count=20)(retry_delay=3)(address=(protocol=tcps)(port=1522)(host=adb.eu-frankfurt-1.oraclecloud.com))(connect_data=(service_name=fwuydcbkqbsqo83_studentdb_high.adb.oraclecloud.com))(security=(ssl_server_cert_dn="CN=adwc.eucom-central-1.oraclecloud.com, OU=Oracle BMCS FRANKFURT, O=Oracle Corporation, L=Redwood City, ST=California, C=US"))))

studentdb_low = (description= (retry_count=20)(retry_delay=3)(address=(protocol=tcps)(port=1522)(host=adb.eu-frankfurt-1.oraclecloud.com))(connect_data=(service_name=fwuydcbkqbsqo83_studentdb_low.adb.oraclecloud.com))(security=(ssl_server_cert_dn="CN=adwc.eucom-central-1.oraclecloud.com, OU=Oracle BMCS FRANKFURT, O=Oracle Corporation, L=Redwood City, ST=California, C=US"))))

studentdb_medium = (description= (retry_count=20)(retry_delay=3)(address=(protocol=tcps)(port=1522)(host=adb.eu-frankfurt-1.oraclecloud.com))(connect_data=(service_name=fwuydcbkqbsqo83_studentdb_medium.adb.oraclecloud.com))(security=(ssl_server_cert_dn="CN=adwc.eucom-central-1.oraclecloud.com, OU=Oracle BMCS FRANKFURT, O=Oracle Corporation, L=Redwood City, ST=California, C=US"))))

studentdb_tp = (description= (retry_count=20)(retry_delay=3)(address=(protocol=tcps)(port=1522)(host=adb.eu-frankfurt-1.oraclecloud.com))(connect_data=(service_name=fwuydcbkqbsqo83_studentdb_tp.adb.oraclecloud.com))(security=(ssl_server_cert_dn="CN=adwc.eucom-central-1.oraclecloud.com, OU=Oracle BMCS FRANKFURT, O=Oracle Corporation, L=Redwood City, ST=California, C=US"))))

studentdb_tpurgent = (description= (retry_count=20)(retry_delay=3)(address=(protocol=tcps)(port=1522)(host=adb.eu-frankfurt-1.oraclecloud.com))(connect_data=(service_name=fwuydcbkqbsqo83_studentdb_tpurgent.adb.oraclecloud.com))(security=(ssl_server_cert_dn="CN=adwc.eucom-central-1.oraclecloud.com, OU=Oracle BMCS FRANKFURT, O=Oracle Corporation, L=Redwood City, ST=California, C=US"))))

Ln 1, Col 1      100%  Windows (CRLF)  UTF-8
```

Fig. 1.56: TNSNAMES.ORA content – connection specification details

Database connections are made based on the execution workload, so choose appropriate and copy the definition (for the following part, *high service type* will be used). To allow you to connect, it is necessary to extend it with the encryption keys. Without them, it would be impossible to communicate – the whole communication is always strictly encrypted, ensuring complex security. Thus, after the definition, add one new clause pointing to the location of the keys (encryption keys are in the *EWALLET* file). So, take the wallet archive, unzip it, and copy it to a separate folder. Extend the definition by locating encryption keys:

```
(description=(retry_count=20)(retry_delay=3)
  (address=(protocol=tcps)(port=1522)
    (host=adb.eu-frankfurt-1.oraclecloud.com))
  (connect_data=(service_name=
    fwuydcbkqbsqo83_studentdb_high.adb.oraclecloud.com))
  (security=(ssl_server_cert_dn=
    "CN=adwc.eucom-central-1.oraclecloud.com,
    OU=Oracle BMCS FRANKFURT,
    O=Oracle Corporation,
    L=Redwood City,
    ST=California,
    C=US"))))
```

Let the location of the e-wallet be: *C:\oracle_wallet* (use any address you want, do not use diacritics or spaces in the path). Then, interconnect these elements by adding the *MY_WALLET_DIRECTORY* clause with the pointer to the e-wallet directory. In my case, it would look like the following:

```
(MY_WALLET_DIRECTORY="C:\oracle_wallet")
```

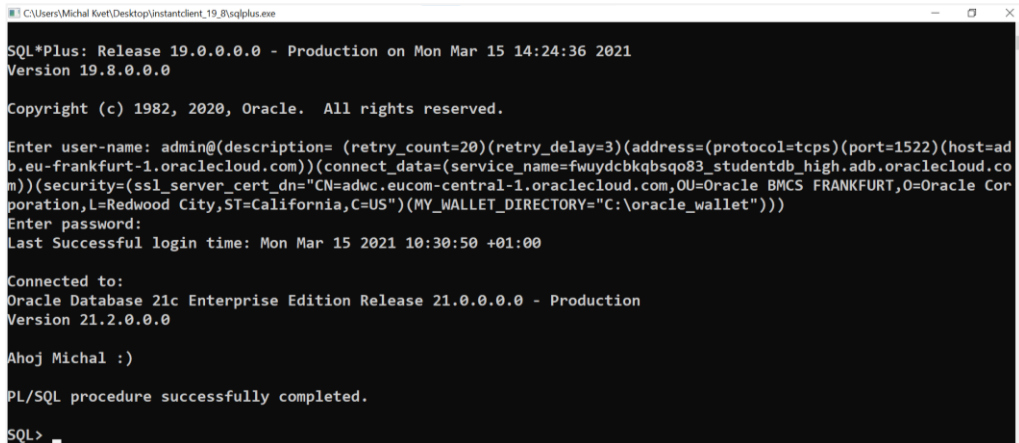
Finally, create the whole connect string by joining those elements:

```
(description=(retry_count=20)(retry_delay=3)
  (address=(protocol=tcps)(port=1522)
    (host=adb.eu-frankfurt-1.oraclecloud.com))
  (connect_data=(service_name=
    fwuydcbkqbsqo83_studentdb_high.adb.oraclecloud.com))
  (security=(ssl_server_cert_dn=
    "CN=adwc.eucom-central-1.oraclecloud.com,
    OU=Oracle BMCS FRANKFURT,
    O=Oracle Corporation,
    L=Redwood City,
    ST=California,
    C=US"))
  (MY_WALLET_DIRECTORY="C:\oracle_wallet"))
```

Take emphasis on the brackets, please. The above definition is a *full connection specification* – connect string. To connect via *command line*, use your login followed by the at (@) and connect string:

```
login@connect_string
```

```
admin@(description=(retry_count=20)(retry_delay=3)(address=
(protocol=tcps)(port=1522)(host=adb.eu-frankfurt-1.oraclecloud.com))
(connect_data=(service_name=fwuydcbkqbsqo83_studentdb_high.adb.oraclecloud.c
om))(security=(ssl_server_cert_dn="CN=adwc.eucom-central-
1.oraclecloud.com,OU=Oracle BMCS FRANKFURT,O=Oracle Corporation, L=Redwood
City,ST=California,C=US"))(MY_WALLET_DIRECTORY="C:\oracle_wallet")))
```



```
C:\Users\Michal Kvet\Desktop\instantclient_19_8\sqlplus.exe
SQL*Plus: Release 19.0.0.0.0 - Production on Mon Mar 15 14:24:36 2021
Version 19.8.0.0.0

Copyright (c) 1982, 2020, Oracle. All rights reserved.

Enter user-name: admin@(description=(retry_count=20)(retry_delay=3)(address=(protocol=tcps)(port=1522)(host=adb.eu-frankfurt-1.oraclecloud.com))(connect_data=(service_name=fwuydcbkqbsqo83_studentdb_high.adb.oraclecloud.com))(security=(ssl_server_cert_dn="CN=adwc.eucom-central-1.oraclecloud.com,OU=Oracle BMCS FRANKFURT,O=Oracle Corporation, L=Redwood City,ST=California,C=US"))(MY_WALLET_DIRECTORY="C:\oracle_wallet"))
Enter password:
Last Successful login time: Mon Mar 15 2021 10:30:50 +01:00

Connected to:
Oracle Database 21c Enterprise Edition Release 21.0.0.0.0 - Production
Version 21.2.0.0.0

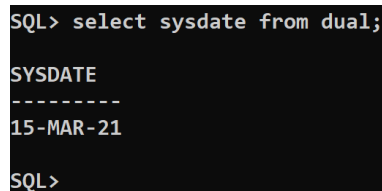
Ahoj Michal :)

PL/SQL procedure successfully completed.

SQL>
```

Fig. 1.57: Full connection

Similar to the *SQL Developer*, a script can be defined in such an environment. *SQL Developer* is, however, significantly better in terms of user experience and user-friendly environment.



```
SQL> select sysdate from dual;

SYSDATE
-----
15-MAR-21

SQL>
```

Fig. 1.58: Command-line client environment

1.2.2 Alternative 2 – connect identifiers

As evident, the above principles are too complicated for daily activity. The definition is complicated, so the user data must be stored somewhere to be copied. The second alternative is based on the stored connect identifiers, which are then referenced. To do so, download the *wallet* if you have not done it already. Then, extract the archive and copy the content to the directory. In my case, I will use the folder path “C:\oracle_wallet”. Then, you must create a system variable named *TNS_ADMIN* pointing to such a repository. System variables can be specified. For *Windows* operating system, navigate to the *Control Panel* => *System* => *Advanced system settings* (in the left panel). Alternatively, you can type “path” in the *Start* menu and then choose “Edit the system environment variables”. Another way to open the System Properties window is to type “SystemPropertiesAdvanced.exe” in the *Start* menu or in the *Run* window (opened by WinKey+R shortcut).

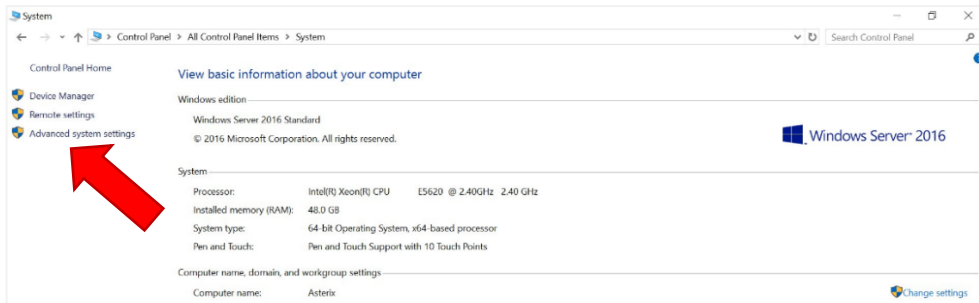


Fig. 1.59: Environment variable definition (1)

Click on the *Environment Variables* button:

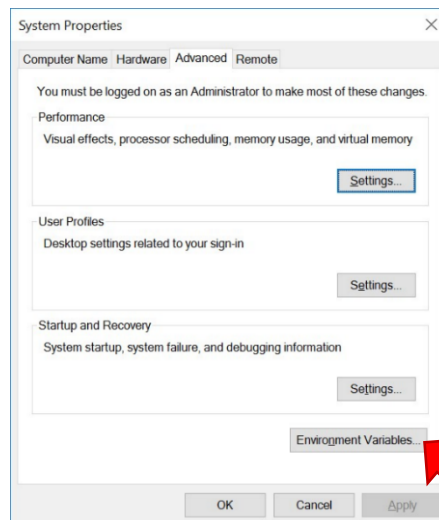


Fig. 1.60: Environment variable definition (2)

A new window will be launched consisting of a list of system variables. Click on the *New* button to add a new environment variable.

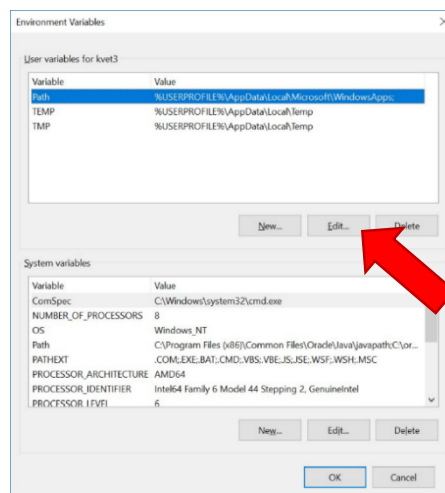


Fig. 1.61: Environment variable definition (3)

The variable's name is “*TNS_ADMIN*” – the name is strict, be aware while specifying it. *Variable value* is a path to the *Oracle Wallet* extracted folder (in my case: *C:\oracle_wallet*).

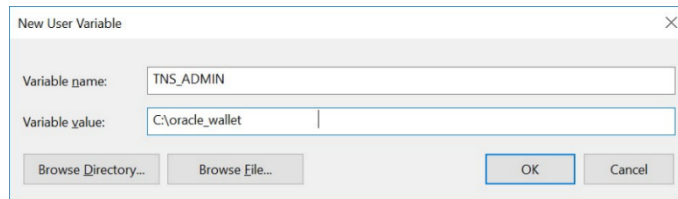


Fig. 1.62: Environment variable definition (4)

The list of connect strings is located in the *TNSNAMES.ORA* file (such file name cannot be changed!). Open such file, the name before the equality sign reflects the *connect identifier*, you can change its name. The name should not contain spaces and special symbols and should be unique.

The last step is the encryption key association. Open the file *SQLNET.ORA* inside the extracted folder and modify the element *WALLET_LOCATION*, part *DIRECTORY*. Use the folder with the extracted wallet (in my case: *C:\oracle_wallet*).

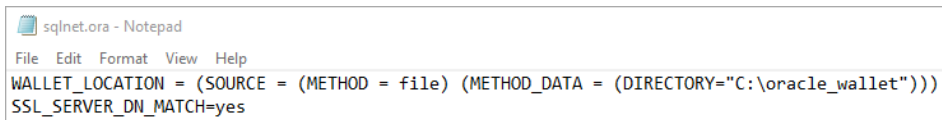


Fig. 1.63: Oracle wallet location definition

Save the changes and try to connect.

In the *Instant client*, you have two options – specify the full connection details (*alternative 1*) or use *connect identifier*. I have not changed the name of the identifier so that I will use the original one – *studentdb_high*. In your case, use the name located in the *TNSNAMES.ORA* file. The username of the *Instant client* will then be like the following:

login@connect_identifier

admin@studentdb_high

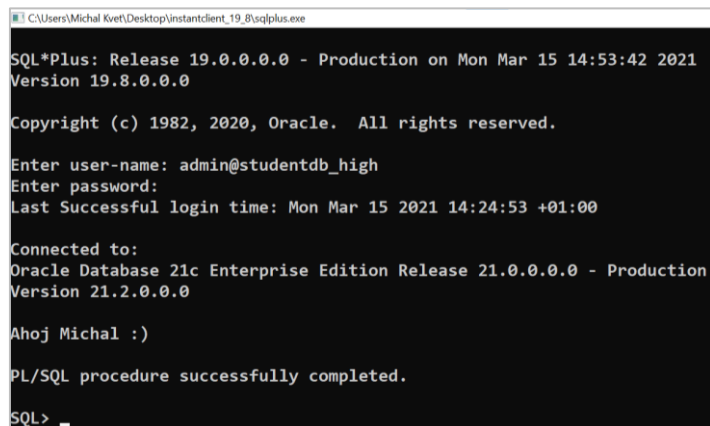


Fig. 1.64: SQL Instant client connection definition

Note that during the password definition, the cursor does not move. Do not be stressed. It will work correctly.

For now, open the enclosed file *STUDENT_DATA.SQL* and execute the whole script. It consists of the object table definitions, and data are consecutively loaded. I recommend you study the code at least briefly. You can see that tables are created using *Create table* command followed by the *relationship definition* used for the table interconnection – *references, joining*. Then, individual rows are created invoking the *Insert statement*. Such principles, syntax, and usage will be complexly described in the following sections. However, in this phase, it is inevitable to load data to the table as quickly as possible to deal with the example data.

The following Select statement can be used to get the list of tables created in your schema (user account definition). Value of the **table_name** attribute will be selected from the object **tabs** (**tabs** reflects the *synonym* to the *data dictionary view* (system table) **user_tables** consisting of the table definition. More about *data dictionary views* can be found in chapter [Lab 14 – Data dictionary views](#)):

```
select table_name from tabs;
```

Note that all data in the *data dictionary views* are uppercase:

```
SQL> select table_name from tabs;

TABLE_NAME
-----
PERSON
SUBJECT_PREF
RESULT_TAB
LOG_STUDENT
NEW_STUDENT
```

Fig. 1.65: Select statement result set

Schema of the table can be provided using the *description command* (**desc**). Table schema defines the structure of the table – name of the attributes with their definitions (data types, NULL / NOT NULLs, ...). More about the table structure can be found in chapter [Lab 4 – Data modeling](#) and [Lab 5 – Create, Alter and Drop commands](#).

```
desc table_name;
```

```
SQL> desc personal_data

Name                               Null?    Type
-----
PERSONAL_ID                       NOT NULL CHAR(11)
NAME                              VARCHAR2(15)
SURNAME                           VARCHAR2(15)
STREET                             VARCHAR2(20)
TOWN                               VARCHAR2(50)
ZIP                                CHAR(5)
NATIONALITY                       CHAR(2)
```

Fig. 1.66: Table description

In this part, it is also necessary to understand the differences between various data types.

Tab. 1.1: Data types

Data type	Special characteristics
Date	Date and time spectrum in the HH:MI:SS format (not the date only!)
Timestamp	Contains date and time (HH:MI:SS:FFFF) up to 9 precision level
Char(x)	String format with fixed size definition – x characters exactly
Varchar2(x)	String format with variable size – maximum x characters
Integer, Long, Float, Double precision, Number	Self-explanatory data types, <i>integer</i> is a subgroup of <i>number</i> data type with no fractional part
LOB	Large objects (BLOB, CLOB, NCLOB, BFILE) for storing binary files (music, photos, etc.) or complex textual data

To get the results of the simple *Select* statement, write the following code. The aim is to get data values of the following attributes – *personal_id*, *name*, and *surname*. It can be obtained from the table *personal_data*, which schema has been described in the previous code example.

```
select personal_id, name, surname from personal_data;
```

The provided result set is following:

```
SQL> Select personal_id, name, surname from personal_data;

PERSONAL_ID NAME          SURNAME
-----
841106/3456 Michael      Pearce
840312/7845 Jack        Smith
860907/1259 John         Young
850130/3695 Carol        Pearce
841201/1248 Carol        Pearce
830514/5341 William     Whittel
```

Fig. 1.67: Select statement result set

Notice that each *SQL command* must end with a semicolon (;). It is often repeated the mistake of students – no data will be returned, whereas a command is not finalized and thus impossible to be executed.

To get the results of the last command (usually stored in *afiedt.buf* file), slash (/) can be used. Moreover, the last executed *SQL command* can be edited using the editor launched using the *ed* command. Management inside the editor is based on previously described editor principles. Notice that *SQL statements* are not case sensitive except data in dictionary views. Mentioned editor data can hold only one command.

```
I  afiedt.buf          Row 2   Col 1   8:11  Ctrl-K H for help
Select personal_id, name, surname from personal_data
/
```

Fig. 1.68: Modifying buffered statement

Be aware, here, in the last command editor, the end of the statement is not delimited by the semicolon but by a slash (/) located in the last row (as the only one character).

Moreover, this command can be used only if at least one *SQL command* has been executed in the session. If not, you will get the following error information:

```
SP2-0107: Nothing to save.
```

Launching the last statement from the buffer can be performed using the slash (/).

The default editor type can be changed using the following command in *sqlplus* (notice the space before the underscore symbol).

```
define _editor=editor_name
```

The following command shows the example of assigning *joe* as the default editor in *sqlplus*.

```
define _editor=joe
```

Besides editor management and login using *SQL*Plus* command, all activities can also be performed in the *SQL developer* tool.

1.2.3 Capturing activities in SQL

All performed activities can be stored using recording (capturing) technology. In that case, the console output is also routed to the file using stream. First, starting recording is provided using *spool* command followed by the file name (in *SQL developer*, usually full path is used).

```
spool file_name
```

From that moment, all statements and results shown in the console are automatically stored in the defined file until the process is stopped using the *spool off* command.

```
spool off
```

So, let's have the example of the recording. Start the process and write some *SQL* commands (use the previously defined statement). Afterward, stop the process and show the file data.

```
spool data_output.txt
select personal_id, name, surname from personal_data;
spool off
```

The file can be edited using the *ed* command followed by the file name with extension:

```
ed data_output.txt
```

```
SQL> Select personal_id, name, surname from personal_data;

PERSONAL_ID NAME          SURNAME
-----
841106/3456 Michael      Pearce
840312/7845 Jack        Smith
860907/1259 John         Young
.....
860712/6475 Frederico    Ducato

43 rows selected.

SQL> spool off
```

Fig. 1.69: Spooling

Notice that if you omit the file extension, default extension **.lst* will be used when the recording process is started.

You can redirect the execution to the operating system commands (OS Linux) using the *host* command. Otherwise, it would be necessary to exit *SQL*Plus* and execute such a command in a *Linux* environment (determined by the *\$* symbol).

```
host ls file_name
```

1.2.4 Working with Help

Connection to the database system allows you to use an embedded *helper*. To view the basic *Help* menu, the command *help* should be used followed by the topic (category), you are looking for some hints.

```
help topic
```

In the following example, the help for the *start* command is shown.

```
help start
```

```
START
-----

Runs the SQL*Plus statements in the specified script. The script can be
called from the local file system or a web server.

STA[RT] {url|file_name[.ext]} [arg ...]

where url supports HTTP and FTP protocols in the form:

    http://host.domain/script.sql

STARTUP
-----

Starts an Oracle instance with several options, including mounting,
and opening a database.

STARTUP options | upgrade_options

where options has the following syntax:
[FORCE] [RESTRICT] [PFILE=filename] [QUIET] [ MOUNT [dbname] |
[OPEN [open_options] [dbname1] ] ]
```

Fig. 1.70: Help

Despite that, it is strongly recommended to use official documentation for *DBS Oracle* covering the latest *patches* (<https://docs.oracle.com/>).

1.2.5 Working with multiple commands

We strongly recommend writing commands to the files when dealing with multiple commands as well as for dealing with the following labs.

The file can be created and edited either in the OS environment but also in the *SQL*Plus*. The usage is the same. The first following solution describes the command in the OS. The second one reflects the direct use in *SQL*Plus*. Notice that if you try open editor with the file name (and extension as well) and such file does not exist, it is automatically created. Vice versa, if it exists, it is opened in edit mode. We use *joe* editor in the OS environment for explanation purposes, but feel free to use that one you prefer.

```
joe file_name.sql          (in OS Linux)
```

```
ed file_name.sql           (in SQL*Plus)
```

```
host joe file_name.sql     (in SQL*Plus, accessing Linux)
```

In the file, any *SQL commands* can be written and consequently started on the database system server. Notice that individual commands must be ended with the semicolon (;). Editor management is described in chapter [4.9.7 Working with directories and files](#) (saving changes can be done in *joe* using **CTRL+K+X** shortcuts).

Notice that the SQL developer tool uses a more user-friendly environment, so file management is far more manageable.

1.2.6 Comments

It is inevitable to comment on your code. One row comment is characterized by two dashes (-) followed by at least one space. Multiple row comment starts with a slash (/) followed by an asterisk (*) and at least one space or new line. The reverse order is used to end multiline comment – at least one space (or newline) followed by asterisk and slash.

```
-- one-row comment
```

```
/* multiple  
   line  
   comment  
*/
```

The created file can be started in *SQL*Plus* to execute the commands written inside using the **start** command followed by the file name and extension (it can be omitted if the extension is **.sql*). The first following command shows the syntax. The rest are examples. The last two commands are equivalent. Notice that SQL statements are commonly written into the file with the **.sql* extension.

```
start file_name.sql  
start lab1.sql  
start lab1
```

Currently, a connected *username* can be obtained using the following commands (*dual* reflects special table and is described in chapter [2.2.2 Dual table](#)).

```
show user
```

Notice that the **show user** command is not an SQL command. Therefore, it is unnecessary to end it with a semicolon (using it does not cause any error).

```
select user from dual;
```

1.2.7 Working with procedures and functions

The code of the methods (procedure, function) is stored in the files and consecutively loaded into the system preceded by the compilation process. The compilation process is a significant part of the loading, ensuring correctness. The code of the method is parsed and stored in the database data dictionary. Thus, if the method is compiled successfully, original code from the file is no longer necessary (it is possible to reconstruct code from the data dictionary).

If you attempt to create a stored method, which cannot be compiled successfully, although it will be loaded, the **status** of such method will be **invalid** and cannot be executed at all. Transforming **invalid** object to **valid** is always performed by the compilation process. Vice versa, the **invalid** object can originate from unsuccessful loading or by changing dependent objects.

So, let's have the simple procedure example stored in the file (*first_procedure.sql*). The name of the procedure is **proc_get_row_number**.

```
create or replace procedure proc_get_row_number
  v_count integer;
begin
  select count(*) into v_count
  from personal_data;
  dbms_output.put_line('The number of the rows' ||
                      'in personal_data table is: ' || v_count);
end;
/
```

Be aware that each procedure, function, trigger, or package must end with a slash (/) as a separate character in the last line. It delimits the final separator, whereas multiple blocks can be nested.

If you omit it, it will not be compiled, and the system will wait to add it. Inside this procedure, the local variable **v_count** is defined for storing a number of rows of the *personal_data* table. Such value is obtained using *Select* statement – result set (one value) is stored in a defined local variable (*select count(*) into v_count*). Afterward, the value of that local variable is printed on the console screen by calling the *put_line* method of the *dbms_output* package.

Launching the code from the file is performed in *SQL*Plus* using the **start** command. Thus, in our case, the process will look like this, resulting in compilation error identification:

```
start first_procedure.sql
```

```
Warning: Procedure created with compilation errors.
```

If errors are identified during the loading, the **show err** can obtain a list of the problems.

```
show err
```

The output of the method looks like this. The numeric value in the first part of the line expresses the line number, where the *error* is located. In our case, it is value “2”.

```
2/3      PLS-00103: Encountered the symbol "V_COUNT" when expecting one
of the following: ( ; is with authid as cluster compress order using
compiled wrapped external deterministic parallel_enable pipelined
result_cache
The symbol "is" was substituted for "V_COUNT" to continue.
```

Be aware that the line number in the file does not need to correspond to the line (in the file) during the compilation!

It can be caused by other things by *PL/SQL optimize level*. So, the real code in such line can be obtained using the *list <line_number>* command (it can be abbreviated to *l*) command – in our case, *list 2 (l2)*.

```
list <line_number>
```

```
12
```

The output of the method based on the previously defined procedure *proc_get_row_number* will be following:

```
v_count integer;
```

Thus, the error is located on line **2** – the code *v_count integer;*. The problem is based on missing keyword before the second line command. Please add the word “**IS**”, save the file, and compile it once again.

```
create or replace procedure proc_get_row_number
is
  v_count integer;
begin
  select count(*) into v_count
  from personal_data;
  dbms_output.put_line('The number of the rows' ||
                      'in the personal_data table is: ' || v_count);
end;
/
```

Now the procedure will be successfully compiled. Notice that if multiple errors have been identified, always remove the problems up to down, whereas some problem corrections can remove numerous consecutive errors.

If the procedure is created without compilation errors, the *status* will be *valid* and such method will be possible to be executed (using *execute* command):

```
execute procedure_name
```

```
execute proc_get_row_number
```

The result is following:

```
The number of the rows in the personal_data table is: 35
```

Executing function is similar, but the result must be assigned (e.g., to a local variable).

Notice that the output display must be enabled to see the results. The *SERVEROUTPUT* setting controls whether SQL prints the output generated by the *dbms_output* package

from PL/SQL procedures to the environment. It must be enabled for the session (or for the whole server) before the first execution of the *dbms_output* package (see chapter 9.5 Executing stored method). Otherwise, no output will be printed to the user.

```
set serveroutput on
```

1.2.8 Connection and session termination

Individual changes to the data must be confirmed using the **commit** command (transaction is ended successfully). It ensures that data is durable and cannot be lost in any case. Thus, it is better to do it relatively often. In the following section, each first part defines the syntax. Then the server answer is listed.

```
commit;
```

```
commit complete.
```

The opposite of the **commit** command is a **rollback**, which removes all values changed in the current transaction. It is related to the beginning point of the transaction. The transaction starts automatically when connecting to the database or directly after ending the previous one.

```
rollback;
```

```
rollback complete.
```

More about transaction management can be found in chapter 3.8 Transactions.

Disconnecting from the database is provided using the **disconnect** command.

```
disconnect;
```

```
Disconnected from Oracle Database 19c Enterprise Edition Release  
19.0.0.0.0 - Production Version 19.3.0.0.0
```

Reconnecting or changing the user signed in the session can be ensured using the **connect** command followed by the *login* and *connect string* defining database instance. Notice that the *connect string* can be defined in the *TNSNAMES.ora* file located in *\$ORACLE_HOME/network/admin*.

```
connect login@orcl;  
connect login@xe;
```

In Cloud environment, the connection identifier is delimited name of the database followed by the extension (*low*, *medium*, *high*, *tp*, *tp_urgent*):

```
connect login@library_low;  
connect login@student_tp;
```

After work completion, command **exit** can be used to exit the *SQL*Plus* environment. The same functionality is also provided by pressing *CTRL+D* keys. Notice that physical implementation of the **exit** command automatically invokes **commit**.

```
exit;
```


Never turn off SQL*Plus (SQL Client, console) using the Close button of the window (the cross of the right corner). This is not the correct completion of work. Changes are not committed. Thus, after new login, executed not-committed statements will not be found in the database. Moreover, if some table or row is locked, such state remains for a defined period, even after re-login.

Consequently, you will not be able to access all data and work fully. It is evident that **Process Monitor** (PMON) does not proactively control connected user processes to minimize network and communication system load workload. However, if the user process does not communicate during the defined time (does not send any request), the corresponding server process is killed by the PMON, and used server resources are freed. It means that all work is rolled back, and locks are released.

1.3 Syntax symbols

The following chapters cover all main SQL commands extended by multiple examples and characteristics. We will use the standardized syntax definition:

[] ... optional part
 { } ... multiple choices (one should be chosen)
 | ... divisor of the choices in { }
 < **object_name** > ... the name of the object, which is replaced by the real reference, like STUDENT for table name, etc.

These symbols should not be written explicitly in the statements. Instead, they only describe the syntax possibilities.

On the other hand, standard parentheses () are part of the syntax. Therefore, they must be part of the command.

As you noted above, any code or SQL command will appear in a red box in this book:

This is a line of code

The outputs from the code are displayed in either a gray box or a gray table:

This is a line of code output

	COLUMN_1	COLUMN_2	COLUMN_N
1	Value 1	Value 2	Value N

If you see a code in the text marked in a red dashed box with red font, we want to warn you that the code is incorrect or otherwise erroneous:

This is a line of wrong code

Chapter content summary is listed in the following format:

Summary of the section

Please note that individual figures are not strictly referenced in the book. Instead, we use the approach of the description strictly directly preceding the figures throughout the whole book.

Lab 2 – Basics of data retrieval

This lab introduces the main clauses of the `Select` statement definition. Projection can be made by defining the list of attributes, expressions, or values in the `Select` clause. Selection limits the number of rows part in the result set by applying conditions in the `Where` clause.

This chapter summarizes the person identification principles used in Slovak and Czech region – personal_id, composed of the date of birth, gender and distinguishing part.

Individual queries can deal with the attributes, but also expressions and function calls can be used. Section 2.3 proposes the summary of the most important methods, categorized into the string, math, date and time and conversion function types.

Among the `Select` and `Where` clause, the focus is done on the table reference using the relationships, operated by the Join, located in the `From` clause of the `Select` statement. If the `Join` operation is not done properly across multiple tables, a Cartesian product combining all data is present.

The value comparison across the membership in a set can be made by using IN or EXISTS or their negative variants (`NOT IN`, `NOT EXISTS`). By using `Join` operations, duplicate values can be present for particular rows, which can be limited by using the Distinct keyword in the `Select` clause. It can also be generally used to remove duplicates from the set.

Finally, the importance of the aliases for the tables and attributes is discussed.

2.1 Introduction

Data manipulation is the central part of the user activities accessing the database. The user connects to the database, gets and processes required data from the database. Therefore, the main challenge is to select rows to be changed, deleted, or just retrieved. Usually, the database consists of hundreds, thousands, or even more data rows, and the manual accessing, and evaluation process would be complicated and time-consuming. Therefore, the `Select` statement definition has been proposed allowing you to specify which data you want as well as a form of the result set. Database system optimizer automatically evaluates the defined query and provides the searching for you. The easiest way is to get all data stored in the table, but commonly you want to get all the rows that satisfy a condition or multiple conditions, even based on multiple tables. In this lab, a `Select` statement definition with individual clauses is introduced supported by multiple examples qualifying result sets.

The `Select` statement is used to query the database and retrieve selected data that match the criteria you specify. It has six main clauses for the command definition. Each clause has many options, selections, parameters, etc. Individual clauses will be listed below, but each will be covered in more detail later in this book.

Syntax of the `Select` statement is following:

```
SELECT [{ALL | DISTINCT}] column1 [, column2, ...]
FROM table1 [JOIN table2 {ON condition | USING condition} ...]
[WHERE conditions]
[GROUP BY column_list]
[HAVING conditions]
[ORDER BY column_list [{ASC | DESC}]];
```

Please, separate each used clause to the new line and align the code. When using more complex statements, such practice will be appreciated.

For consecutive data management, it is inevitable to describe principles of data identification. It is done by using the **primary key**, which uniquely identifies each record in a database table. The **primary key (PK)** must contain **UNIQUE** values and cannot hold **NULL**. A table can have only one primary key, consisting of single or multiple fields (**composite primary key**). In the model, the primary key element is signed as “**PK**”. More about the primary key definition, management, and importance will be described later in chapter [11.3.2 Primary key](#).

2.2 Projection, selection, column alias

The easiest **Select** statement does not contain any condition, and all attribute values are obtained. In principle, it is possible to list all attribute names detached by colons (,), but also asterisk (*) wildcard can be used, which lists all attributes based on the table schema automatically (the order of attributes is delimited by the schema and can be gotten using the description of the table – e.g., **desc personal_data**). Thus, the following two commands are providing the same results. Naturally, if the attribute order is significant, a named notation must be used.

```
select personal_id, name, surname, street,
       town, zip, nationality
from personal_data;
```

```
select * from personal_data;
```

The result set will look like this (export of *SQL Developer* tool):

	PERSONAL_ID	NAME	SURNAME	STREET	TOWN	ZIP	NATIONALITY
1	841106/3456	Michael	Pearce	Kamenna 27	Banska Bystrica	97401	SK
2	840312/7845	Jack	Smith	Zelena 9	Nove Mesto nad Vahom	91501	SK
3	860907/1259	John	Young	Slnečne namestie	Komarno	94501	SK
4	850130/3695	Carol	Pearce	Stred 49/7	Povazska Bystrica	01701	SK
5	841201/1248	Carol	Pearce	Juh 2100/456	Trencin	91101	SK
6	830514/5341	Wiliam	Whittel	Tahanovce 38/12	Kosice	04001	SK

As you can see, some attribute values are denoted with the **(null)** values. Notice that such value is not physically stored in the database, but it expresses undefined value – thus, there is no address information for *Simone Smith*. In the command line (*SQL Client*), a **NULL** value is modeled by an empty string representing the same fact.

	PERSONAL_ID	NAME	SURNAME	STREET	TOWN	ZIP	NATIONALITY
1	845210/6525	Simone	Smith	(null)	(null)	(null)	(null)

In the previous case, all data table rows have been selected. **Where** clause of the **Select** statement can limit the result set based on defined conditions. In the following example, we will list only persons whose first name is *Michael*. As we can see, *four rows are selected*. Data are compared based on equality. Thus, also the font style should be highlighted (*lower / upper case*).

```
select *
  from personal_data
 where name = 'Michael';
```

	PERSONAL_ID	NAME	SURNAME	STREET	TOWN	ZIP	NATIONALITY
1	841106/3456	Michael	Pearce	Kamenna 27	Banska Bystrica	97401	SK
2	830301/7789	Michael	Simson	Lesna 7/12	Ruzomberok	03401	SK
3	740210/6536	Michael	Flower	(null)	(null)	(null)	(null)
4	880329/1233	Michael	Smith	(null)	(null)	(null)	(null)

Also, multiple conditions can be used cooperating based on **OR** or **AND** evaluation techniques. When multiple conditions are used, parentheses usage is preferred. Thus, in the following example, we will list only persons living in *Zilina* town. Moreover, the name of the person must be “*Jack*”. The condition connector is **AND** (conditions must apply at the same time). The second condition limits the number of data rows in the result set to 1.

```
select *
  from personal_data
 where town = 'Zilina' and name = 'Jack';
```

	PERSONAL_ID	NAME	SURNAME	STREET	TOWN	ZIP	NATIONALITY
1	791229/5431	Jack	Robinson	A. Bemolaka 14/20	Zilina	01001	SK

The order of the conditions to be evaluated is not essential, whereas a database query optimizer can rearrange the order to speed up the evaluation by limiting the data amount to be processed in the next step. Naturally, using parentheses forces the system to use user-predefined order.

In the previous examples, condition *Where* removed data tuples, that do not meet the conditions – *relational algebra Selection* operation has been used:

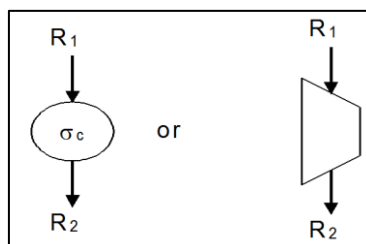


Fig. 2.1: Selection

The second *relational algebra* operation is just **Projection**, which removes some attributes from the result set. The list of attributes, which values should be obtained, is defined in the *Select* clause of the statement delimited by commas (,).

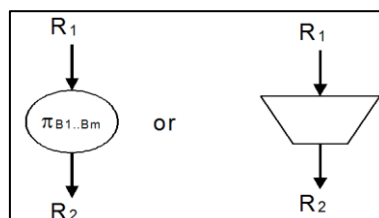


Fig. 2.2: Projection

Pure *Projection* operation does not contain *Selection* inside.

```
select name, surname
from personal_data;
```

However, in real application usage, individual statements are usually created by the combination of *Selection* and *Projection*.

```
select name, surname
from personal_data
where town='Zilina';
```

As we can see, attribute names in the result set are the same as the attribute name in the table definition. It is possible to rename them in the result set using *aliases*. The syntax is highlighted.

```
select name as first_name, surname as family_name
from personal_data;
```

	FIRST_NAME	FAMILY_NAME
1	Michael	Pearce
2	Jack	Smith
3	John	Young
4	Carol	Pearce

In the standard environment, the attribute name must contain only one word, but this restriction can be replaced by encapsulating the alias into the quotes (" ")

```
select name as "first name", surname as "family name"
from personal_data;
```

Keyword “**AS**” is optional and does not need to be used.

```
select name "first name", surname "family name"
from personal_data;
```

However, some limitations must be described and followed. Otherwise, an exception will be raised. For example, in principle, if the attribute name is changed (column alias is defined), such alias cannot be used in the *Where* clause of the same *Select* statement, but the original name must be used. The reason is, that the *Where* clause is evaluated during the first stage of the processing. Vice versa, these aliases must be used for superior *Select* statements.

Thus, the first example is incorrect. The right solution is shown in the second example.

```
select name as first_name, surname as family_name
from personal_data
where first_name='Michael';
```

It is not possible to use defined alias in the *Where* clause of the same statement:

```
ORA-00904: "FIRST_NAME": invalid identifier
```

```
select name as first_name, surname as family_name
from personal_data
where name='Michael';
```

As you can see in the result set, the original attribute name is renamed to *first_name*. The same principle is applied for *surname* as well.

	<i>FIRST_NAME</i>	<i>FAMILY_NAME</i>
1	Michael	Pearce
2	Michael	Simson

We will explain the structure, meaning, and importance of the *personal_id* attribute before the consecutive processing and the rest clauses explanation.

2.2.1 Personal_id structure

Personal_id refers to the birth number as the numerical identifier assigned to Slovakia and Czech Republic people. *Personal identification number* belongs to the *personal data*. It is formed from the *person's date of birth*, *gender*, and the *terminal digit*, which is a distinctive sign of people born on the same calendar day.

The first two digits of the birth number are the last two digits of the person's birth year. The second two digits express the numeric designation of the month of birth of the person (value is increased by 50 for women), the third two digits express the numerical designation of the date of birth of the person in that calendar month. This is the example of the man and woman:

90 06 23 / 1234

MAN

Year = 90

Month = 6

Day = 23

90 56 23 /1239

WOMAN

Year = 90

Month = 6

Day = 23

The whole personal identifier should be divided by 11 without remainder (this rule is not strict, sometimes, it can happen that the division rule does not pass):

9056231239 / 11 = 823293749

2.2.2 Dual table

The **Dual** table is a special type of table consisting of one row and one attribute. It is always present in the DBS, and such a table cannot be changed based on the structure nor the data inside. In *DBS Oracle*, it is defined by a single **VARCHAR2(1)** attribute called **DUMMY**, which carries the value "X". Such a table is useful for obtaining pseudo column values or results of the functions independent of the table data.

Let's consider some examples. The first one returns the actual date and time. The second one returns the value 20 based on the defined mathematical operator "+". The last one returns the length of the string, value 12. Notice that string value should be encapsulated in apostrophes.

```
select sysdate from dual;
```

```
select 10+10 from dual;
```

```
select length('some text...') from dual;
```

2.3 Using functions

It is not necessary to work only with defined attributes themselves, but the conditions, as well as the *Select* clause itself, can also consist of function results to be evaluated. This lab

will only deal with system-defined functions; the most important ones will be described based on parameters and usage. User-defined functions can also be part of the statements if some prerequisites and conditions are met (user-defined function management is described in [Lab 9 – Procedures, functions and packages](#)).

DBS Oracle provides a wide range of functions in the *standard* package for dealing with values by obtaining their part, convert the value to another data type, or transforming output. Such functions can be generally categorized into the following groups (*Oracle documentation categorization is used*):

- String/Char functions,
- Numeric/Math functions,
- Date/Time functions,
- Conversion functions,
- Analytic functions,
- Advanced functions,
- Miscellaneous functions.

2.3.1 Character string functions

This category carries the functions dealing with strings. We will list the most important ones with their characteristics and usage.

ASCII function

ASCII function returns the numeric representation of the character in the *ASCII table*. The parameter of the function should be a single character. However, if multiple characters are pushed, only the first one is evaluated.

```
select ASCII(single_character) from dual;
```

```
select ASCII('A') from dual;
```

CONCAT function

The *Concat* function allows you to connect (concatenate) two strings together.

```
select CONCAT(first_string, second_string) from dual;
```

```
select CONCAT(name, surname) from personal_data;
```

The disadvantage of the mentioned function is that it can accept only two parameters, thus if multiple strings need to be concatenated together, such function must be executed multiple times, and the code is hardly readable or even adjustable. The previous example puts two strings together without any space.

	CONCAT(NAME, SURNAME)
1	MichaelPearce
2	JackSmith
3	JohnYoung

Thus, the correct solution (with space) can look like the following:

```
select CONCAT(name, CONCAT(' ', surname)) from personal_data;
```


	CONCAT(NAME,CONCAT(",SURNAME))
1	Michael Pearce
2	Jack Smith
3	John Young

To concatenate four string values, *Concat* function must be used at least three times:

```
select CONCAT(CONCAT(CONCAT('A', 'B'), 'C'), 'D') from dual;
```

For definition and management simplification, function management definition provides *pipe (||) operator* to concatenate any string count.

```
select string1 || string2 [|| ... string_n] from dual;
```

```
select 'A' || 'B' || 'C' || 'D' from dual;
```

As you can see, constant strings must be delimited by the *apostrophes* ('). In the English language, however, such a symbol can also have a special denotation. Therefore, the system must be able to distinguish whether the apostrophe character is part of the standard string or should be treated as a delimiter. For these purposes, if the apostrophe is part of the string text, it must be doubled. Both following examples provide the same results.

```
select 'Let's' || ' study Informatics' from dual;
```

```
select 'Let' || ''' ' || 's' || ' study Informatics'
from dual;
```

	LET''' 'S'''STUDYINFORMATICS'
1	Let's study Informatics

String character case management (LOWER, UPPER, INITCAP functions)

The size of the string can be transformed using three functions, which provide sufficient power to ensure the correct string format to be stored in the database. *Lower* function converts all characters in the specified string to lowercase. Vice versa, the *Upper* function converts all characters to uppercase. Notice that characters, which are not letters (like numbers), are not changed at all. Special functionality provides the *InitCap* function, which sets the first character of each word to uppercase. The rest ones are lowercase. The principles are shown in the following example:

```
select lower('DATABASE SYSTEMS'),
       upper('database systems'),
       initcap('DatabaSE SySTems')
from dual;
```

LOWER('DATABASESYSTEMS')	UPPER('DATABASESYSTEMS')	INITCAP('DATABASESYSTEMS')
database systems	DATABASE SYSTEMS	Database Systems

If there is a necessity to set only the first letter of the first word to uppercase, a combination of the previously described functions can be used:

```
select upper(substr('DatabaSE SySTems', 1, 1)) ||
       lower(substr('DatabaSE SySTems', 2))
from dual;
```

LENGTH function

Length function returns the number of characters of the specified string.

```
select length(string_value) from dual;
```

```
select length('some text...') from dual;
```

The value *12* will be returned in the previous example:

```
12
```

SUBSTR function

Substr function extracts a substring from the provided input string.

```
select substr(string, start_position [, length]) from dual;
```

The first parameter (*string*) defines the source string to be processed. The second parameter delimits the starting position (*start_position*). **Notice that the numbering in DBS starts with 1.** The last parameter (*length*) is optional and defines the number of characters to be extracted. If the value of the third parameter is not defined, the rest part of the string up to the end will be returned.

```
select substr(personal_id, 5, 2) from personal_data;
```

```
select substr(personal_id, 5) from personal_data;
```

Let's consider the results of the previous *Select* statements in comparison with the whole *personal_id* value. The first *Select* statement returns the day sequence number of the birth. The second one will start with day, followed by the slash and the rest part of the *personal_id* value up to the end of the string.

PERSONAL_ID	SUBSTR(PERSONAL_ID,5,2)	SUBSTR(PERSONAL_ID,5)
781015/4431	15	15/4431
791229/5431	29	29/5431
800407/3522	7	07/3522
810101/8079	1	01/8079

TRIM function

The *Trim* function removes all specified characters either from the beginning or from the end of the provided string. The default option is **BOTH**. Using the **LEADING** keyword sets the system to start with the beginning, whereas **TRAILING** starts from the end. Parameter *trim_character* defines which symbol should be removed.

```
select TRIM([ [LEADING | TRAILING | BOTH] trim_character FROM ] string)
from dual;
```

Generalization is the function *Trim* with only one string parameter, which ensures that all whitespace characters are removed either from the beginning or from the end.

```
select trim(string) from dual;
```

Let's consider the following example. The input string contains three spaces from the beginning and four spaces at the end. The original string length is 18. The **Trim** function removes those spaces, so the post-processed length is 11.

```
select length('   test string   ') from dual;
```

```
18
```

```
select trim('   test string   ') from dual;
```

```
test string
```

```
select length(trim('   test string   ')) from dual;
```

```
11
```

Be aware, such a function removes spaces only as the first or last letters. Thus, the result of the following statement is the same as a solution without using the **Trim** function. The reason is based on the first and last character – asterisk (*).

```
select trim('*   test string   *') from dual;
```

```
TRIM('*TESTSTRING*')
```

```
*      test string      *
```

2.3.2 Numeric and Math functions

ABS function

This function returns the absolute value of the provided numerical input value (*num_input*).

```
select abs(num_input) from dual;
```

```
select abs(125) from dual;
```

```
125
```

```
select abs(-125) from dual;
```

```
125
```

CEIL function

Ceil function returns the smallest integer value that is greater or equal to a defined number (*num_input*).

```
select ceil(num_input) from dual;
```

```
select ceil(2.5) from dual;
```

```
3
```

```
select ceil(3) from dual;
```

```
3
```

```
select ceil(-2.5) from dual;
```

```
-2
```

ROUND function

Round function returns a numerical input value (*num_input*) rounded based on the defined number of decimal places (*decimal_places*). If the second parameter is omitted, the value is rounded completely. However, also the negative value of the second parameter can be provided. In that case, it defines the position in the main part.

```
select round(num_input, [decimal_places]) from dual;
```

```
select round(67812.345) from dual;
```

```
67812
```

```
select round(67812.345, 2) from dual;
```

```
67812.35
```

If the precision parameter (*decimal_places*) value is negative, it defines the principle of rounding the main part of the number. Thus, if value “-2” is defined, the value is rounded based on the second value of the main part from the right part – see the example:

```
select round(67812.345, -2) from dual;
```

```
67800
```

The value “8” is rounded by the followed value 9:

```
select round(67892.345, -2) from dual;
```

```
67900
```

FLOOR function

Floor function returns the largest integer value equal or less than the defined number (*num_input*).

```
select floor(num_input, [decimal_places]) from dual;
```

```
select floor(2.7) from dual;
```

```
2
```

```
select floor(3) from dual;
```

```
3
```

```
select floor(-2.5) from dual;
```

```
-3
```

TRUNC function

The **Trunc** function has many variants. It returns truncated value to the defined number of decimal places for numerical inputs. In comparison with the **Round** function, this definition puts away decimal value.

```
select trunc(num_input, [decimal_places]) from dual;
```

```
select trunc(67812.345) from dual;
```

```
67812
```

```
select trunc(67812.345, 2) from dual;
```

```
67812.34
```

```
select trunc(67812.345, -2) from dual;
```

```
67800
```

The value "8" is truncated regardless of the following values:

```
select trunc(67892.345, -2) from dual;
```

```
67800
```

MOD function

Mod function returns the remainder of *m* divided by *n*.

```
select mod(m, n) from dual;
```

```
select mod(7, 3) from dual;
```

```
1
```

```
select mod(-7, 3) from dual;
```

```
-1
```

To get a real month of birth from the *personal_id* attribute value, *mod* functionality provides sufficient power. Consider the following definition. First of all, the month segment is extracted and subsequently processed to remove the woman impact.

```
select mod(substr(personal_id, 3, 2), 50)
from personal_data;
```

2.3.3 Date and Time functions

Date attribute value management is a complicated process, which must highlight multiple conditions to get desirable and correct results. First of all, the *Date* attribute value in DBS Oracle always **stores the date and the time sphere**. So please, never forget it, it is essential.

Adding or subtracting numerical value from the *Date* expresses the number of days (or its part).

```
select to_date('15.02.2021', 'DD.MM.YYYY') + 1 from dual;
```

```
TO_DATE('15.02.2021','DD.MM.YYYY')+1
```

```
16.02.2021 00:00:00
```

Notice that the conversion function *To_date* in the example deals only with the day, month, and year elements. In that case, conversion causes, undefined components (hours, minutes, seconds) are replaced with zero values.

The result of subtracting two *Date* values is the number of days between them (also with the decimal part). So, the result value will be "2".

```
select to_date('15.02.2021', 'DD.MM.YYYY') -
       to_date('13.02.2021', 'DD.MM.YYYY')
from dual;
```

```
2
```

Direct transformation of the result to a number of months or years would be incorrect because inexact values would be provided (some months have 30 days; some have 31 days. Moreover, *February* is special). Therefore, for dealing with time, provided functionality does not provide exact results (e.g., for flights or VISA, the exact age of the person must be provided).

SYSDATE function

The *Sysdate* function has already been used in the previous example. It returns the current system date and time on the local database. It does not have any parameters and reflects the *Date* type as output format. The precision of the *Date* format is up to seconds – *DD.MM.YYYY HH24:MI:SS*. Remember, that *Date* type value always stores the date as well as time.

```
select sysdate from dual;
```

SYSDATE
27.02.2021 09:57:24

The format of the output of the *Date* functions and *Date* value management depends on the actual system or session settings. To get the current format set for your session, query *NLS_SESSION_PARAMETERS* view. To get the format of the server property, required information can be obtained by querying the *NLS_DATABASE_PARAMETERS* view. For both views, the parameter name is *NLS_DATE_FORMAT*. Notice that it reflects the *data dictionary view*, where all values are uppercase (see [Lab 14 – Data dictionary views](#)).

```
select value
from nls_session_parameters
where parameter = 'NLS_DATE_FORMAT';
```

```
select value
from nls_database_parameters
where parameter = 'NLS_DATE_FORMAT';
```

Changing actual settings can be done by altering the session or system. *Alter* command is used (principles of *Alter* command can be found in chapter [5.4.2 Alter](#)).

```
alter session set nls_date_format='format';
```

```
alter session set nls_date_format='DD.MM.YYYY HH24:MI:SS';
```

For the whole system property changing, use the previous command, but reflect “*system*” instead of “*session*”.

SYSTIMESTAMP function

The *Systimestamp* function also returns the current system date and time, but also second fractions and time zone can be provided. The output data type is *Timestamp(n)*, where *n*

defines the number of fractions of the seconds from the integer range <0 ; 9>. The *default* value is 6, if the precision is not specified explicitly.

```
select systimestamp from dual;
```

SYSTIMESTAMP
02.03.17 11:15:52,413000

ADD_MONTHS function

The *Add_months* function has two parameters – date value (*date_val*) and a number of months to be added (*number_months*) or subtracted. The result is the input date with processed months. Such functionality can also be done explicitly by parsing the *Date* attribute. However, the transition between years and months must be handled explicitly.

```
select add_months(date_val, number_months) from dual;
```

```
select add_months(to_date('15.02.2021', 'DD.MM.YYYY'), 7) from dual;
```

ADD_MONTHS(TO_DATE('15.02.2021','DD.MM.YYYY'),7)
15.09.2021 00:00:00

```
select add_months(to_date('15.02.2021', 'DD.MM.YYYY'), -7) from dual;
```

ADD_MONTHS(TO_DATE('15.02.2021','DD.MM.YYYY'),-7)
15.07.2020 00:00:00

Notice that the function automatically correctly manages transitions through years leap years...

```
select add_months(to_date('15.02.2021', 'DD.MM.YYYY'), 17) from dual;
```

ADD_MONTHS(TO_DATE('15.02.2021','DD.MM.YYYY'),17)
15.07.2022 00:00:00

The following example shows the principle of adding a month to the *Date* value regarding the last day of the month. What will be the result if you want to add one month to 31.1.2017? One month is added, so *February* is reflected, namely the last day of that month.

```
select add_months(to_date('31.01.2017', 'DD.MM.YYYY'), 1) from dual;
```

ADD_MONTHS(TO_DATE('31.01.2017','DD.MM.YYYY'),1)
28.02.2017 00:00:00

EXTRACT function

Extract function returns the defined subpart of the *Date/Timestamp* attribute value – *day*, *month*, *year*, *minute*, *hour*, and *second* can be provided. However, using one function execution, only one element can be provided. Notice the difference between *Date* and *Timestamp* values. Using this function, the time spectrum can be obtained only for *Timestamp* values. *Extract* from *Date* value can provide only *day*, *month*, and *year*.

```
select extract({YEAR | MONTH | DAY} from date_val) from dual;
```

```
select extract({YEAR | MONTH | DAY | HOUR | MINUTE | SECOND})
       from timestamp_val)
       from dual;
```

```
select extract(month from to_date('15.02.2017', 'DD.MM.YYYY'))
       from dual;
```

```
2
```

```
select extract(year from to_date('15.02.2017', 'DD.MM.YYYY'))
       from dual;
```

```
2017
```

```
select extract(minute from to_date('15.02.2017', 'DD.MM.YYYY'))
       from dual;
```

```
ORA-30076: invalid extract field for extract source
```

```
select extract(minute from to_timestamp('15.02.2017', 'DD.MM.YYYY'))
       from dual;
```

```
0
```

LAST_DAY function

The *Last_day* function returns the last day of the month based on the input *Date* value (*date_val*).

```
select last_day(date_val) from dual;
```

```
select last_day(to_date('15.12.2017', 'DD.MM.YYYY'))
       from dual;
```

```
31.12.2017
```

Naturally, it manages also leap years.

```
select last_day(to_date('15.2.2017', 'DD.MM.YYYY'))
       from dual;
```

```
28.2.2017
```

```
select last_day(to_date('15.2.2016', 'DD.MM.YYYY'))
       from dual;
```

```
29.2.2016
```

MONTHS_BETWEEN function

As the function's name indicates, the *Months_between* function returns the number of months between two defined dates. To get a positive value, the first parameter value (*date_val1*) must be greater than the second parameter (*date_val2*). If not, a negative result will be provided.

```
select months_between(date_val1, date_val2) from dual;
```



```
select months_between(to_date('15.12.2017', 'DD.MM.YYYY'),
                     to_date('15.2.2017', 'DD.MM.YYYY'))
from dual;
```

10

```
select months_between(to_date('15.2.2017', 'DD.MM.YYYY'),
                     to_date('15.12.2017', 'DD.MM.YYYY'))
from dual;
```

-10

The result of such function can also be the real number:

```
select
  months_between(to_date('15.12.2017 6:22:12', 'DD.MM.YYYY HH:MI:SS'),
                to_date('13.2.2017 5:13:12', 'DD.MM.YYYY HH:MI:SS'))
from dual;
```

10,066

Division the difference between two *Date* values by 30 does not provide relevant data due to leap years and the number of days in an individual month. However, this function is robust works correctly, so to get actual age, use this function result.

NEXT_DAY function

The output of the *Next_day* function is the first *weekday* greater than the defined input date (*date_val*).

```
select next_day(date_val, weekday) from dual;
```

Tab. 2.1: Weekday

Weekday	Description
SUNDAY	First Sunday greater than input date (<i>date_val</i>).
MONDAY	First Monday greater than input date (<i>date_val</i>).
TUESDAY	First Tuesday greater than input date (<i>date_val</i>).
WEDNESDAY	First Wednesday greater than input date (<i>date_val</i>).
THURSDAY	First Thursday greater than input date (<i>date_val</i>).
FRIDAY	First Friday greater than input date (<i>date_val</i>).
SATURDAY	First Saturday greater than input date (<i>date_val</i>).

For the illustration purposes, we will also get the day of the processed date. The *date* of the first example is *Wednesday*, the second example deals with *Sunday* date:

```
select actual_date,
       to_char(actual_date, 'DAY'),
       next_day(actual_date, 'SUNDAY'),
       to_char(next_day(actual_date, 'SUNDAY'), 'DAY')
from (select to_date('15.2.2017', 'DD.MM.YYYY') as actual_date
      from dual);
```

ACTUAL_DATE	TO_CHAR (ACTUAL_DATE,'DAY')	NEXT_DAY (ACTUAL_DATE,'SUNDAY')	TO_CHAR (NEXT_DAY(ACTUAL_DATE,'SUNDAY'),'DAY')
15.02.2017 00:00:00	WEDNESDAY	19.02.2017 00:00:00	SUNDAY

What about the difference between the function result *last_day* and getting last day based on *trunc* and *add_month* function? Does it provide the same results? Let's consider the following examples. The first one is based on invoking the *last_day* function. Compared to the second solution, the time spectrum is not trimmed away.

```
select last_day(sysdate) from dual;
```

LAST_DAY(SYSDATE)
31.03.2017 19:53:05

```
select TRUNC(ADD_MONTHS(sysdate,1), 'MM')-1 from dual;
```

TRUNC(ADD_MONTHS(SYSDATE,1),'MM')-1
31.03.2017 00:00:00

2.3.4 Conversion functions

Conversion functions are used for transferring the input value to another data type. Most of them are called automatically by DBS (implicit conversions). However, some transformations are recommended to be done explicitly to avoid future problems with incorrect data processing. Moreover, explicit conversion gives performance benefits.

TO_CHAR function

To_char function converts the input value (date or numeric) into the string format. Whereas the principles are different, we will describe them separately.

Conversion of the numeric value (*num_val*) into string format can also be done automatically by implicit conversions called automatically by the server. However, such a function can also be called explicitly to ensure that the value will be processed and evaluated as a string data type value.

```
select to_char(num_val, [format]) from dual;
```

To_char function dealing with numeric values can also accept the second parameter – *format*, which is optional. In principle, it can influence the style of the returned string format. It can take various format parameter values. We will mention the most important of them. Value “0” returns leading zero values. Value “9” expresses specified format – if the digit is not present, *space* will be used instead. Used *Comma* defines the position of the comma in the output format. It is possible to define multiple commas constructing the number format model.

```
select to_char(123, '999999'), to_char(123, '000000'),
       to_char(123456, '999,999')
from dual;
```

TO_CHAR(123,'999999')	TO_CHAR(123,'000000')	TO_CHAR(123456,'999,999')
123	000123	123,456

Parameter value “RN” transfers input value into *Roman numerals*.

```
select to_char(123, 'RN') from dual;
```

CXXIII

To_char method can also be used for *Date* format models. For *Date* value conversions, two parameters should be highlighted. The first value is the value of the *Date* or *Timestamp* data type (*date_val*). The second parameter defines the *format_mask*. Some standard format models are shown in the following table.

```
select to_char(date_val [, format_mask]) from dual;
```

Tab. 2.2: *Format_mask*

Abbreviation	Meaning
Date:	
DD	Day of the month (01 up to 31)
Day	Day in the week – the first letter is uppercase (e.g., Saturday)
MM	Month (01 up to 12)
Month	Name of the month, the first letter is uppercase
MON	The first three letters from the month name (uppercase)
YY	Year (00 up to 99)
YYYY	Year (including century, e.g., 1999 or 1901)
RR	Year (00 up to 99)
RRRR	Year (including century, e.g., 2001 or 2002)
Time:	
HH	Hour (01 up to 12)
HH24	Hour (00 up to 23)
MI	Minute (00 up to 60)
SS	Second (00 up to 59)

The third parameter also characterizes the language used during the conversion (primarily used, if text format should be provided).

```
select to_char(date_val [, format_mask] [, nls_language]) from dual;
```

This is an example of its usage:

```
select to_char(sysdate, 'DD.MM.(Month).YYYY HH24:MI:SS',
              'nls_date_language=English')
from dual;
```

```
TO_CHAR(SYSDATE,'DD.MM.(MONTH).YYYYHH24:MI:SS','NLS_DATE_LANGUAGE=ENGLISH')
14.10.(October ).2021 14:21:27
```

```
select to_char(sysdate, 'DD.MM.(Month).YYYY HH24:MI:SS',
              'nls_date_language=Slovak')
from dual;
```

```
TO_CHAR(SYSDATE,'DD.MM.(MONTH).YYYYHH24:MI:SS','NLS_DATE_LANGUAGE=SLOVAK')
14.10.(Október ).2021 14:21:40
```

Additional spaces can be removed by trimming the output. It can be done by extending the format using “FM”:

```
select to_char(sysdate, 'FM DD.MM.(Month).YYYY HH24:MI:SS',
              'nls_date_language=Slovak')
from dual;
```

Never use substring functionalities (substr) to get elements from the date. It is incorrect and hazardous. If the substr function is used, the input value must be a string. Thus, an implicit conversion function from Date data type to Varchar2 data type is used, and server/session date format is used. From that string, the defined substring is obtained. Therefore, if the Date format was changed, an incorrect substring would be obtained because of the element's position in the string. It significantly limits the applicability and deployment options of the solution. Never do it. You simply cannot guarantee that the format will never be changed.

Let's consider the following example.

```
select substr(sysdate, 1, 2), sysdate from dual;
```

SUBSTR(SYSDATE,1,2)	SYSDATE
15	15.08.2021 01:51:21

```
alter session set nls_date_format='YYYY.MM.DD';
select substr(sysdate, 1, 2), sysdate from dual;
```

SUBSTR(SYSDATE,1,2)	SYSDATE
20	2021.08.15

TO_DATE function

To_date function is similar to the *To_char* function but in the opposite direction. Value is transformed from the string format into the *Date* type value. It consists of two parameters, which should be mentioned – string value and the format, which carries the same principles as the format in the *To_char* method. The optional *nls_language* parameter will be described later.

```
select to_date(string_val [, format_mask] [, nls_language]) from dual;
```

Never let the system use Date implicit conversions. It can lead to incorrect data if the predefined system time format is changed.

TO_NUMBER function

To_number function is used to transfer string value into the numerical value. It is usually called automatically, if necessary.

```
select to_number(string_val) from dual;
```

TO_TIMESTAMP function

To_timestamp function is used to convert a string (*string_val*) into the *timestamp* data type. This is also the second function, which is highly preferable not to allow to be executed implicitly. The second parameter (*format_mask*) of the function defines the format of the string to be transferred. If omitted, the system-defined format is used. However, it can cause problems – functions will not provide correct results if the format is changed on the server. It also significantly limits the possibility to extend the application and put it into another server. If the server time format is not the same, incorrect results will be provided. See the following results. We use two system *Date* formats. Without using the conversion function, wrong data binding will occur. In the first case, the first values delimit

day followed by the month. In the second example, the order is reversed. *DD* defines the day, *MM* reflects months, the year is expressed by *YYYY*, time reflects *HH24* (24-hour format), *HH* (12-hour format), *MI* (minutes), and *SS* (seconds). *FF* delimits fractions of the seconds.

```
select to_timestamp(string_val [, format_mask] [, nls_language])
from dual;
```

```
select to_timestamp('2.9.2016 15:24:13.1', 'DD.MM.YYYY HH24:MI:SS.FF')
from dual;
```

```
select to_timestamp('9.2.2016 15:24:13.1', 'MM.DD.YYYY HH24:MI:SS.FF')
from dual;
```

Multilanguage solution can look like this:

```
select to_timestamp('2.March.2016 15:24:13.1',
                  'DD.Month.YYYY HH24:MI:SS.FF',
                  'nls_date_language=American')
from dual;
```

```
select to_timestamp('2.marec.2016 15:24:13.1',
                  'DD.Month.YYYY HH24:MI:SS.FF',
                  'nls_date_language=Slovak')
from dual;
```

2.3.5 Advanced functions

CASE conversion function

Case function is a conditional type with the *If-Then-Else* functionality. DBS searches for the first *When ... Then* branch, for which input expression (*in_expr*) is equal to comparison expression (*cond_expr*, respectively *val_expr*) and returns the *return_expr* for the particular branch. If none of the conditions meets, and the *Else* clause exists, then such branch value (*else_expr*) is returned. Otherwise, a *NULL* value is returned. Notice that the *Case* function ends with the keyword “*End*”.

In principle, we can distinguish two types. The first one puts the condition inside each *When* branch. The second one separates the expression to be evaluated, which is located directly after the keyword *Case*.

```
CASE in_expr
  WHEN val_expr_1 THEN return_expr_1
  WHEN val_expr_2 THEN return_expr_2
  ...
  WHEN val_expr_n THEN return_expr_n
  ELSE else_expr
END
```

```
CASE
  WHEN cond_expr_1 THEN return_expr_1
  WHEN cond_expr_2 THEN return_expr_2
  ...
  WHEN cond_expr_n THEN return_expr_n
  ELSE else_expr
END
```

Let's have a simple example to describe functionality. The aim is to get the status of the student in text form based on the attribute *status* of the student table:

- student.status:
 - S = student (actual),
 - E = ended successfully,
 - A = aborted,
 - X = fired due to disciplinary commission decision.

```
select student_id,
       case status
         when 'S' then 'student'
         when 'E' then 'ended successfully'
         when 'A' then 'aborted'
         when 'X' then 'fired - disciplinary'
       end as student_result
from student;
```

```
select student_id,
       case
         when status='S' then 'student'
         when status='E' then 'ended successfully'
         when status='A' then 'aborted'
         when status='X' then 'fired - disciplinary'
       end as student_result
from student;
```

The results are the same:

STUDENT_ID	STUDENT_RESULT
550545	aborted
550020	student
501567	ended successfully
501319	student

COALESCE function

Coalesce function has an unlimited number of parameters, which usually can hold *NULL* values. It returns the first *NOT NULL* value from the list. If none of them is *NOT NULL*, there is no other chance, a *NULL* value is returned.

```
COALESCE(in_expr_1, in_expr_2, ... in_expr_n)
```

```
select COALESCE(married_surname, birth_surname) from person;
```

```
select COALESCE(employee_to, sysdate) from employee;
```

DECODE function

The *Decode* function has similar functionality as the *If-Then-Else* conditional processing. This function is used for data transformation. Principles are identical to the previously mentioned *Case* function. This is the syntax of the *Decode* functionality and an example of the *gender* and month transformation to the text format.

```
DECODE(in_expr, search_1, result_1 [, ...]
       [, search_n, result_n]
       [, else_result])
```

```
select decode(substr(personal_id, 3, 2), '5', 'female',
                                             '6', 'female',
                                             'male')
from personal_data;
```

```
select name, surname,
       'Born in ' || decode(mod(substr(personal_id, 3, 2), 50),
                            1, 'January', 2, 'February', 3, 'March',
                            4, 'April', 5, 'May', 6, 'June',
                            7, 'July', 8, 'August', 9, 'September',
                            10, 'October', 11, 'November', 12, 'December',
                            'unknown')
from personal_data;
```

NULLIF function

Nullif function compares two provided parameter values (*expr1*, *expr2*). If the values are the same, the *NULL* value is returned. Otherwise, it returns the first parameter value (*expr1*). In principle, data types should be the same.

Due to the return value definition, the *expr1* expression cannot be *NULL* literal (but the expression can hold *NULL* value).

```
select NULLIF(expr1, expr2) from dual;
```

```
select NULLIF(null, null) from dual;
```

```
ORA-00932: inconsistent datatypes: expected - got CHAR
```

```
Select NULLIF(1, 1) from dual;
```

```
NULL
```

```
Select NULLIF(1, null) from dual;
```

```
1
```

NVL function

NVL function allows you to substitute the input value (*in_val*) with (*ret_val*) value, if *NULL* value is encountered. If the *in_val* value is *NOT NULL*, the original value is returned.

```
select NVL(in_val, replace_val) from dual;
```

```
select NVL(1, 2) from dual;
```

```
1
```

```
select NVL(null, 2) from dual;
```

```
2
```

NVL2 function

NVL2 extends the possibilities of the previously described *NVL* function by another attribute. The return value depends on the first parameter value (*in_val*). If it is *NULL*, the third parameter value is returned (*replace_val_NULL*). Otherwise, the second parameter value is used (*replace_val_NOT_NULL*).


```
select NVL2(in_val, replace_val_NOT_NULL, replace_val_NULL) from dual;
```

```
select NVL2(1, 2, 3) from dual;
```

```
2
```

```
select NVL2(null, 2, 3) from dual;
```

```
3
```

USER function

The *User* function returns the login of the connected user to the actual session. The function has no parameters.

```
select user from dual;
```

SYS_CONTEXT function

Sys_context is a useful function that can provide you with information about the current environment. It has been introduced in Oracle 8i version to replace the existing *USERENV* function. It can accept several parameters and provide you with a range of information like *current user information*, *session information*, *services*, or *nls_parameters* set in the *actual session*. Individual parameter list evolves over the versions. Therefore, to be sure, use the actual *DBS Oracle documentation* release.

There are syntax and some examples:

```
select SYS_CONTEXT(namespace, parameter) from dual;
```

Tab. 2.3: *Sys_context* parameters

Parameter	Explanation
CURRENT_SCHEMA	Returns the default schema used in the current schema
CURRENT_USER	Name of the current user
CURRENT_USERID	USERID of the current user
DB_DOMAIN	Domain of the database, from the DB_DOMAIN initialization parameter
DB_NAME	Name of the database, from the DB_NAME initialization parameter
DB_UNIQUE_NAME	Name of the database, from the DB_UNIQUE_NAME initialization parameter
EXTERNAL_NAME	External name of the database user
HOST	Name of the host machine from which the client has connected
INSTANCE	The identification number of the current instance
INSTANCE_NAME	The name of the current instance
IP_ADDRESS	IP address of the machine from which the client has connected
ISDBA	Returns TRUE if the user has DBA privileges. Otherwise, it will return FALSE
LANG	The ISO abbreviate for the language
LANGUAGE	The language, territory, and character of the session. In the following format: language_territory.characterset

Parameter	Explanation
NETWORK_PROTOCOL	Network protocol used
NLS_CALENDAR	The calendar of the current session
NLS_CURRENCY	The currency of the current session
NLS_DATE_FORMAT	The date format for the current session
NLS_DATE_LANGUAGE	The language used for dates
NLS_TERRITORY	The territory of the current session
SERVER_HOST	The hostname of the machine where the instance is running
SESSION_USER	The database username of the user logged in
SID	Session number

Source: https://www.techonthenet.com/oracle/functions/sys_context.php

```
select SYS_CONTEXT('USERENV', 'IP_ADDRESS') from dual;
```

```
select SYS_CONTEXT('USERENV', 'CURRENT_SCHEMA') from dual;
```

2.4 Managing NULL values

If the attribute has no value, it can be said that it is *NULL* or contains *NULL*. Such undefined value can be associated with any data type for any column, which is not restricted to be *NOT NULL* (e.g., *primary key*) by definition. It denotes that actual value is either not known or not meaningful. Several functions have been introduced to deal with *NULL* values, to replace undefined values with another real value in the reports, like *NVL*, *DECODE*, *CASE*, *COALESCE*, ... Comparison to the nullity in the conditions must be done using the *IS NULL* or *IS NOT NULL* keywords.

```
select 1 from dual
where null is null;
```

```
select 1 from dual
where 1 is not null;
```

Be aware, it can never be compared to the equality using the mathematical operator =, !=, <> and so on, because conditions would always be evaluated as NULL and routed to the ELSE clause of the evaluation processing.

```
select 1 from dual
where 1 = null;
```

```
select 1 from dual
where 1 <> null;
```

```
select 1 from dual
where null = null;
```

```
select 1 from dual
where null <> null;
```

All of these four commands return no data:

```
no rows selected
```

Consider the following table consisting of multiple examples characterizing the evaluation condition when *NULL* values are provided.

Tab. 2.4: *NULL* value evaluation

<i>Input value (in_val)</i>	<i>Condition</i>	<i>Evaluation result</i>
5	in_val IS NULL	False
5	in_val IS NOT NULL	True
NULL	in_val IS NULL	True
NULL	in_val IS NOT NULL	False
5	in_val = NULL	NULL
5	in_val <> NULL	NULL
NULL	in_val = NULL	NULL
NULL	in_val <> NULL	NULL
NULL	in_val = 5	NULL
NULL	in_val <> 5	NULL

As you can see, equality or non-equality evaluation always results in a *NULL* value.

Therefore, remember that any arithmetic expression containing a *NULL* value is always evaluated as *NULL*. Thus, all operators (except concatenation) return *NULL* values when using the *NULL* value operand.

```
select 1 + null from dual;
```

```
NULL
```

The concatenation of the strings works a bit differently. In that case, the *NULL* value represents the empty string. Thus, the original string is returned if two string values are concatenated, one of which is *NULL*. If both of them are *NULL*, the returned value is *NULL*. So, to conclude, if the string consists of at least one character, *concatenation* will replace *NULL* values with the empty string. However, if all operands are *NULL*, then a *NULL* value is returned.

```
select 'string text' || null from dual;
```

```
string text
```

```
select null || null from dual;
```

```
NULL
```

2.5 Comparing strings (equality, operator Like)

Condition management based on string values can be done either as equality of the whole value or just its part passing specific delimitation. To provide such functionality, simple conditional evaluation can be done using equality or non-equality definition. Generally, either two string variables (column values) can be evaluated and compared together, or also constant string can be used, which is bordered by the apostrophes. The following example shows how to list only persons whose name is “*Michael*”.

```
select name, surname
from personal_data
where name = 'Michael';
```

```
-- 4 rows selected.
```

Then, a similar *Select* statement is used, but the non-equality is tested. What about the number of rows selected and the cardinality of the table? Is there any problem? If so, why?

```
select name, surname
  from personal_data
  where name <> 'Michael';           -- 38 rows selected.
```

The first *Select* statement gets 4 rows. The second *Select* statement returns 38 rows. However, the total number of rows in the *personal_data* table is 43. Thus one row is missing. Where is the problem? Look at the model and schema of the table. Sure, the problem resides in the attribute definition, whereas it can hold *NULL* values. In that case, if the row tuple consists of a *NULL* value for the attribute *name*, it does not pass the condition to be part of the first result set, neither the second one. To see the problem explicitly, extend the first *Select* statement by *NULL* value evaluation condition. As you can see, the result set will be extended by one row. And that's the solution.

```
select name, surname
  from personal_data
  where name = 'Michael' or name is null;   -- 5 rows selected.
```

Comparison of the string format data values can also be made using the *Like operator*. In that case, individual characters can be modeled, respectively replaced during the evaluation by the wildcards. Two types can be distinguished:

- **Percentage (%)**, which characterizes any string length (also empty string is covered).
- **Underscore (_)**, which delimits precisely one character.

Let's consider the following example. The first *Select* statement will return all names and surnames of the people whose first name starts with “Carol”. So, if “Caroline” is present in the database, such rows will be listed, too.

```
select name, surname
  from personal_data
  where name like 'Carol%';
```

If you want to get the name list (name, surname) of the persons whose first character of the name is “J”, multiple solutions can be used (*Like operator* vs. *substr* function).

```
select name, surname
  from personal_data
  where name like 'J%';
```

```
select name, surname
  from personal_data
  where substr(name, 1, 1) = 'J';
```

However, if you want to limit the result set to only name list of the persons, whose first character of the name is “J”, but the *length* of the name is 4 (*John* and *Jack* are present, but *Jacob* not), if you use *substr* function, it must be then extended by the second condition managing string *length*.

```
select name, surname
  from personal_data
  where substr(name, 1, 1) = 'J' and length(name) = 4;
```

Like operator can provide such requirements using only one condition based on defined wildcard *underscore* (`_`).

```
select name, surname
from personal_data
where name like 'J_';
```

These wildcards can be used anywhere in the string, so if you want to get the list of persons (*name*, *surname*) whose *name* contains the character “o”, the solution can look like the following:

```
select name, surname
from personal_data
where name like '%o%';
```

In the previous case, the character “o” is inside the name, but the name cannot start with that character (whereas the value for the *name* starts with an uppercase character). To get the right solution, process the string value regardless of the size of the character (by using the *lower* function):

```
select name, surname
from personal_data
where lower(name) like '%o%';
```

The problem of the *Like* operator is just management of the *underscore* and *percentage* sign inside the string (as a standard character) to be evaluated. In that case, it is necessary to differentiate between attribute value and part of the *Like* operator itself by using the *Escape* clause of the operator. Let's have the table *Tab_like* with one attribute (*str*) holding these three values:

- *abcd_f*
- *abcdef*
- *abcd_fgh*

The aim is to evaluate the fifth character of the string. Following *Select* statement will list the first and the last value, whereas it requires the fifth character to be *underscore* irrespective of the string length. Thus, the *percentage* symbol is considered as part of the *Like* operator, while the *underscore* is considered as standard string character (it is preceded by escape separator). The solution looks like the following:

```
select * from tab_like
where str like 'abcd\_%' escape '\';
```

The last possibility for dealing and comparing string values is based on regular expressions.

2.6 Using Order By clause

The result set of the defined *Select* statement is not automatically sorted. Data are returned based on their location in the memory or physical storage. Using **Order By** clause allows you to sort the data either *ascending* (by using *ASC* keyword (*default option*)) or *descending* (by using *DESC* keyword). Naturally, multiple elements can be listed delimited by the comma. The element name in *Order By* clause can be either original name, or column alias can be used, if defined, too.

```
... order by column_list [{ASC | DESC}] [, ...]
```

```
select name, surname as familyname
from personal_data
order by familyname, name;
```

```
select name, surname as familyname
from personal_data
order by surname, name;
```

Moreover, also sequential numbers (order) of the element in the *Select* clause can be used. So, it is sorted based on the *surname* (second attribute of the *Select* statement) followed by the attribute *name*.

```
select name, surname
from personal_data
order by 2, 1;
```

In DBS Oracle, the result set can also be sorted by using attributes not listed in the *Select* clause. However, it is not a general rule for the whole database system spectrum.

```
select name, surname
from personal_data
order by personal_id;
```

As described sooner, undefined values – *NULL* – cannot be compared, which also affects the *Order By* clause. By default, *when comparing, NULL values are considered the highest values*, so they are listed first for descending sort and last for ascending sort. However, such an approach can be changed by using ***NULLS FIRST*** or ***NULLS LAST*** keywords. Thus, in the first and second examples, *NULL* values will be listed at the end, the third and fourth examples, *NULL* values are listed first.

```
select field_id, specialization_id, field_name, spec_name
from st_field
order by spec_name, field_name;
```

```
select field_id, specialization_id, field_name, spec_name
from st_field
order by spec_name DESC NULLS LAST, field_name;
```

FIELD_ID	SPECIALIZATION_ID	FIELD_NAME	SPEC_NAME
200	3	Information systems	Information and communication systems
200	1	Information systems	Decision support systems
200	2	Information systems	Applied informatics
202	0	Computer engineering	(null)
200	0	Information systems	(null)
102	0	Management	(null)
201	0	Information management	(null)

```
select field_id, specialization_id, field_name, spec_name
from st_field
order by spec_name NULLS FIRST, field_name;
```

```
select field_id, specialization_id, field_name, spec_name
from st_field
order by spec_name DESC, field_name;
```

FIELD_ID	SPECIALIZATION_ID	FIELD_NAME	SPEC_NAME
101	0	Computer engineering	(null)
202	0	Computer engineering	(null)
100	0	Informatics	(null)
201	0	Information management	(null)
200	0	Information systems	Applied informatics
102	0	Management	(null)
200	3	Information systems	Information and communication systems
200	1	Information systems	Decision support systems

2.7 Table joining

In the previous definitions, we have dealt only with attributes based on one table. Generally, listed attributes can be from any table. For these purposes, it is necessary to highlight the relationships between the tables to connect tables together. Joining is based on a *primary key* (described in chapter 2.1 [Introduction](#) and chapter 4.4 [Entity-relational conceptual model](#)) of the referenced table and *foreign key*.

A *foreign key* is a key used to link two tables together. It is a field (or collection of fields) in one table that refers to the *primary key* of another table (or can hold a *NULL* value, if not constrained). The table containing the *foreign key* is called the *child table*, and the table containing the *primary key* is called the *referenced* or *parent table*. More about *primary* and *foreign* keys can be found in [Lab 4 – Data modeling](#).

Several opportunities and switches are influencing the *Join* type. In this section, we will deal only with *INNER Join*. Other options will be provided in chapter 8.6 [Extended versions of table joining](#).

This is the syntax of the *Join* operation.

```
select ...
  from table_name1 [ { INNER | {LEFT | RIGHT | FULL} [OUTER] } ] JOIN
    table_name2 { ON(joining_conditions) | USING(column_list) }
    [ { INNER | {LEFT | RIGHT | FULL} [OUTER] } ] JOIN
    table_namen { ON(joining_conditions) | USING(column_list) }
```

The following figure shows the graphical representation of the relational algebra operation *Join*.

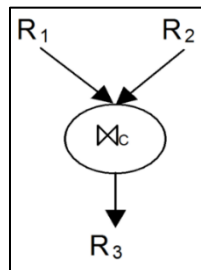


Fig. 2.3: Relational algebra operation *Join*

Inner Join selects all rows from both tables if they match the *joining* criterion (*foreign key references the primary key of another table*). Thus, if you *inner join* the table *personal_data* and *student*, no information about the persons who are not students (or have never been) will be listed. Notice the cardinality of the following *Select* statements.

The cardinality of the second *Select* statement (*distinct* keyword remove duplicates) is smaller than the cardinality of the third statement, whereas each person can be listed multiple times in the student table. Moreover, a person does not need to be listed as a student (the difference between statement 1 and statement 2). On the other hand, the third and fourth statement cardinality is always the same. A student cannot exist without *personal_data* information (obligatory membership of the relationship – as a consequence, attribute *personal_id* in table student is *NOT NULL*).

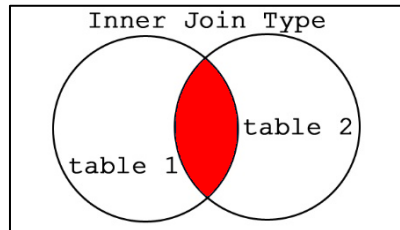


Fig. 2.4: Inner Join

```
select personal_id from personal_data; -- 35 rows selected.
```

```
select distinct personal_id from student; -- 33 rows selected.
```

```
select personal_id, student_id
from personal_data join student using(personal_id);
-- 37 rows selected.
```

```
select personal_id from student; -- 37 rows selected.
```

When tables are joined together, **ON** or **USING** keywords must be used to define joining criteria. **ON** keyword can be used anytime. **USING** keyword can be used only if the names of the primary and foreign key attributes are the same. Also, notice the difference in the result sets. When **ON** is used, individual attributes to be joined are present from both tables. The difference between them is based on the table origin. Therefore, table names or table aliases must be used to differentiate them. Both values expressing *personal_id* (from *personal_data* as well as *student* table) are always the same.

```
select name, surname, student.personal_id, personal_data.personal_id
from personal_data
join student on (personal_data.personal_id = student.personal_id);
```

NAME	SURNAME	PERSONAL_ID	PERSONAL_ID_1
Michael	Pearce	841106/3456	841106/3456
Jack	Smith	840312/7845	840312/7845
John	Young	860907/1259	860907/1259
Carol	Pearce	850130/3695	850130/3695
Peter	Roger	781015/4431	781015/4431

Vice versa, by using **USING** keyword, the output set consists of such attributes only once, so table names nor aliases for such attributes can be used. Highlight the following example and compare it to the previous one.

```
select name, surname, personal_id
from personal_data join student using (personal_id);
```


NAME	SURNAME	PERSONAL_ID
Michael	Pearce	841106/3456
Jack	Smith	840312/7845
John	Young	860907/1259
Carol	Pearce	850130/3695
Peter	Roger	781015/4431

If the *Using* clause for table joining is used, a particular attribute CANNOT be prefixed by table name or table alias.

```
select name, surname, personal_id
  from personal_data join student using(personal_id);
```

```
select name, surname, student.personal_id
  from personal_data join student using(personal_id);
```

Generally, any number of tables can be joined together. The order of the joining is commonly not important and database optimizer selects the most suitable plan for the execution.

Be aware of the primary key definition, specifically when dealing with composite primary keys. The whole primary key must be joined, otherwise, a *Cartesian product* (system combines each element of the first table with each element of the second table, see chapter 2.8 *Cartesian product*) will be created. Thus, if you want to list each student's field name and specialization name, table *st_field* and *student* must be joined together based on the composite primary key.

When *USING* keyword is used, individual attributes are delimited by the *comma* (,).

```
select student_id, field_name, spec_name
  from student join st_field using(field_id, specialization_id);
```

When the *ON* keyword is used, joining conditions are delimited by the *AND*.

```
select student_id, field_name, spec_name
  from student join st_field on (st_field.field_id = student.field_id
                                AND
                                st_field.specialization_id = student.specialization_id);
```

Be effective when joining tables. The fundamental principle is straightforward – **avoid not necessary JOINS**. It can significantly impact the performance, mainly if the amount of data is relatively high.

Let's consider the following example. We want to list *students* and their *registered subjects*, where the number of credits (*ects*) is not the same as *studying plans*. Therefore, student identifier, subject identifier, and values of credits (expected and real) are selected. In the following example (fig. 2.5), three tables are joined together – *student*, *subject*, and *subject_year*.

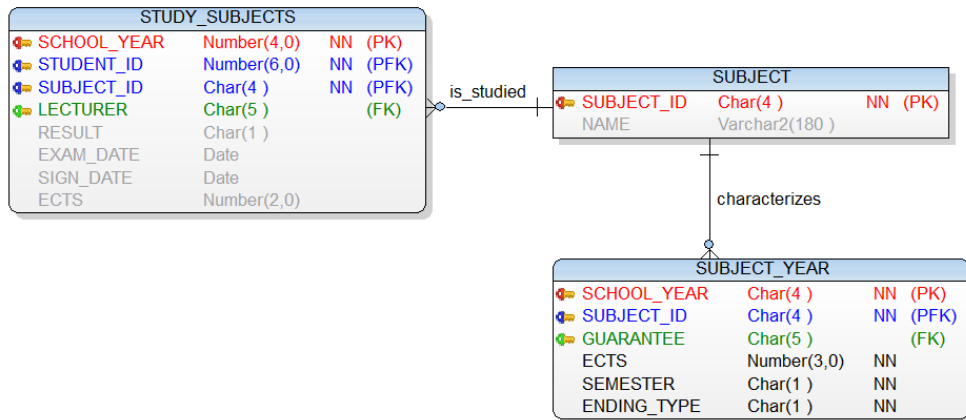


Fig. 2.5: Data model – Study_subjects, Subject, Subject_year

```

select student_id, subject_id,
       study_subjects.ects as real_ects,
       subject_year.ects as expected_ects
from study_subjects join subject using(subject_id)
       join subject_year using (subject_id)
where subject_year.school_year = study_subjects.school_year
       and study_subjects.ects != subject_year.ects;
  
```

The results are correct; however, do you need any data from the *subject* table? The answer is *NO*. Therefore, do not join it:

```

select student_id, subject_id,
       study_subjects.ects as real_ects,
       subject_year.ects as expected_ects
from study_subjects join subject_year using(subject_id)
where subject_year.school_year = study_subjects.school_year
       and study_subjects.ects != subject_year.ects;
  
```

Referenced values are composite, therefore put *school_year* management to the *JOIN* operation.

```

select student_id, subject_id,
       study_subjects.ects as real_ects,
       subject_year.ects as expected_ects
from study_subjects join subject_year using(subject_id, school_year)
where study_subjects.ects != subject_year.ects;
  
```

2.8 Cartesian product

A *Cartesian Join* or *Cartesian Product* is a join of every row of one table to every row of another table (fig. 2.6). This happens typically when no matching join columns are specified, or you refer to an incomplete key. For example, if *table A* with *100* rows is joined with *table B* with *1000* rows, a *Cartesian Join* will return *100,000* rows.

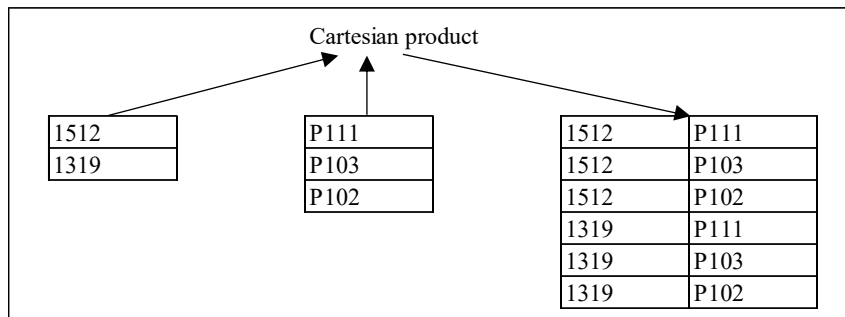


Fig. 2.6: Cartesian product

A Cartesian product may indicate a missing join condition. A query must have at least $(N-1)$ join conditions to prevent a *Cartesian product*, where N is the number of tables in the query. The visual denotation of the *Cartesian product* is following:

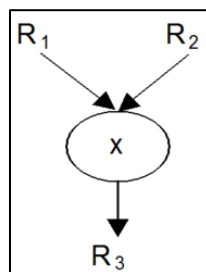


Fig. 2.7: Relational algebra operation – Cartesian product

The composite primary key is significant for *joining*. The whole primary key definition must be used. Otherwise, a *Cartesian product* is generated. What will happen if you want to get the full name of the *field* and *specialization* of this study? The correct solution looks like this (*USING / ON*) – 37 rows are selected.

```
select * from student join st_field using(field_id, specialization_id);
```

	STUDENT_ID	FIELD_NAME	SPEC_NAME
27	500425	Information systems	Information and communication systems
28	500426	Information systems	Information and communication systems
29	500427	Information systems	Information and communication systems
30	500428	Information systems	Information and communication systems
31	500429	Information systems	Information and communication systems
32	500430	Information systems	Information and communication systems
33	500431	Computer engineering	(null)
34	500432	Information systems	Information and communication systems
35	500433	Information systems	Information and communication systems
36	500438	Information systems	Applied informatics
37	500439	Information systems	Applied informatics

If you omit one attribute, the following data are produced, which are incorrect (such students do not study all specializations in the defined field at all). Therefore, be strictly aware, when joining composite primary key tables. 88 rows were selected, although table student contains only 37 rows!

```
select * from student join st_field using(field_id);
```

	STUDENT_ID	FIELD_NAME	SPEC_NAME
78	500433	Information systems	Applied informatics
79	500433	Information systems	Decision support systems
80	500433	Information systems	(null)
81	500438	Information systems	Information and communication systems
82	500438	Information systems	Applied informatics
83	500438	Information systems	Decision support systems
84	500438	Information systems	(null)
85	500439	Information systems	Information and communication systems
86	500439	Information systems	Applied informatics
87	500439	Information systems	Decision support systems
88	500439	Information systems	(null)

However, sometimes, the *Cartesian product* can be useful. The following example shows all subjects, which can be registered for any student. It is necessary to distinguish between attributes “name” from the table *subject* and *personal_data*. Therefore, the table name must prefix the attribute name.

```
select personal_data.name, surname, personal_id, subject_id,
       subject.name
from personal_data, subject;
```

2.9 SETs operations (IN, EXISTS)

Controlling the existence or non-existence of the particular data in another table or the named set can be done using set definition, which is consequently compared to the defined attribute expression. When the named list is used, individual values are listed, enclosed by the parentheses, and delimited by commas.

```
select name, surname
from personal_data join student using(personal_id)
where class in (1, 2, 3);
```

Set can be, however, also formed by the nested *Select* statement. Such a solution is mainly used to evaluate the data existence in another table based on *referential integrity* (*primary* and *foreign keys*). Notice that existence determination and evaluation can also be processed using table *JOINing*. However, it is not so effective due to a more significant amount of data. Vice versa, evaluating non-existence cannot be managed by table *Inner JOINing* at all, whereas we would lose the data we are searching for. Therefore, set operations using *IN* and *EXISTS* have been proposed with also their negative meaning – *NOT IN* / *NOT EXISTS*.

Let’s have a simple example to get a named list of the persons who have never studied anything – data about the person are not referenced in the student table. To write such a query, several operations are performed. Therefore, we will describe principles step by step. First of all, realize that the tables *personal_data* and *student* cannot be joined together using the *personal_id* attribute (we are looking for rows, which cannot be joined ☺). Thus, get the list of the *personal_id* values, which are part of the student table. The proposed design would look like this:

```
select personal_id from student;
```

Then, get the name list of the persons from the *personal_data* table.

```
select name, surname from personal_data;
```

Now, link two proposed *Select* statements together. What does it mean that person has not studied yet? His *personal_id* value is not present in the *student* table. Thus, the solution can look like this:

```
select name, surname
from personal_data
where personal_id NOT IN (select personal_id
                           from student);
```

That's the first working solution. Individual *personal_id* attribute values are compared together. It can be, however, rewritten by using the *NOT EXISTS* set operator. In that case, the solution can look like the following. Whereas no direct comparison is made from the inner *Select* clause, constant string "x" is used.

```
select name, surname
from personal_data
where NOT EXISTS
      (select 'x'
       from student
        where personal_data.personal_id = student.personal_id);
```

Be aware, never forget to add the *Where* clause, which compares the values from the main and nested *Select* statement using *IN* set operator. It is inevitable, and many coders forget to code it. However, it leads to incorrect data. If the connection between both *Select* statements is not present, all table data will be listed or none. In principle, if the table *student* is empty, data about all *persons* will be listed. Vice versa, if table *student* consists of at least one row, none will be listed. Let's consider, therefore, the following example. There is no connection between individual *Select* statements. Thus, no rows will be selected, whereas we have some data portions in the *student* table. Be aware. It is an example of incorrect usage, do not use it like this.

```
select name, surname
from personal_data
where NOT EXISTS (select 'x'
                   from student);
```

Generally, operator *IN* can always be rewritten to *EXISTS* operator, but it is not valid from the opposite side. If multiple attributes need to be evaluated (e.g., composite primary key), *EXISTS* set type should be used. The following example is correct.

```
select field_name, spec_name
from st_field
where NOT EXISTS (select 'x'
                  from student
                   where st_field.field_id = student.field_id
                     AND
                     st_field.specialization_id = student.specialization_id);
```

Notice that new releases of the *DBS Oracle* allow you to compare multiple attributes by using *IN* set. However, it is not standard functionality, and many other database systems do not support it.

```
-- ONLY IN NEW ORACLE DBS RELEASES:
select field_name, spec_name
from st_field
where (field_id, specialization_id) NOT IN
      (select field_id, specialization_id from student);
```

Controlling the existence and management of composite primary key must be done *as the whole set*. It cannot be handled by multiple single element sets. The characteristics and differences are shown in the following example. Let's have two tables consisting of these data (fig. 2.8). Both tables have *two attributes*, and the aim is to find the combination of values from table *T2*, which are not present in table *T1*.

	ID	ID2					
	1	1					
	1	2					
	1	3					
	2	1					
	2	2					
	2	3					

T1

|

	VAL1	VAL2	
1	1	1	
2	2	2	
3	3	3	
4	4	2	

T2

Fig. 2.8: Set comparison

As you will see in the following part, such a solution is not correct at all.

```
select *
from T2
where val1 NOT IN (select id from T1)
and val2 NOT IN (select id2 from T1);
```

First, let's evaluate the first condition: *val1 NOT IN (select id from T1)*. In this case, the third (*val1=3 ; val2 = 3*) and forth (*val1=4 ; val2 = 2*) rows of the table *T2* meet the condition. Then, evaluate the second condition based on attribute *val2*: *val2 NOT IN (select id2 from T1)*. All values of the attribute *val2* are present in table *T1* – attribute *id2*. Therefore, no rows would be selected. Mapping is shown in fig. 2.9.

	ID	ID2					
1	1	1					
2	1	2					
3	1	3					
4	2	1					
5	2	2					
6	2	3					

T1

|

	VAL1	VAL2	
1	1	1	
2	2	2	
3	3	3	
4	4	2	

T2

Fig. 2.9: Set comparison

A different situation arises if the pair is compared using one condition *NOT EXISTS*. In this case, values (1,1) and (2,2) are present. However, the combination of values (3,3) and (4,2) are not present. Thus, the result of the following query is (3,3) and (4,2). Such values are compared as a pair, not separately.

```
select *
  from T2
     where NOT EXISTS (select 'x'
                        from T1
                        where val1 = id
                        and val2 = id2);
```

VAL1	VAL2
1	1
2	2

Please note that working with the set *IN* does not work correctly by comparison of *NULL* values.

As mentioned earlier, *NULL* values cannot be compared directly using mathematical operators. Let's have the table **contact** consisting of the contacts of the persons in the *personal_data* table. The structure of the *contact* table row is following:

Name	Null	Type
-----	----	-----
CONTACT_ID		NUMBER
PERSONAL_ID		CHAR(11)
TYPE		CHAR(1)
VALUE		VARCHAR2(50)

If the foreign key value (attribute *personal_id* of the *contact* table) can contain *NULL* values, incorrect results will be provided using set type *IN*. Consider that *three* people do not have contacts. Some of them have multiple. Moreover, two contacts are stored without *personal_id* reference. Therefore, the following *Select* statement will provide **no data**.

```
select *
  from personal_data
     where personal_id NOT IN (select personal_id
                              from contact);
```

To get correct results, *NULL* values must be removed, either by the *Where* clause or by the transformation to another existing value (*NLV*, *CASE*, *DECODE*, ...).

```
select *
  from personal_data
     where personal_id NOT IN (select personal_id
                              from contact
                              where personal_id IS NOT NULL);
```

	PERSONAL_ID	NAME	SURNAME	STREET	TOWN	ZIP	NATIONALITY
1	921225/7452	Sim	Eas	Kolarovce 12	Kolarovce	01401	SK
2	830324/7887	Daniel	Gomes	Razusa 40/10	Prievidza 4	97101	SK
3	860103/2238	John	Young	Bratislavská cesta 2	Zilina	01001	SK

When the *EXISTS* set type is used, *NULL* values are not problems because of the *Where* clause definition, which automatically removes such non-joinable *NULL* data.

```
select *
  from personal_data
     where NOT EXISTS
           (select 'x'
            from contact
             where contact.personal_id = personal_data.personal_id);
```

	PERSONAL_ID	NAME	SURNAME	STREET	TOWN	ZIP	NATIONALITY
1	921225/7452	Sim	Eas	Kolarovce 12	Kolarovce	01401	SK
2	830324/7887	Daniel	Gomes	Razusa 40/10	Prievidza 4	97101	SK
3	860103/2238	John	Young	Bratislavská cesta 2	Zilina	01001	SK

2.10 Managing duplicate values

Duplicate value management is a significant part of data management. A person can study multiple times (can be present in the table student numerous times), each subject can be studied more than once by the same person (if the previous attempt to pass the exam failed). *Distinct* keyword removes the duplicate tuples from the output.

The following example shows the principles of using the *Distinct* keyword applied to the result set before returning to the user.

Let's consider the following example, person *Jack Robinson* is listed multiple times:

```
select name, surname, personal_id
  from personal_data join student using(personal_id);
```

NAME	SURNAME	PERSONAL_ID
Peter	Roger	781015/4431
Jack	Robinson	791229/5431
Jack	Robinson	791229/5431
Jack	Robinson	791229/5431

To remove the duplicate data tuples from the result set, the *distinct* keyword can be used. In that case, *Jack Robinson* will be listed only once, whereas it reflects the same person (the same *personal_id* value). If the values of *personal_id* were not the same (they would be only namesakes – same name and surname), these data would both be in the result set.

```
select distinct name, surname, personal_id
  from personal_data join student using(personal_id);
```

NAME	SURNAME	PERSONAL_ID
Peter	Roger	781015/4431
Jack	Robinson	791229/5431
Mark	Bailey	800407/3522

The previous result set will consist of the *name*, *surname*, and *personal_id* of the persons who started studying at least once. The person will be listed only once, regardless of the number of his data in the student table. Thus, such *Select* statement can be rewritten using set operators providing the same results. Solution:

```
select name, surname, personal_id
  from personal_data
     where personal_id IN (select personal_id
                          from student);
```



```
select name, surname, personal_id
from personal_data
where EXISTS (select 'x'
              from student
              where
                student.personal_id = personal_data.personal_id);
```

However, if you remove the *personal_id* attribute and apply the rule for duplicate tuples removing, incorrect data will be provided due to namesakes – in the result set, e.g. person *Milan Clarke* is listed only once, although it references two persons with the same name and surname – they have different *personal_id* attribute values.

```
select distinct name, surname
from personal_data join student using(personal_id);
```

Milan	Clark	840409/7900
Milan	Clark	840410/6777



Milan Clarke

Fig. 2.10: Select statement result set

So to conclude, the *distinct* keyword is vital for the processing, but be aware of using the unique identifier to remove possibilities to get non-reliable data. If you want to use the *Distinct* operation without providing *personal_id* to the result set, two phases must be done. Firstly, remove duplicates dealing with the trinity (*name*, *surname*, *personal_id*), then list just the first two attributes by nesting the query.

2.11 Table alias

Besides the attribute (column) alias, also table name can be aliased to simplify the naming and improve code readability. In the previous chapter, if the attribute must be enhanced by the table name (to distinguish which attribute is involved), the fully qualified name has been used – the name of the attribute has been preceded by the table name.

```
select name, surname, personal_id
from personal_data
where EXISTS
  (select 'x'
   from student
   where student.personal_id = personal_data.personal_id);
```

The alias name can be placed directly after the name of the table. It replaces the original table name in the defined query. Therefore, alias must be used instead of the original table name itself. Let's consider the following examples:

```
select name, surname, s.personal_id
from personal_data p join student s on (p.personal_id = s.personal_id);
```

```
select name, surname
from teacher t join study_subjects ss on (t.teacher_id = ss.lecturer)
where ss.school_year = 2008;
```

```
select p.name || ' ' || p.surname as student,
       t.name || ' ' || t.surname as teacher
from personal_data p join student s using(personal_id)
join study_subjects ss using(student_id)
join teacher t on (ss.lecturer = t.teacher_id);
```

Be aware if table alias is defined, original table name cannot be used at all. So, the following example will raise an *exception*:

```
select name, surname, student.personal_id
from personal_data p
join student s on (personal_data.personal_id = student.personal_id);
```

2.12 Practice

1. List all the *data* about the *students*.
2. *Select the name list of the second-class students.*
3. *Select the name list of the students born in 1985 – 1989 (year).*
4. *Select the name list of the students who study in the detached office in the Slovak town Prievidza (the second character of the studying group is “P”).*
5. *Order the previous results based on surname.*
6. *Select the name list of the students studying the subject “BI06” and sort them.*
7. List all combinations of the *lecturer / subject_id*. Remove *duplicates*.
8. Extend the previous *Select* statement by the *name of the teacher* and the *name of the subject*.
9. Select the *name list* of the *lecturers* for *subjects* taught during the *second class* of the *bachelor study* (the number of *field_id* belongs to <100; 199>).
10. *Select the subject names that are studied by the student “Smith”.*
11. Get the *number of the rows* in the table “*study_subjects*”.
12. Select the *name list* of the people studying the subject “*Database systems*”. Distinguish the different school years.
13. *Select the name list with the date of birth.*
14. *Select the number of the student's credits with student_id = 500439 based on the successfully passed exams.*
15. *Select the name list of the students in the second class together with their actual age.*

Dealing with the Cartesian product

(Solution code is described at the end of this lab):

1. Get the list of all *subjects*, which each person can register during the study.
2. For each *student* (actual or ended successfully), list all the *subjects* which can be registered to him – only those can be selected, which have not been passed successfully yet. For simplicity, evaluate the subject passing based on exam results.
 - student.status:
 - S = student (actual),
 - E = ended successfully,
 - A = aborted,
 - X = fired due to disciplinary commission decision.
3. Extend the previous solution by covering the *ending type* of the subject:
 - attribute **ending_type** of the **subject_year** table:
 - B = exam + accreditation to exam,
 - E = exam,
 - S = semester only (no exam).

Cartesian product – solutions

1. **Get the list of all subjects, which each person can register during the study.**
The solution is based on using two tables – *personal_data* and *subject* without any *JOINS*. Do not forget to use column alias and attribute qualified name.

```
select pd.name as name, surname, subject_id,
       subj.name as "NAME OF THE SUBJECT"
from personal_data pd, subject subj;
```

2. **For each student, list all the subjects, which can be registered for him – only those can be selected, which have not been passed successfully yet. For simplicity, evaluate the subject passing based on exam results.**

Also, table *personal_data* and *subject* is used without any *JOINS*. Then, two conditions are evaluated. The first one is characterized by the set type *IN* to ensure that the person is an actual student (*status='S'*).

```
personal_id IN (select personal_id
                from student
                where status = 'S')
```

The second condition limits the list of the subjects possible to be registered by a particular person. The linking is based on the *personal_id* attribute, whereas the *subject* can be passed only once by one specific *person* regardless of the number of his studies (actual or correctly ended). Thus, if some subject has been passed successfully, it cannot be registered later by the same person (if the referenced study has been either completed (*status='E'*) or stated as an actual student (*status='S'*)). The passed exam, in this case, is delimited by the exam result – *A*, *B*, *C*, *D*, and *E*.

```
NOT EXISTS (select 'x'
            from student stud join study_subjects ss using(student_id)
            where pd.personal_id = stud.personal_id
            and subj.subject_id = ss.subject_id
            and ss.result in ('A', 'B', 'C', 'D', 'E')
            and status in ('S', 'E'));
```

Thus, the complete *Select* statement can look like this:

```
select pd.name as name, surname, pd.personal_id,
       subject_id, subj.name "NAME OF THE SUBJECT"
from personal_data pd, subject subj
where personal_id IN (select personal_id
                    from student
                    where status = 'S')
and NOT EXISTS
  (select 'x'
   from student stud join study_subjects ss using(student_id)
   where pd.personal_id = stud.personal_id
   and subj.subject_id = ss.subject_id
   and ss.result in ('A', 'B', 'C', 'D', 'E')
   and stud.status in ('S', 'E'));
```

3. **Extend the previous solution by covering the ending type of the subject:**
 - attribute **ending_type** of the **subject_year** table:
 - B = exam + accreditation to exam,
 - E = exam,
 - S = semester only (no exam).

In this case, the condition to evaluate the subjects is more complex. In principle, three situations based on ending type can occur, which must be handled:

- 1) Subject ends with semester only (*ending_type* = 'S'). In this case, if the student has a value of *sign_date*, the particular subject is passed successfully.

```
ending_type = 'S' and sign_date IS NOT NULL
```

- 2) Subject ends with exam only (*ending_type* = 'E'). In this case, two spheres must be managed – exam result and exam date must be filled. For exam results, only values *A*, *B*, *C*, *D*, and *E* are suitable.

```
ending_type = 'E'
and sign_date IS NOT NULL
and result in ('A', 'B', 'C', 'D', 'E')
```

- 3) Subject ends with the exam as well as accreditation to the exam. In that case, *exam_date* and *sign_date* must be filled. Moreover, correct exam results must be provided – only values *A*, *B*, *C*, *D*, and *E* are suitable.

```
ending_type = 'B'
and sign_date IS NOT NULL
and exam_date IS NOT NULL
and result in ('A', 'B', 'C', 'D', 'E')
```

All other cases are considered as an unsuccessful subject ending.

So, the complete solution can look like this. Notice that the tables *study_subjects* and *subject_year* can be directly joined without dealing with the *subject* table. However, be aware of two attributes to be joined together (*subject_id* and *school_year*). Reference to the outer *Select* statement is done using the *subject_id* attribute. Therefore, the *ON* option must be used for *JOIN*.

```
select pd.name as name, surname, pd.personal_id,
       subject_id, subj.name "NAME OF THE SUBJECT"
from personal_data pd, subject subj
where personal_id IN (select personal_id
                     from student
                     where status = 'S')
and NOT EXISTS
(select 'x'
 from student stud
  join study_subjects ss using(student_id)
  join subject_year sy on(ss.subject_id = sy.subject_id
                        and ss.school_year = sy.school_year)
 where pd.personal_id = stud.personal_id
       and subj.subject_id = ss.subject_id
       and status in ('S', 'E')
       and ((ending_type = 'S' and sign_date IS NOT NULL)
            or (ending_type = 'E'
                and sign_date IS NOT NULL
                and result in ('A', 'B', 'C', 'D', 'E'))
            or (ending_type = 'B'
                and sign_date IS NOT NULL
                and exam_date IS NOT NULL
                and result in ('A', 'B', 'C', 'D', 'E'))))
);
```

Lab 3 – Insert, Update, Delete statements and transactions

This lab deals with the data manipulating operations modifying the database. Namely, adding new tuples is operated by the Insert statement, which can be specified explicitly by listing values (loading one row per command) or composed by the Select statement. The Update statement modifies the existing rows. The Delete statement removes the rows from the tables. The key fact is that each command can reference only one table, thus, no Joins are available there!

The second part of the chapter deals with the referential integrity operated by the primary and foreign keys. These are the core elements dealing with the integrity and availability to interconnect multiple tables.

Whereas each statement is part of the transaction, after the processing, it must be ended either by approving it (making the data changes durable and available across multiple sessions and transactions) or by refusing it (getting original data states).

3.1 Introduction

Data manipulation language (DML) is a family of syntax command elements covering *Insert*, *Update*, *Delete* and *Select* statements. In this lab, we will deal with destructive operations, which modify data stored in the database. However, also *Select* statement will be used as part of the conditions. It is necessary to emphasize that *Insert*, *Update*, and *Delete* can always manage **ONLY ONE TABLE**. If the condition is based on another table, the nested *Select* statement must be used (operated by the *IN* or *EXISTS* operation set reference). Once again, no *join* operators can be used inside destructive DML. *Insert* statement is used for adding new row tuples to the database. The *Update* reflects changing existing values and *Delete* removes rows from the database (*always the whole row!*). The next part describes the syntax, possibilities, and principles of usage.

3.2 Insert statement

```
insert into table_name[(list_of_attributes)]
{
    values(list_of_values)
    |
    select ...
};
```

Above, *Insert* statement syntax is defined, where ***list_of_attributes*** represents the list of attributes, the order is essential and reflects the appropriate values inside the ***list_of_values*** set. The clause ***list_of_attributes*** is *optional*. If not written explicitly, the order is defined by the table definition (*desc table_name*), and all attribute values must be added (with the assumption that they are not set using a trigger). By using the *Insert* statement, at least **NOT NULL** values without default values must be defined directly. Moreover, constraints must be met (*primary key*; the *foreign key* must reflect the *primary key* (or *unique index*, respectively)). Otherwise, the data will not be inserted.

Primary key is a specific set of attributes associated with the table, which uniquely identifies each record in a database table. The *primary key (PK)* must contain *UNIQUE* values and cannot hold *NULL*. A table can have only one primary key, consisting of single or multiple fields (*composite primary key*). It must be also minimal (by removing any attribute from the primary key set, aspect of uniqueness would be lost). More about the primary key definition, management, and importance will be described later in chapter 11.3.2 Primary key.

When dealing with the *Insert* statement, it is necessary to distinguish the usage and limitations of the *Values* clause in comparison with the *Select* statement inside it. *Values* clause is used for adding specific values explicitly – constant strings. One statement with a *Values* clause can add only one row. Using the *Select* clause allows you to *Insert* the result set of the *Select* statement into the table. One statement with used *Select* statement can add multiple rows to the table. Be aware, do not combine the *Values* clause with *Select* inside one statement, although it is possible to define it like that in some cases.

3.2.1 Insert – values type

Let's get the structure of the *personal_data* table using the *desc* command.

```
desc personal_data
```

Name	NULL?	Type
-----	-----	-----
PERSONAL_ID	NOT NULL	CHAR(11)
NAME		VARCHAR2(15)
SURNAME		VARCHAR2(15)
STREET		VARCHAR2(20)
TOWN		VARCHAR2(50)
ZIP		CHAR(5)
NATIONALITY		CHAR(2)

In the first type of statement – the order of inserted attributes is not managed *explicitly*. In that case, the order, *number, and data types are delimited by the table structure definition*. Therefore, values, which should not be inserted, must be set as *NULL* explicitly. Naturally, it is possible to define *NULL* value only if no specific (*NOT NULL*) constraints are defined.

```
insert into personal_data
values('905612/8576', 'Michael', 'Flower', null, null, null);
```

If the list of attributes is noted, the order of values (*list_of_values*) must be the same as the order of attribute definition in the *list_of_attributes*. Thus, the attribute values of *street*, *town*, *zip*, and *nationality* are not specified in the following example. Therefore, they will be set to *NULL* automatically (generally, *default value* can be specified to replace undefined value).

```
insert into personal_data(name, surname, personal_id)
values ('Michael', 'Flower', '905612/8576');
```

Be aware. One *Insert-values* statement can insert **ONLY ONE ROW** to **ONE TABLE**. **There are no JOINS allowed.**

3.2.2 Insert – Select type

The disadvantage of the *Insert-values* statement type is that it can insert only *one row* to the table. Thus, if we have an auxiliary table with data to be inserted into the main table, it would be necessary to create one new statement for each row. Although it can be done relatively simply by statement generation, it is unnecessary to do it as explained. In addition, it would be complicated, time-consuming, and resource-demanding. Instead, we can use the *Insert-Select* statement type to solve the problem, which allows adding multiple rows to the table by one statement, based on another table data. In this case, again, it is possible to put data only to one table. Thus, no *JOINS* are allowed (only in the embedded *Select* statement).

Insert statement using auxiliary table can look like this:

```
insert into personal_data(name, surname, personal_id)
  select name, surname, pid
  from student_results
  where result in ('P', 'P'); -- passed
```

Notice that there is no *Values* clause.

It is also possible to combine *Select* statement results and constants:

```
insert into student(student_id, personal_id, field_id, specialization_id,
                  status, class, first_date)
  select st_id, pid, field, specialization, 'S', 1, sysdate
  from student_results
  where result in ('P', 'P');
```

Let's have the following example. We will describe what to do and how to get the desired results step by step. The aim is to add to all *Informatics* students in the first class all *obligatory* subjects for them in the academic year 2016/2017.

So, get the first class student set (*field_id* is 100, *specialization_id* is 0):

```
select student_id
  from student
  where class = 1
        and field_id = 100
        and specialization_id = 0;
```

Get the list of subjects, which should be added to students:

```
select sy.subject_id, sy.school_year, guarantee, ects
  from st_program stp JOIN subject_year sy ON
        (stp.subject_id = sy.subject_id
         and
         stp.school_year = sy.school_year)
  where class = 1 and school_year = 2016
        and field_id = 100 and specialization_id = 0
        and mandatory_type = 'M';
```

Creating a list of all students and subjects, which should be assigned to them. Notice that the *Cartesian product* is used. Legitimate usage is, naturally, allowed. Thus, two *Select* statements are merged. *Student* data are shown in bold.

```

select sy.subject_id, sy.school_year, guarantee, ects, student_id
from student s, st_program stp JOIN subject_year sy ON
      (stp.subject_id = sy.subject_id
       and
        stp.school_year = sy.school_year)
where stp.class = 1 and stp.school_year = 2016
      and stp.field_id = 100
      and stp.specialization_id = 0
      and mandatory_type = 'M'
      and s.class = 1
      and s.field_id = 100 and s.specialization_id = 0;

```

Now, such data can be inserted into the *study_subjects* table.

The complete solution can look like this – the previously defined *Select* statement is encapsulated into the *Insert* statement.

```

insert into study_subjects(subject_id, school_year, lecturer, ects,
                           student_id)
select sy.subject_id, sy.school_year, guarantee, ects, student_id
from student s, st_program stp JOIN subject_year sy ON
      (stp.subject_id = sy.subject_id
       and
        stp.school_year = sy.school_year)
where stp.class = 1 and stp.school_year = 2016
      and stp.field_id = 100
      and stp.specialization_id = 0
      and mandatory_type = 'M'
      and s.class = 1
      and s.field_id = 100 and s.specialization_id = 0;

```

3.3 Update statement

```

update table_name
set attribute = value
[, attribute2 = values2 ...]
[ where conditions ];

```

Using the *Update* statement, we can modify multiple rows and multiple attributes in one statement. Like other DML statements, **only one table can be modified by one statement**. Thus, if two tables are to be updated, at least two statements must be defined.

No *JOINS* are allowed there. Thus, if the condition must be evaluated based on other table data, the *subquery* must be defined.

If new values are in the auxiliary table, the subquery can be used in the **SET** clause.

Be aware, do not forget to add conditions to avoid data loss. Moreover, all new values should be present in the auxiliary table. Otherwise, they will be replaced by *NULL* values.

Let's have the following example. We will change the attribute value for the directly defined row by the primary key. The solution can look like the following. Two attribute values will be changed (*status*, *final_date*). How many rows will be changed? No more than one, because of the condition in the *Where* clause – *student_id* is a unique identifier. If such a *student* does not exist, no rows will be affected. When multiple attributes are updated, these values are delimited by a comma (,), the keyword **Set** is used only once.


```
update student
  set status = 'a',
      final_date = sysdate
  where student_id = 12345;
```

Changes can be based on conditions outside the table. In that case, a nested *Select* statement based on *IN* or *EXISTS* must be used. In the following case, *status* and *final_date* attribute values will be changed for students who graduated last year. The condition is based on the *graduate_students* table. Therefore, a nested *Select* statement must be used.

```
update student
  set status = 'E',
      final_date = sysdate
  where student_id IN (select id
                      from graduate_students
                      where year = to_char(sysdate, 'YYYY') - 1);
```

Also, the new value can be obtained from the *Select* statement. However, do not forget to link data together (by the *Where* clause, *s.student_id* references the table row to be updated). It will ensure that no data will be lost, whereas auxiliary table *new_student* does not need to store the same for the students.

```
update student s
  set st_group = (select new_group
                 from new_student new
                 where new.student_id = s.student_id)
  where exists (select 'x'
               from new_student new
               where new.student_id = s.student_id);
```

However, how to update data based on the composite primary key? The evaluation based on another table should cover several conditions, which **MUST** be evaluated as one composite condition (never try to evaluate composite conditions separately, incorrect results will be provided). Thus, consider the following example. We want to set the value of the *first_date* attribute to actual time for students studying *Informatics*:

```
update student
  set first_date = sysdate
  where first_date is NULL
     and (field_id, specialization_id)
        IN (select field_id, specialization_id
            from st_field
            where field_name = 'Informatics'
              and spec_name is null);
-- ONLY IN NEW RELEASES OF THE DBS ORACLE!!!
```

Notice that mentioned syntax is not part of the *SQL* norm. Therefore, it cannot be used in almost all database system types. However, new versions of the Oracle database systems offer that functionality.

The universal solution uses subquery based on *EXISTS* keyword:

```
update student s
  set first_date = sysdate
  where first_date is NULL
        and exists (select 'x'
                    from st_field stf
                    where field_name = 'Informatics'
                      and spec_name is null
                      and stf.field_id = s.field_id
                      and stf.specialization_id = s.specialization_id);
```

Be aware, never separate composite primary key into separate conditions! It is a significant fault and produces incorrect data (for further information, see chapter 2.9 SETs operations (IN, EXISTS)).

```
update student
  set first_date = sysdate
  where first_date is NULL
        and (field_id) IN (select field_id
                          from st_field
                          where field_name = 'Informatics'
                            and spec_name is null)
        and (specialization_id) IN (select specialization_id
                                   from st_field
                                   where field_name = 'Informatics'
                                     and spec_name is null);
```

3.4 Delete statement

```
delete from table_name
[ where conditions ];
```

One statement can **delete data only from only one table**. If the *Where* clause is omitted, all table data are deleted. Be aware of respecting referential integrity (by default, you cannot delete the row with a *primary key* if other data rows reference it – e.g., you cannot *delete person*, if he is referenced in the *Student* table. Similarly, *student* cannot be deleted if any particular *student_id* is referenced by the *Study subjects* table).

If the condition to be evaluated is based on data from another table, a subquery should be used.

Consider the following examples. All data will be deleted:

```
delete from study_subjects;
```

Rows are deleted based on the same table condition:

```
delete from study_subjects
  where student_id in (12345, 13627);
```

Rows to be deleted are based on another table conditions:

```
delete from subject_year
  where student_id in (select student_id
                      from student
                      where status = 'A');
```

If the condition is based on multiple columns, the order and correct treatment are inevitable. The previous example is based on only one column – *student_id*. The same result will be reached by using *EXISTS* principle:

```
delete from subject_year
where exists (select 'x'
              from student s
              where status = 'A'
              and subject_year.student_id = s.student_id);
```

However, suppose the condition based on referencing another table composite primary key. In that case, it must be handled as one condition and managed using *EXISTS* set operator:

```
delete from student s
where EXISTS (select 'x'
              from st_field stf
              where field_name = 'Informatics'
              and spec_name is null
              and s.field_id = stf.field_id
              and s.specialization_id=stf.specialization_id)
and student_id not in (select student_id
                      from study_subjects);
```

3.5 The order of operations

Changes performed on data should meet the correct order of operations to reflect consistency. References (foreign keys) should always cover the existing primary key!

3.6 Foreign key definition

The foreign key's value should point to the associated primary key value or *NULL* value. However, it is not inevitable to reflect the *primary key* in general. Also, *unique index* reflection is satisfactory. These rules managing order of operations must be met:

- **Insert statement (+ load)** – associated *primary key* must be *inserted* sooner than the reference to it by a *foreign key*.
- **Delete statement** – *Foreign key* values must be *deleted* sooner than the row with the referenced *primary key*.

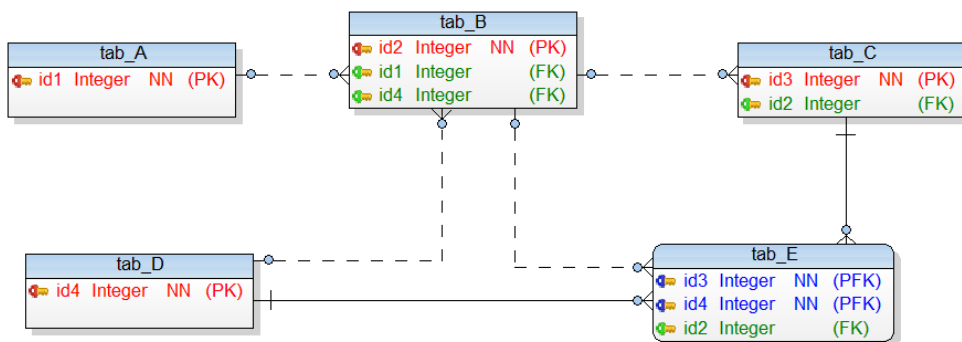


Fig. 3.1: Data model

Tab. 3.1 consists of the provided operation order evaluated by the possibility to do that. Reference model is shown in fig. 3.1.

Tab. 3.1: Operation order possibilities

Order	Insert	Delete
Tab A, Tab B, Tab C, Tab D, Tab E	NO	NO
Tab A, Tab D, Tab B, Tab C, Tab E	YES	NO
Tab E, Tab C, Tab B, Tab D, Tab A	NO	YES
Tab D, Tab A, Tab B, Tab C, Tab E	YES	NO

3.7 Changing the primary key value

Primary key values are usually set based on *sequence* using *triggers*. In that case, the primary key does not have special denotation, and it is not necessary to update it. A typical example can be *student_id* – a simple numeric value. However, it can also have a special meaning. A typical example can be found in the *personal_data* table. The primary key consists of a *personal_id* attribute, which includes *birth date* information. If there is any mistake in the *Insert* statement execution (clerical error), it is necessary to update it later. However, it must be done to emphasize references – foreign key (in table *student*). It is not possible to write a direct *update* statement, whereas the *personal_id* attribute in the *student* would point to a non-existing row in the *personal_data* table, which is not permitted and would cause raising an error.

```
update personal_data
set personal_id = '810701/8079'
where personal_id = '810101/8079';
```

```
SQL Error: ORA-02292: integrity constraint (STUDENT_ENG.SYS_C0010300)
violated - child record found
02292. 00000 - "integrity constraint (%s.%s) violated - child record
found"
*Cause:      attempted to delete a parent key value that had a foreign
dependency.
*Action:     delete dependencies first then parent or disable constraint.
```

Therefore, the natural question is based on the correct order of operations to reflect the necessary change. For the illustration, take the following example.

The aim is to correct the *personal_id* value.

Preliminaries:

1. To change the *primary key* in the *personal_data* table, the *foreign key* in the table *student* must be changed sooner.
2. *Foreign key* – *personal_id* attribute can be changed only to the existing value, not *NULL* mark.

The method for changing the *personal_id* attribute value is, therefore, as follows:

- 1) Create a copy of the personal data with the updated primary key version. Thus, the concerned person will be temporarily stored twice. No problem, it is done inside the transaction (see section 3.8), so it is not visible to any other users / sessions / transactions.

```
insert into personal_data
select '810701/8079', name, surname, street, town, zip, nationality
from personal_data
where personal_id = '810101/8079';
```

Change the reference in the student table to the corrected version of the *personal_id* value.

```
update student
  set personal_id = '810701/8079'
  where personal_id = '810101/8079';
```

Remove the original person from the *personal_data* table.

```
delete from personal_data
  where personal_id = '810101/8079';
```

3.8 Transactions

Each performed database request (query) is encapsulated by the transaction, although you have not perceived it yet. The transaction ensures *atomicity*, *consistency*, *isolation*, and *durability* (ACID). Thanks to that, it is still an easy way to get the original values before approving any change. On the other hand, when any change is approved, it cannot be reversed. Such activity is controlled by the **Transaction Control Language (TCL)**, consisting of these commands:

- **Commit** – approving (confirmation) transaction. All changes are durable without the possibility of getting rid of changes. It is provided by reflecting all changes to the physical *redo log file* by the *Log Writer* background process. If the system's crash occurs, it is possible to reconstruct data using stored log files.
- **Rollback** – getting the changes back.
- **Savepoint** – *Savepoint* command does not end the current transaction, but it creates a point to which the transaction can be reversed. At the end of the transaction, all *Savepoints* are removed automatically.

Notice that the *Exit* command of the *SQL*Plus (SQL Client)* automatically executes *Commit*. Therefore, never shut down the environment tool (console) by clicking on the cross in the right part of the window.

Be aware each DDL (and also DCL) statement automatically generates the Commit command. It cannot be changed. Principles will be described later.

Let's have the following example describing the management of the transaction and consecutive value stored.

```
create table Tab1 (id integer);
insert into Tab1 values(1);
insert into Tab1 values(3);
insert into Tab1 select id+1 from Tab1;
commit;
-- Commit complete.
select * from Tab1 order by 1;
```

What about the values stored? Attribute *id* will hold these values: 1,2,3,4. If you *rollback* the transaction, the same results will be obtained. Why? Because no change has been made since ending the previous transaction. However, let's *insert* one new row (*ID* value = 10). What will happen if you select all data from the table?

```
insert into Tab1 values(10);

select * from Tab1;
```

Naturally, the inserted value will be present. Thus, the output will consist of 1,2,3,4 and 10.

```
ID
-----
1
2
3
4
10
```

What will happen if you rollback the transaction? Value 10 will be removed.

```
rollback;

select * from Tab1;
```

```
ID
-----
1
2
3
4
```

However, notice that there is also *isolation* characteristic of the transaction. Thus, no other transaction will see the values before the successful end of the transaction. Moreover, the transaction manager ensures that no data can be lost after transaction *Commit*.

Such a principle is described in the following example. Let's us assume two sessions of the same user, who has created a previous table (*Tab1*) with defined values (1,2,3,4). From *session 1*, the user adds a new row using the *Insert* statement with value 100. As you can see, Such row is not visible in *session 2* until the transaction approving (*commit*). Similarly, if *session 1* deletes some rows from the database, they will still be visible and available in session 2 until the successful end of the transaction (*commit*).

The order of operations of the following example is essential and reflects the sequence from top to bottom.

Tab. 3.2: Transaction management

Session 1	Session 2
Insert into Tab1 values (100);	
	Select id from tab1; -- 1,2,3,4
Select id from tab1; -- 1,2,3,4, 100	
Commit;	
	Select id from tab1; -- 1,2,3,4, 100
Delete from Tab1 where id=100;	
Select id from tab1; -- 1,2,3,4	
	Select id from tab1;

Session 1	Session 2
	-- 1,2,3,4, 100
Commit;	
	Select id from tab1; -- 1,2,3,4

As already mentioned, always remember that each *DDL* and *DCL* commands generate *Commit* automatically.

```
insert into Tab1 values(20);
create table Tab2 as select * from Tab1;
-- commit is generated automatically
select * from tab1;
```

Thus, now, if you write the *Rollback* command, nothing will happen. Value 20 cannot be removed at all (the transaction has ended successfully).

```
rollback;

select * from Tab1;
```

```
ID
-----
1
2
3
4
20
```

3.9 Practice

3.9.1 Insert statements

1. *Insert* the following data into the particular tables (*personal_data*, *student*, *study_subjects*). Set the values, which are not stated explicitly, to *NULL*, if possible, or propose appropriate values based on integrity constraints.

Personal_id	Name	Surname	Student_id	Class	St_group	Field / specification	Scheduled subjects
875622/2134	Martina	Plush	123	1	5ZI012	Informatics, without specification	BI11 (2013), BI02 (2015), BE01 (2014), BE01 (2015)
890422/8454	Peter	New	90	2	5ZSA21	Information systems, Applied informatics	II08 (2012), II07(2007)
906212/4797	Emily	Smith	23	3	5ZP031	Computer engineering, without specification	BH01(2009), BF08(2009)
885121/3767	Bella	Gloth	8	3	5ZI032	Informatics, without specification	BI11(2006), BH18(2009), ...

Fig. 3.2: Input data

Notes:

- Attribute values *lecturer* and *ects* should be set using *subject_year* table (use the value for the highest school year, if not stored for a particular year).

- *The first two rows* of the table should be inserted using the **Insert-values** statement type.
- For loading the *third* and *fourth* row, use the defined tables **person** (personal and student data) and **subject_pref** (the list of subjects, which should be added to the appropriate person). Download script for these two tables from the USB medium or server (*student_pref_script.sql*), respectively. Execute the file (copy the code to the SQL developer and execute it).

```
insert into personal_data(name, surname, personal_id)
select name, surname, pid from person;
```

- **If you are using local server**, the particular source table can be present in the different schema (owned by the different user). To reference another user table, the fully qualified name should be used – the table name must be prefixed by the name of the owner schema (in the following case, the username is *kvet_eng*). Particular data are available on the USB medium or server storage, respectively.

```
insert into personal_data(name, surname, personal_id)
select name, surname, pid from kvet_eng.person;
```

2. *Insert* the information about the new teacher – *name*: Michael, *surname*: Flower. Set the personal identifier to the maximal assigned value increased by one.

3.9.2 Update statements

1. Change the surname of the person “*Peter New*” to “*Peter Old*”.
2. Change the name of the person with *student_id* = 8 to *Susanne*.
3. Change the assigned subject *BI11* to *BI01* only for the *first-class students*.
4. Change the department to *DI* to all teachers without assigned value.
5. Change the class to the value increased by one to students with status “*S*”, but only those, that are not in the last class (*bachelor* – 3 classes, *engineering study* – 2 classes). Change also the *st_group*. Moreover, use only one statement to perform the requirements.

Structure of the *st_group*:

5	Z	S	A	2	2
Faculty	workplace	field	specialization	class	group_id

Notices:

Get substring:	substr(value, from, [size])	
String concatenation:	'Hello' ' world.'	
Bachelor study:	st_field ∈ <100; 199>	3 classes
Engineering study:	st_field ∈ <200; 299>	2 classes

3.9.3 Delete statements

1. *Delete* the subject *BE01* for a student with *student_id* = 123.
2. *Delete* the subject *BI01* for all students with the *st_group* *5ZI022*.
3. *Delete* all data about *students* whose *registration year* was not later than 2008.

Notes:

- To get the date part units, both solutions are equivalent.


```
select to_char(sysdate, 'DD') as day,  
       to_char(sysdate, 'MM') as month,  
       to_char(sysdate, 'YYYY') as year  
from dual;
```

```
select extract(day from sysdate) as day,  
       extract(month from sysdate) as month,  
       extract(year from sysdate) as year  
from dual;
```


Lab 4 – Data modeling

This lab deals with the data modeling principles. The first part describes system analysis, design and technical design. Creating a proper data model is crucial for consecutive processing, whereas any change requires rewriting code, optimization, etc. Thus, there should be a strong focus on that.

The logical database layer consists of the tables and relationships between them. It can be expressed by the linear notation, occurrence diagram, script, or data model. The set of attributes forms each table, the tuple can be unique identified by the primary key. References between the relationships are made by relationships. The foreign key is the reference value to the particular primary key or unique constraint generally. A special type is made by the recursive (self) relationship referencing the same table. For each relationship, it is necessary to manage the following categorization: type (identifying or non-identifying), cardinality (1:1, 1:N, and M:N creating associative entity) and membership (mandatory or optional characterized by the possibility to hold NULL value for the foreign key).

The data modeling theory is then supervised by the Toad modeler providing the tool for creating and maintaining data model in a graphical format using wizards generating the script for various database system types. Thus, it is not only focused on the DBS Oracle.

4.1 Introduction

If we want to design a complex information system, the process must be done in the defined sequence of steps, which should also be technically supported to get the desired benefit. Nowadays, there are multiple tools used for such a process and are called **Computer Assisted Software Engineering (CASE)**. They support the design process itself as well as offer techniques for appropriate documentation maintenance. Individual CASE tools may differ in detail, depending on the used methodology.

The design is usually done in these three steps – system analysis, system design, and technical design.

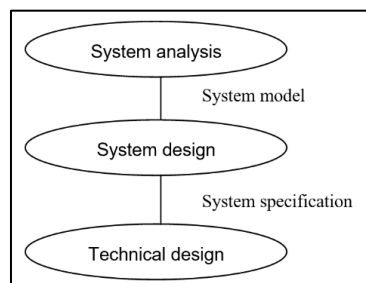


Fig. 4.1: System analysis, System design, Technical design

4.1.1 System analysis

The analysis determines requirements for the system and, on its basis, also a model of the information system is specified. After the specification of the system is implemented, a technical design of the system is carried out, which includes software and hardware requirements for the system.

4.1.2 System design

The design process includes a database design as well as a design of application software and software design to access data stored in the database. Principles are shown in the following figure, which groups data and functional analysis.

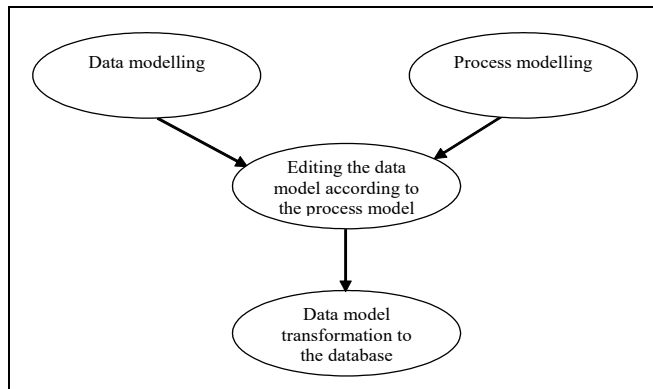


Fig. 4.2: System design

4.1.3 Technical design

The technical design defines data structures and ways to access data depending on the particular application, operating system, etc.

In some methodologies, processes of creating a data model and creating a functional model are separated into two parts, the development of which take place in parallel because they influence each other. The design of the data model is based on user requirements, which are mainly in the form of forms and output assemblies. Part of this process is also the analysis of existing models and related data structures.

Functional modeling often introduces requests for additional data objects definition and handling related to the processing of requests themselves.

The proposed data model is the result of data and functional modeling that interact with each other over the system's entire life cycle.

4.2 Creating data model

As already described in the previous sections, designing a data model concerning application requirements is necessary. From a general point of view, it is first required to identify and solve a conceptual model represented by a conceptual scheme that is subsequently transformed into a logical scheme. The logical diagram generally illustrates a value-oriented data model, in which links between objects are already expressed regarding ensuring data integrity, data model normalization, etc. The implementation of the data model itself is created in the physical design process, which results in the physical (internal) schema of the data model. This schema is deployable and contains a detailed specification of data structures, a way of implementing data types, data organization, and data access methods.

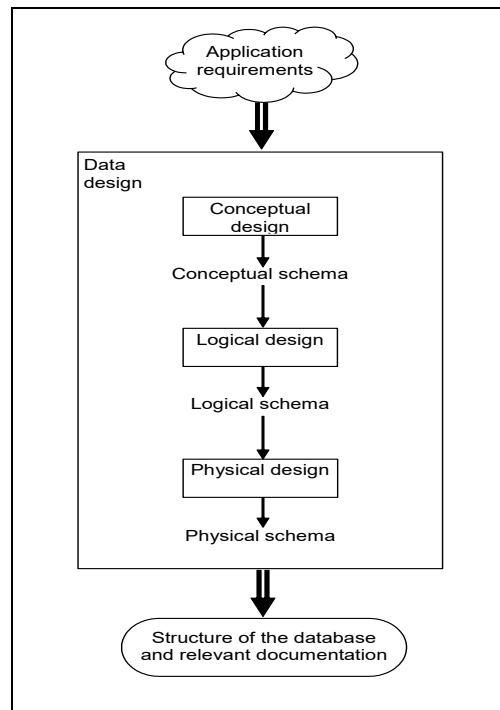


Fig. 4.3: *Creating a data model*

The following figure illustrates the process of creating a data model from the user specification with emphasis on forms and output sets, through the creation of the conceptual data model, the data model itself up to the internal data model. User requirements are defined by forms and output assemblies representing a set of user views on the data. From those, a conceptual data model is created in the conceptual design process, represented by the E-R diagram, in our case. The conceptual data model is mainly transformed through higher-level languages to describe data objects (SQL, C language, COBOL, ...). E.g., in a relational data model, there will be a relational scheme relating a set of tables, references, and integrity constraints between them. The data themselves are stored in database files with a defined organization (index files, index-sequence files, B-trees, inverted files, ...) that are available independently of used the data manipulation commands.

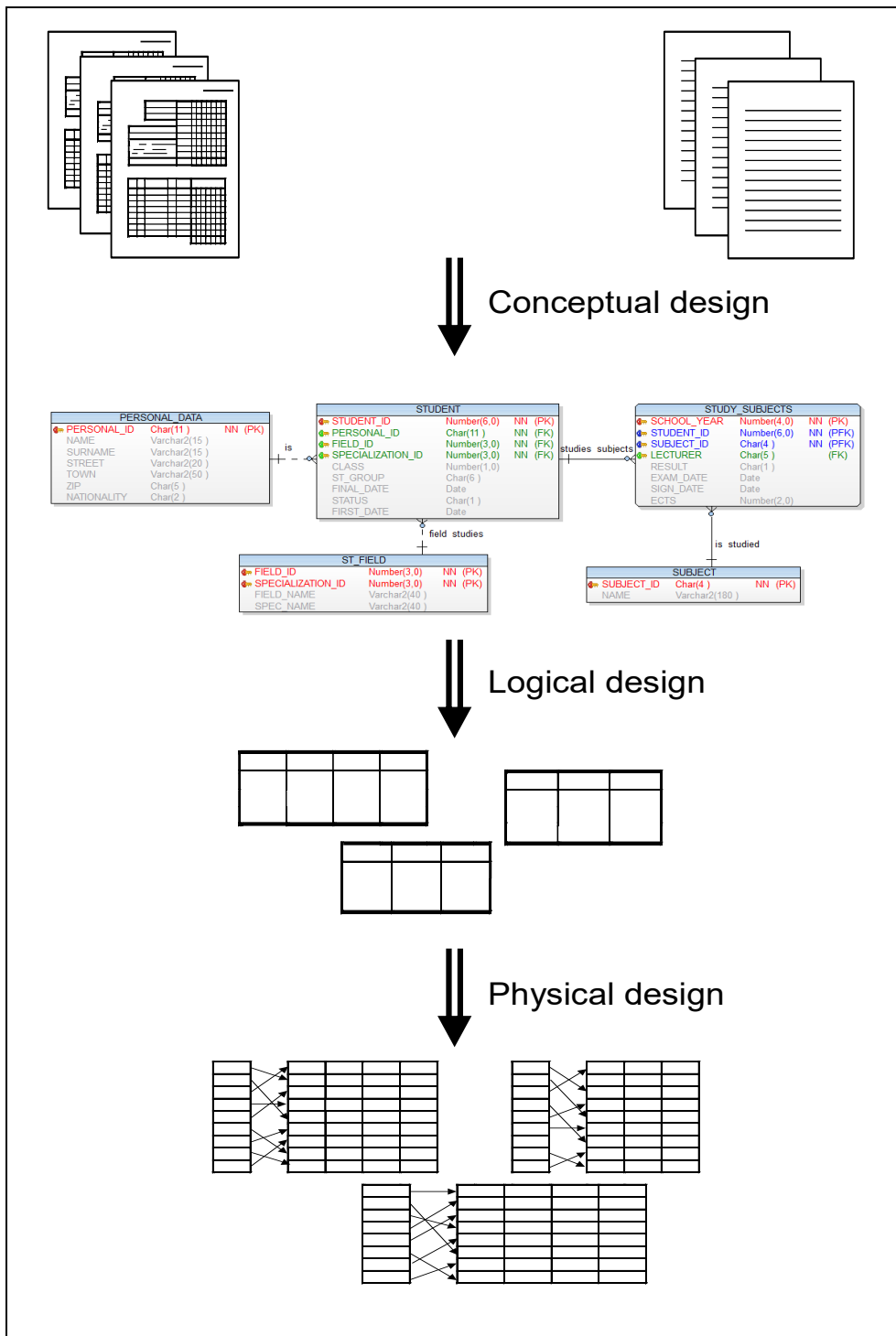


Fig. 4.4: Design

4.3 Conceptual modeling

Transforming information into data and description of the data importance in the database is one of the most comprehensible formalizable components of each DBS. The data themselves, such as "15.5.2017", "1333", "A602", etc., provide no more information value that it reflects the "date", "number", "subject identifier". However, in principle, it is challenging, even impossible, to judge that it is the exam date of the student *Jacob Waxel* from the *Database systems* subject.

Basic knowledge of how to interpret data in a database is stored in a database schema. An example of such a scheme is the following, where *study_subjects* is the schema's name, *school_year*, *student_id*, *subject_id*, etc., are attributes – the name of the data items to be stored in the database.

```
study_subjects(school_year, student_id, subject_id,
               lecturer, result, exam_date, sign_date, ects)
```

STUDY_SUBJECTS		
SCHOOL_YEAR	Number(4,0)	NN (PK)
STUDENT_ID	Number(6,0)	NN (PFK)
SUBJECT_ID	Char(4)	NN (PFK)
LECTURER	Char(5)	(FK)
RESULT	Char(1)	
EXAM_DATE	Date	
SIGN_DATE	Date	
ECTS	Number(2,0)	

Fig. 4.5: *Study_subjects* table

However, the database user will not know whether it reflects a registered subject of the full-time or postgraduate course, whether a person with a given personal number is a student of the first year or another, or whether that subject is compulsory for him or not.

Conceptual models are based on attempts to create a data description in the database – conceptual schema, which is independent of the physical storage of the database. This description should draw the conceptual user view on that part of the real world as closely as possible.

Conceptual models mainly highlight concepts close to *the conceptual point of view*, like *entity*, *object*, *relationship*, *attribute*, *property*, and so on.

Each conceptual model deals with the following issues:

- **Data structure** – From this point of view, it is necessary to identify all objects and their properties, including a description of structures for expressing relationships between objects.
- **Data manipulation** – It is appropriate for each data model and part to design a set of permissible operations over a given data object.
- **Integrity constraints** – For each object, its properties, and relationships between objects, it is necessary to define a set of integrity constraints that limit the basic properties of data objects.

In most cases, the first and third issues are dealt with in detail, data manipulation is resolved at the lower level.

4.4 Entity-relational conceptual model

Definition of the Entity-relational (E-R) conceptual model – The E-R conceptual model (abbreviated E-R model) is a set of concepts and terms that help describe the user's application on the conceptual level of abstraction to specify the structure of the database subsequently.

The E-R model is particularly suited to designing a database schema to access a top-down solution, but it does not mean that it is not possible to create a bottom-up system model.

When designing a system, based on detailed knowledge of modeled reality:

- **Entity types** are identified as sets of objects of the same type.
- **Relationship types** are identified to which entities of the identified types can enter.
- Based on an appropriate level of abstraction, **attributes** to each type of entity and relationship are added, which describe the properties of relationships and entities:
 - *SURNAME* (descriptive type) of the *PERSON* (entity type).
 - *PERSONAL_ID* (descriptive type) of the *PERSON* (entity type).
 - *EXAM_DATE* (descriptive type) when a particular *STUDENT* (entity) passed the exam from the defined *SUBJECT* (entity) and the *RESULT* (relationship description type) itself.
- Multiple **integrity constraints** are identified, expressing the conformation accuracy of the model with reality.

Entities:

STUDENT, *SUBJECT*, *PERSONAL_DATA*, ...

Relationships:

STUDENT (entity) *STUDIES* (relationship) *SUBJECT* (entity), ...

Attributes:

NAME, *SURNAME*, *PERSONAL_ID*, ...

Integrity constraints:

PERSONAL_ID value is the identifier of the *PERSONAL_DATA* table, ...

Definition Entity – Entity (entity) is a real-world object capable of independent existence and can be uniquely differentiated from other objects.

Definition Relationship – A relationship is a connection between two (or more) entities (can be of the same type).

Definition Descriptive type value – Is the value of the descriptive type – we will understand a simple data type – a pair (set of values and set of operations) under the descriptive type.

Definition Attribute – Attribute is a function result associated with the entity or relationship, which expresses the essential property of an entity type, relationship type.

Definitions for the entity, relationship, and attribute are not so strict. There is no unambiguous rule to classify data as an entity or relationship. Often it depends on the analyst's point of view. The indicator may be that in terms of entities, a user often uses nouns while verbs are characteristic to describe relationships.

4.4.1 Identifying key

As already mentioned, each entity should be uniquely identified in the system. A *student* can be identified by *parents*, *personal_id*, *student_id*, etc.

Primary key is a specific set of attributes associated with the table, which uniquely identifies each record in a database table. The *primary key (PK)* must contain *UNIQUE* values and cannot hold *NULL*. A table can have only one primary key, consisting of single or multiple fields (*composite primary key*). It must be also minimal. More about the primary key definition, management, and importance will be described later in chapter [11.3.2 Primary key](#).

4.5 Conceptual schema notation in E-R model

Three types of notations can be distinguished:

- Linear text notation,
- E-R diagram,
- Combination of previously proposed types.

4.5.1 Linear notation

The syntax of the linear notation is expressed in the following schema (fig. 4.6).

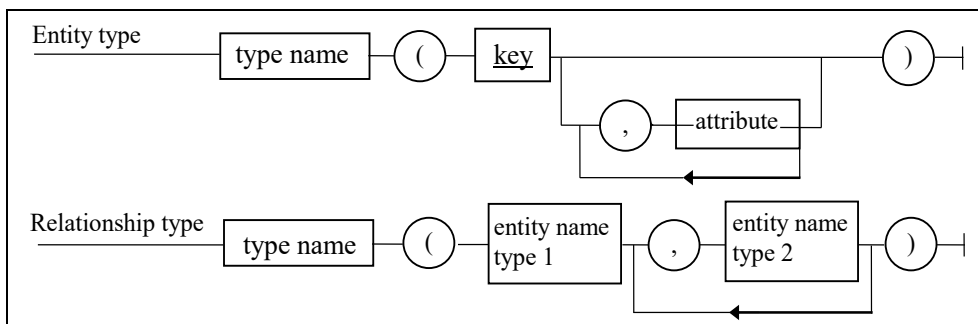


Fig. 4.6: Linear notation

Entity: STUDENT(#STUDENT_ID, CLASS, ST_GROUP, ...)
SUBJECT(#SUBJECT_ID, ...)

Relationship: SUBJECT_REGISTRATION(STUDENT, SUBJECT)

4.6 Type diagram / Occurrence E-R diagram

Type diagram is a more valuable and often used diagram in comparison with *Occurrence E-R diagram*, which shows individual entity and relationship possibilities.

4.6.1 Type diagram

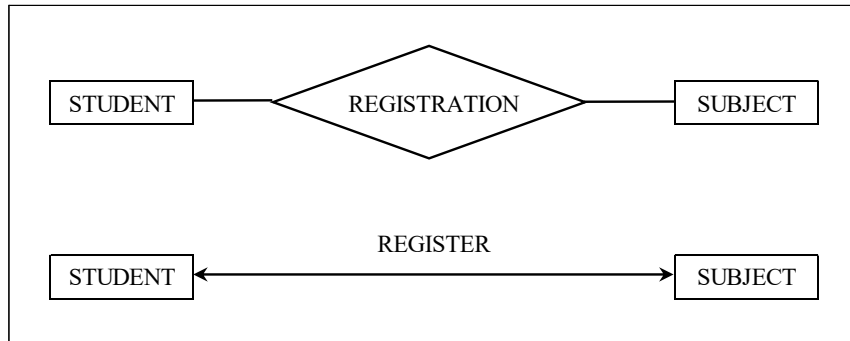


Fig. 4.7: Student – Subject – registration

4.6.2 Occurrence E-R diagram

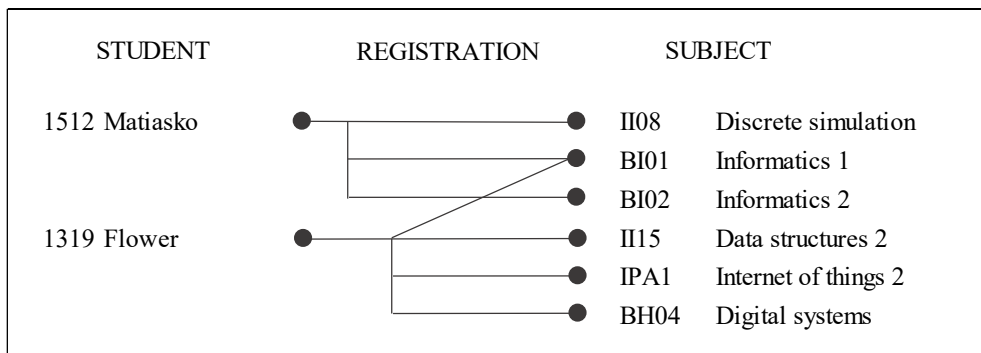


Fig. 4.8: Occurrence diagram

4.7 Attributes

In the previous text, we introduced a basic definition of the attribute and key, which can be considered as a special set of attributes, but the entity and relationship types are described only if each part of descriptive attributes is assigned and described (entity and relationship). Now, let's deal with only atomic (or simple) attributes that give each entity (relationship) no more than one (nonseparable) value.

For each entity type, a separate attribute table should be created. It includes these parts:

- the **name** of the attribute,
- the attribute **type**, in the case of the atomic attribute. It reflects the value set (**domain**) and the set of operations that can be used to the value set. Within this definition, the size of the space (in characters) can be specified, which occupies the outer representation of the attribute value,
- the flag specifying, whether the attribute is **key** (it is part of the identification (primary) key),
- the flag characterizing, whether the particular attribute can hold empty value. It is interpreted as "undefined", "unknown", etc., and modeled using **NULL** value.
- the flag characterizing whether a particular attribute must have a unique value (**UNIQUE, DISTINCT**) or not.

All elements describing the entity type except the attribute name are the integrity constraints defined for the attribute.

CASE modeling tools (such as Toad Modeler, SQL Developer Data Modeler, Erwin, etc.) allow you to display attribute names, defined data types, and possibly some integrity constraints in the graphical view of entities. Consequently, the graphic design might not be complemented by a linear notation. However, for large models, the display of the attributes is not noticeable by the model itself. Therefore, it is often more suitable to display entities such as named rectangles and attribute definitions and their characteristics to delimit by the linear notation or by using special tables provided by the *CASE* tools. Fig. 4.9 shows only primary keys, Fig. 4.10 reflects just the entities.




STUDY_SUBJECTS			
 SCHOOL_YEAR	Number(4,0)	NN (PK)	
 STUDENT_ID	Number(6,0)	NN (PFK)	
 SUBJECT_ID	Char(4)	NN (PFK)	

Fig. 4.9: *Study_subjects table*

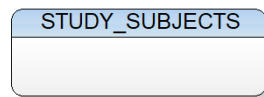


Fig. 4.10: *Study_subjects table*

Entity attribute view in spreadsheet form can be following (Toad Modeler, described in detail later):

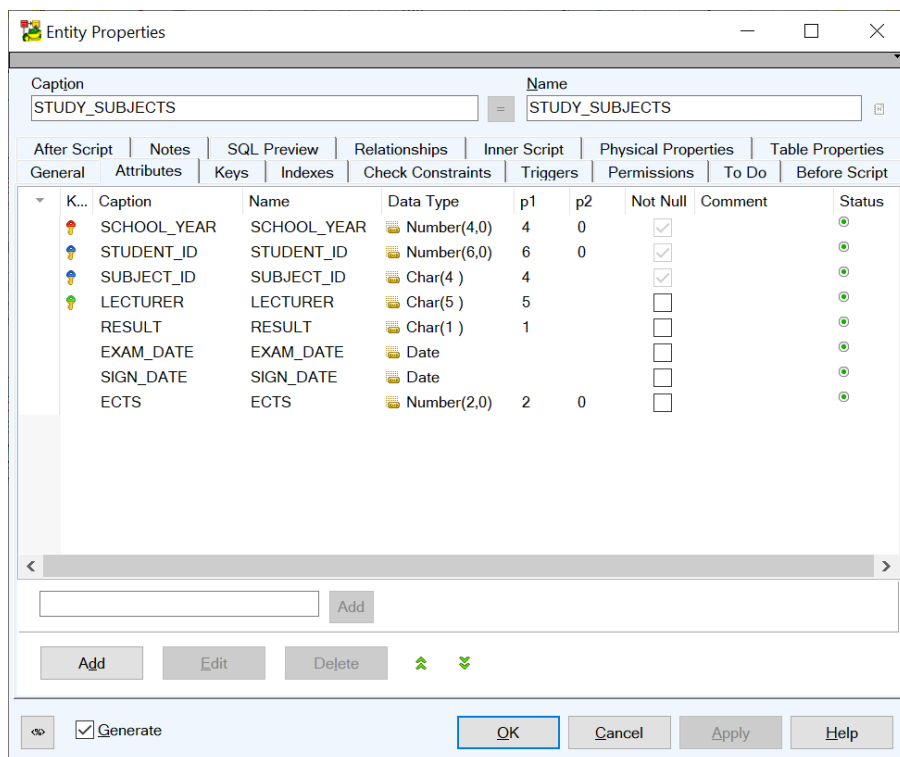


Fig. 4.11: *Entity modeling in Toad Modeler*

4.7.1 Non-atomic attributes

The conceptual model may not be limited to using atomic attributes only. In some cases, it is advisable to create structured attributes (e.g., address, subject identifier, study group number ...). In some cases, it is even reasonable to record several values within one attribute (e.g., authors of the publication). In these situations, it is important how the analyst determines attribute properties, which will be respected throughout the application.

4.7.2 Group attributes

A typical candidate for group attribute is the *ADDRESS*. Generally, this attribute can be split into these parts:

- street name,
- house number,
- the name of the town,
- the name of the country.

Attributes that have such a structure will be called *group* attributes. Their structure does not need to be single-level; attributes can create a hierarchical structure similarly known as a record in programming languages. The *group* attribute value is created by compounding attribute values from several components. A linear description of a group attribute may look like this:

ADDRESS (COUNTRY, TOWN, STREET, NUM)

Group attributes are helpful if we need to access individual components in some cases, but also the whole attribute in other cases. If we always approach only individual elements, it is not effective to associate them with the group.

4.7.3 Multiple value attributes

Another example of using a non-atomic attribute is just *AUTHOR* of the entity type *TITLE*. One title can have several authors, and such limitations cannot be defined in advance. Some conceptual models allow using multivalued attribute definition with variable volume.

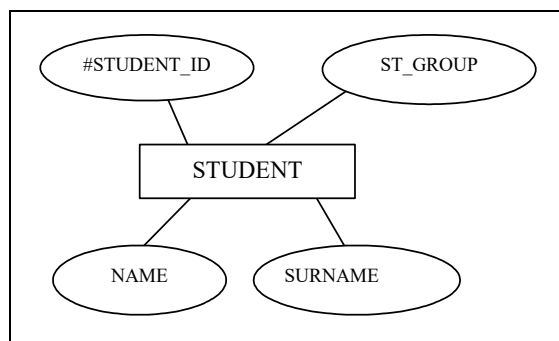


Fig. 4.12: Student model

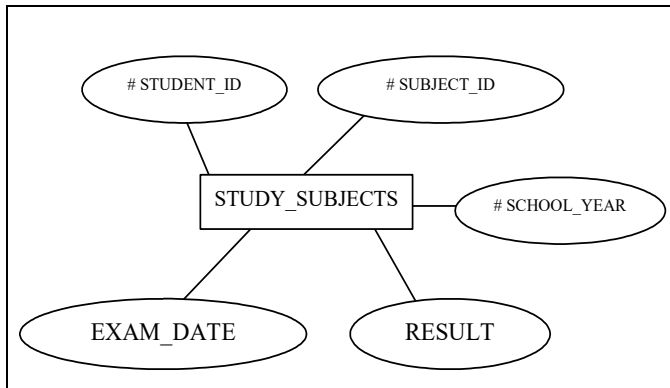


Fig. 4.13: Study_subjects model

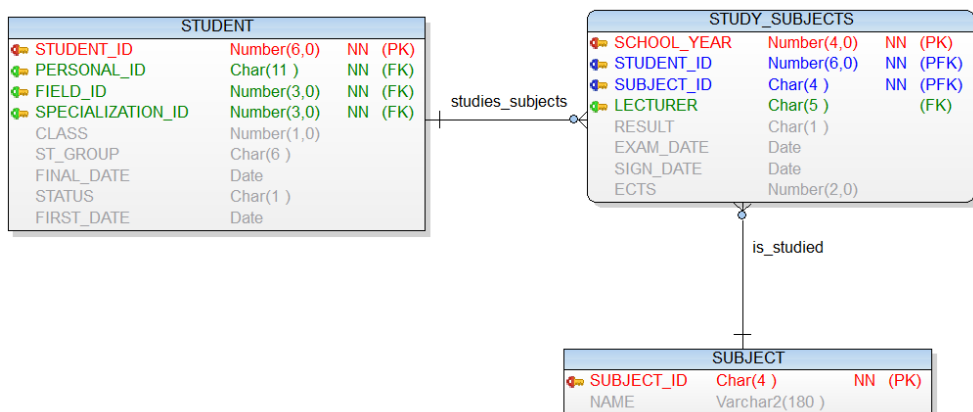


Fig. 4.14: Study_subjects table model

4.8 Relationships and integrity constraints

As stated in the previous section, connections between entities are modeled by the *relationships* that express some kind of integrity constraints – namely – the *cardinality* of relationships as well as the *entities belonging to the relationships*. Important factor is also *relationship type* in terms of *identification / non-identification*.

4.8.1 Identifying and non-identifying relationship

Identifying relationship is a relationship where the key of the *master entity is required for the child entity identification*. The *primary key of such entity is partially (or entirely (fully), if 1:1 cardinality is used) composed from the foreign key and is denoted by the PFK symbol (primary foreign key)*. Thus, once again, a child entity cannot be uniquely identified without a parent. e.g., the driver of the car cannot be identified only by the license plate, whereas several drivers can use the common vehicle.

Identifying relationship is modeled using a *solid line*. The model in the following diagram uses *identifying relationship*.

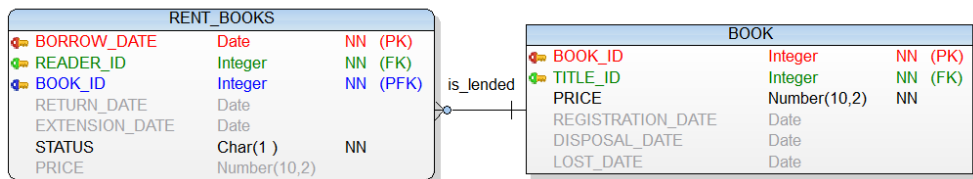


Fig. 4.15: Rent_books, Book model

A non-identifying relationship covers the situations when the primary key attributes of the parent must not become the primary key attributes of the child.

The **non-identifying relationship** is modeled using the *dashed line*. The model in the following diagram uses a **non-identifying relationship**.

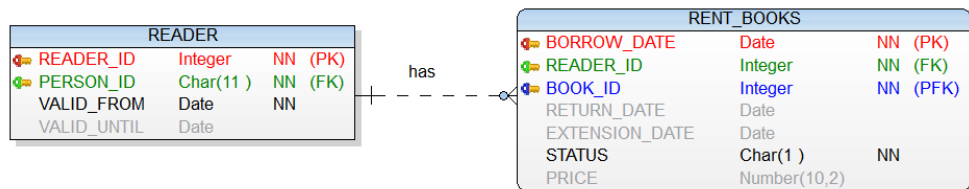


Fig. 4.16: Rent_books, Reader model

Non-identifying relationship can be, generally, enclosed by the integrity rule, specifying, whether the *foreign key* value can hold *undefined (NULL)* value or not. For *identifying relationship*, whereas the *foreign key* is part of the object identification (*primary key*), optionality cannot be applied.

4.8.2 Relationship cardinality

The **cardinality** of a relationship is an integral limitation that expresses the permissible number of entities in a relationship.

Cardinality 1:1

Cardinality 1:1 is an integrity restriction that expresses *the relationship between a maximum of one entity and a maximum of one entity of another*, respectively of the same type, e.g., *teacher* can supervise only one *subject*, the *subject* is supervised by only one *teacher*.

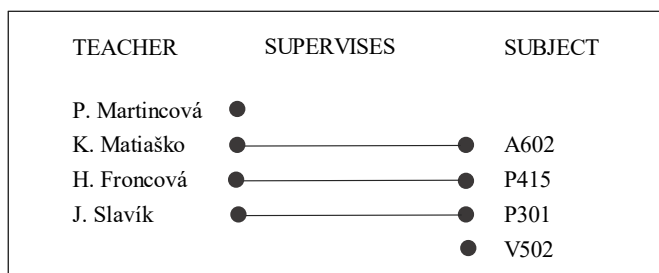


Fig. 4.17: Cardinality 1:1

The relationship in the data model is represented by the value 1.

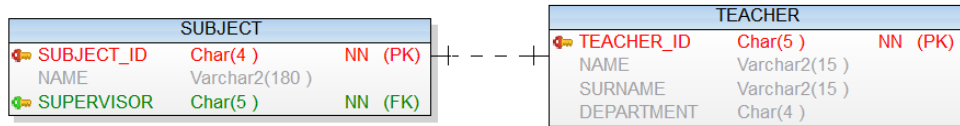


Fig. 4.18: Cardinality 1:1

Cardinality 1:N

Relationship **cardinality 1:N** is an integrity restriction that expresses *the relationship between a maximum of one entity and N entities of another*, respectively, of the same type.

Relationship 1: N corresponds to the following *study rules*:

- a *teacher* can teach more than one *subject*,
- the *subject* is taught by a maximum of one *teacher*.

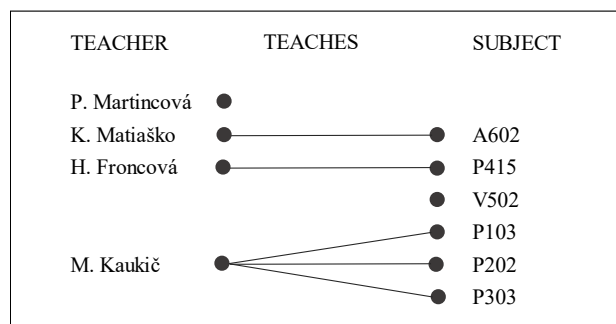


Fig. 4.19: Cardinality 1:N

Notice that relationship type **1:N** generally includes occurrences of **1:0**, **0:1**, and **1:1**, as well. Some of these relationships may be ruled out by stricter rules, e.g.:

- each *teacher* **must learn** more than one *subject*,
- each *subject* **is taught by** only one *teacher*.

The “broom” symbol represents the relationship in the data model.

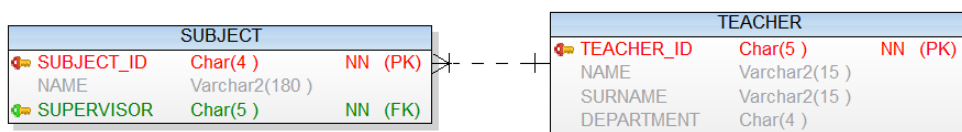


Fig. 4.20: Cardinality 1:N

In relationship **1:N**, the **direction** is significant. In our example, the direction is defined by (one) *teacher* to (many) *subjects*. **1:N** relationship cardinality opposite direction – (one) *subject* to (many) *teachers* would express differently formulated study rules.

Cardinality M:N

Cardinality M:N of the relationship is an integrity constraint that expresses *the relationship between M entities of one type and N entities of another*, respectively of the same type.

The relationship **cardinality M:N** corresponds to the following *study rules*:

- a *teacher* can teach more than one *subject*,
- the *subject* can be taught by more than one *teacher*.

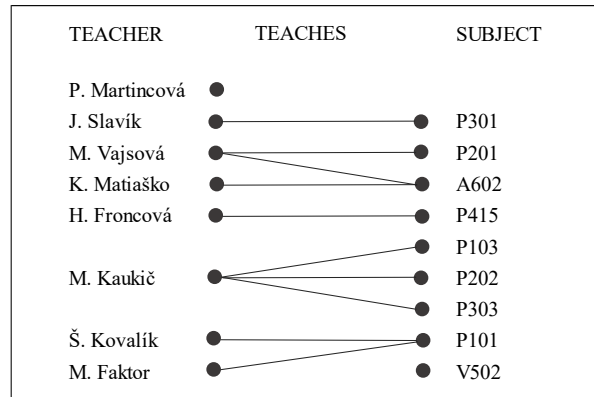


Fig. 4.21: Cardinality $M:N$

Notice that the relationship cardinality $M:N$ also generally includes cases of $1:0$, $0:1$, $1:1$, and $1:N$ (or $N:1$) relationships. The following figures show how we record the cardinality of the relationship to the E-R diagram. For a $1:N$ relationship, it is efficient to name the relationship type. The name represents the direction from the master entity to the slave entity, so in the proposed figures, the relationship name is **TEACHES**, not **IS TAUGHT**.

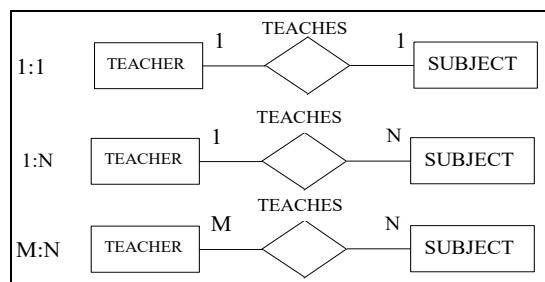


Fig. 4.22: Cardinality $M:N$

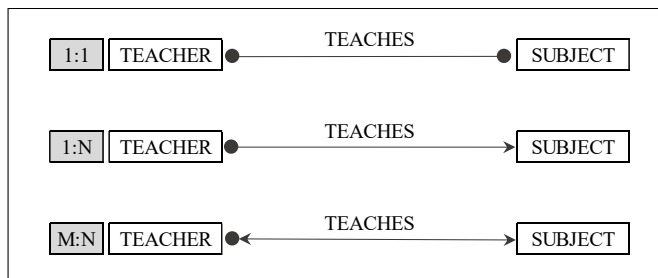


Fig. 4.23: Cardinality $M:N$

The cardinality of the relationship is sometimes expressed by claiming that the entity of one type **uniquely** (does not) determine(s) the entity of the second type, or that the entity of one type is (is not) a **determinant** of an entity of the second type.

4.8.3 Decomposition of the $M:N$ relationship cardinality

We can say that the design of the conceptual scheme is independent of the subsequently used data model. Still, it should be remembered that most database systems cannot express

$M:N$ relationships directly. Other reasons force us to know how to divide relationships $M:N$ into two type $1:N$ relationships.

A common mistake is to assume that decomposition can be done as in the following figure!

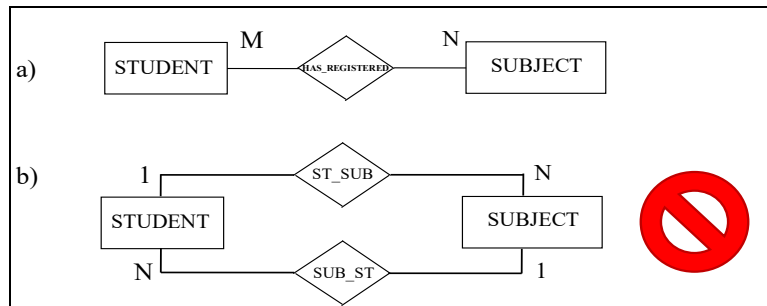


Fig. 4.24: Incorrect modeling

The relationship in the previous diagram (first part) is of the $M:N$ relationship type meaning that there is no functional dependence between the student types of **STUDENT** and **SUBJECT** in either direction. However, the second part of the diagram denotes that the relationships **SUB_ST** imply the functional dependence of **STUDENT** from **SUBJECT** (the instance of the entity type **SUBJECT** is the determinant of the instance of the entity type **STUDENT**), relationship **ST_SUB** implies functional dependence in the opposite direction. Thus, both diagrams clearly show that they express different situations.

To obtain and set correct cardinality, it is advisable to use the occurrence diagram.

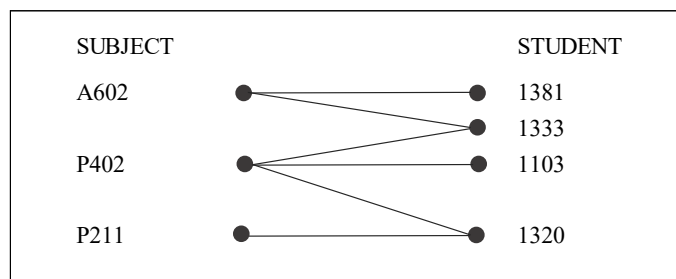


Fig. 4.25: Cardinality $M:N$

It is easy to understand that the relationship **STUDENT – REGISTRATION** is of type $1:N$ and that the relationship **SUBJECT – REGISTRATION** is also type $1:N$. By transforming the *E-R diagram* into an *occurrence diagram*, we obtain the graph shown in the following figure. The defined new entity type (**REGISTRATION**) can be denoted as the intersection entity type.



Fig. 4.26: Cardinality $M:N$

4.8.4 Associative entity

During the process of data modeling, it is often necessary to **decompose** the relationship. It causes the creation of a particular type of entity representing the relationship. If the relationship has $M:N$ cardinality and defined attributes, it is always necessary to create such entity directly in the E-R model.

The following figure shows the relationship with $M:N$ cardinality type.

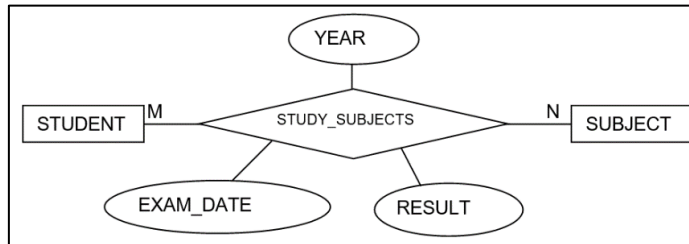


Fig. 4.27: Study_subjects table as an associative entity

And the created model with associative entity looks like this:

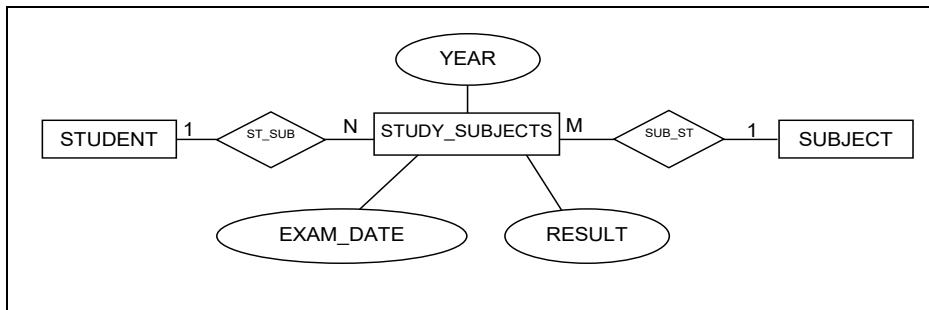


Fig. 4.28: Study_subjects table as an associative entity

Thus, entity *study_subjects* in the model is an **associative entity** between *student* and *subject* entities:

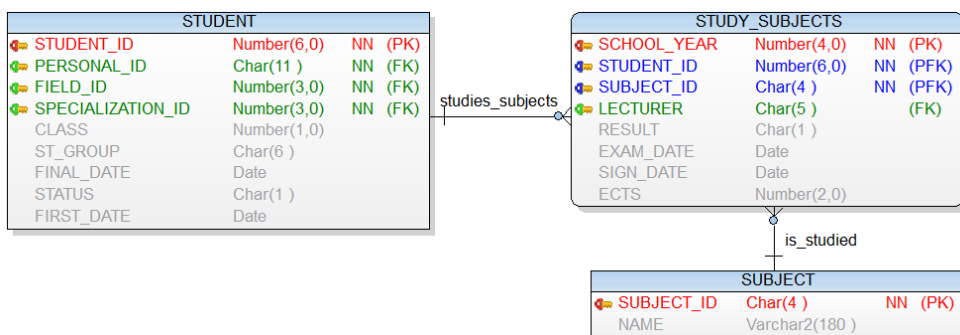


Fig. 4.29: Study_subjects table as an associative entity

In some cases, even despite the cardinality $1:N$, it is preferable to model this relationship using an associative entity. This is especially true in situations where instances of entities would be very extensive in memory space requirements.

A typical example can be in the library sphere – *students may borrow books – and the system deals with only currently rent books with no history.*

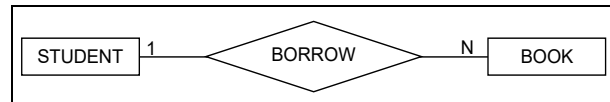


Fig. 4.30: Student, Book table, and associative entity Borrow

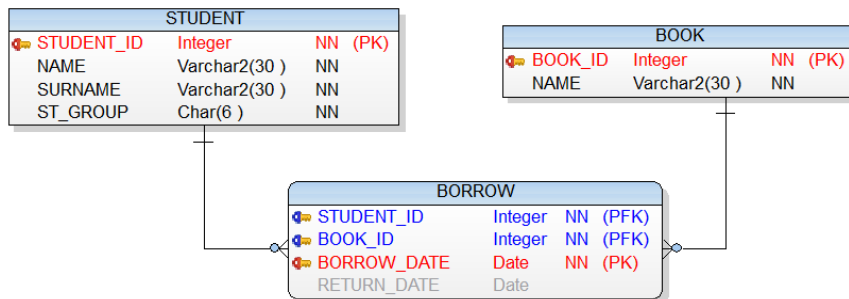


Fig. 4.31: Student, Book table, and associative entity Borrow

4.8.5 Membership types

Membership in a relationship is an integrity constraint that expresses the *necessity of existence*, respectively *possibility of the non-existence of an entity of one type in relationship to the presence of an entity of another type*.

We have shown two different ways in which entities can enter a relationship. Some organizational rules of a modeled reality determine that each occurrence of an entity must be involved in the relationship. Some other cases allow existing entity-type objects outside the relationship. Entity types that are involved in the relationship are named as members of the relationship. Regarding the above-defined concept, we are talking about obligatory and optional membership.

Mandatory membership in a relationship is an integrity restriction that expresses *the need for an entity of one type concerning the existence of an entity of another type*.

Optional membership in a relationship is an integrity constraint that states that *an entity of one type may not exist concerning the existence of an entity of another type*.

Let's have a simple example based on two tables – *teacher* and *department*. The *teacher* is determined by his *teacher_id* and is dedicated to the *department* delimited by its *name*.

In the first example, 1:N relationship cardinality is used with *mandatory* membership types. It means that each *teacher* must be dedicated to the *department*. In other words, the *teacher* cannot be inserted without reference to the *department*.

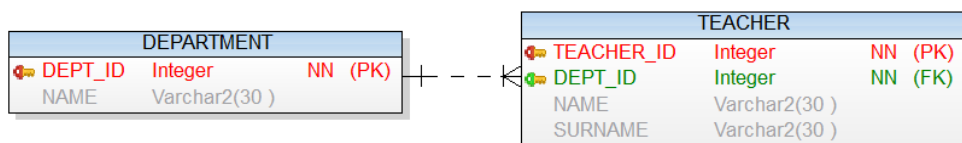


Fig. 4.32: Department, Teacher table

The rule is covered by using a defined relationship, in which the *foreign key* is stated as **NOT NULL**.

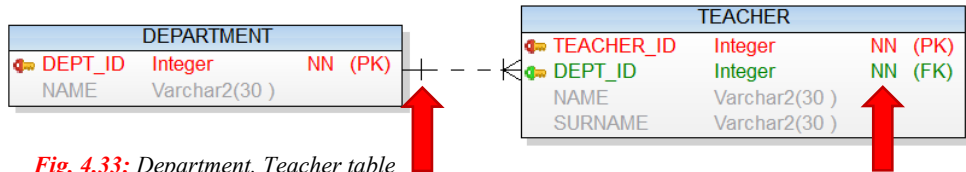


Fig. 4.33: Department, Teacher table

On the other hand, there may be situations where it is appropriate to define a *teacher* without a link to the *department*. In that case, *optional* membership must be defined, which allows putting the *NULL* value as the *foreign key*. Naturally, it can be done only if the relationship is *non-identifying*. In the data model, it is expressed by the *circle* near the department entity (notice that the position of the circle may vary based on the used modeling tool).

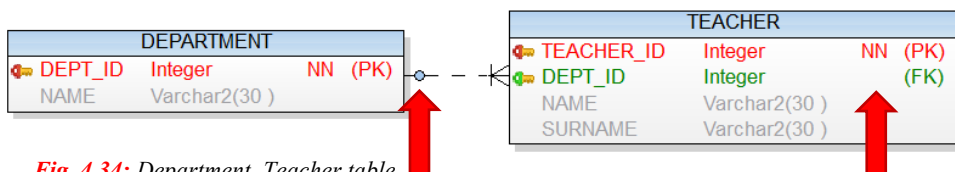


Fig. 4.34: Department, Teacher table

4.8.6 Multiple relationships between same tables

A particular case of relationship management covers the situation that multiple relationships are defined within the same tables. In this case, each relationship covers one connection type. In the following example, the first relationship describes the *student*, the second one *leader (tutor)* of the thesis, and the third defines a reference to the *opponent*. For these purposes, **multiple relationship** term is used. Furthermore, these relationships can have different cardinalities.

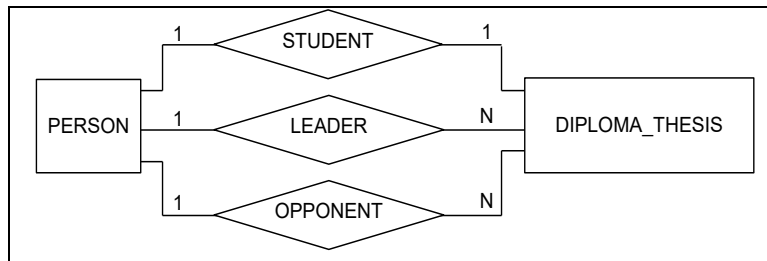


Fig. 4.35: Multiple relationships

Person and *diploma_thesis* tables with regards to relationships can be modeled like this.

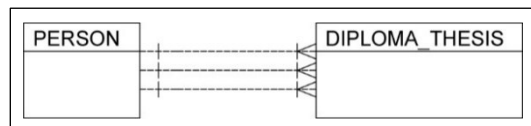


Fig. 4.36: Multiple relationships

Principles of data management and retrieval are described in chapters Lab 2 – Basics of data retrieval and Lab 8 – Advanced techniques of data retrieval.

4.8.7 Recursive (self) relationships

It is often necessary to model the *relationships between entities of the same entity type*. In this case, we refer to the *self-relationship*. This type of modeling is used to provide *hierarchical* relationships between entities, e.g., *employee hierarchy*, *parent-child* relationships, etc.

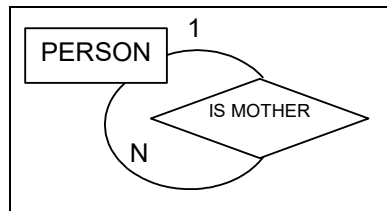


Fig. 4.37: Recursive relationship

An example of the model representation is the following. Notice that the foreign key attribute names must be renamed (*mother_id*, *father_id*).

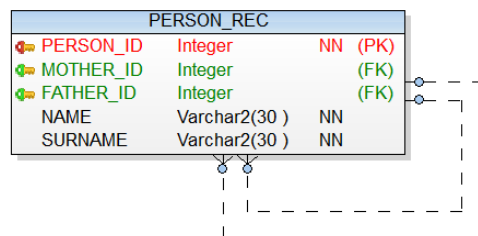


Fig. 4.38: Recursive relationship

4.9 Data modeling in Toad Modeler tool

There are several tools for creating data models. For our purposes, we will use **Toad modeler**, which offers various possibilities for making models and enables accurate changes to data structures across multiple platforms (Oracle, MySQL, MS SQL, DB2, etc.). Furthermore, it allows you to construct *data models* either explicitly or based on the existing system using reverse engineering, compare and synchronize models, quickly generate complex SQL / DDL, create and modify scripts, and reverse and forward engineer both databases and data warehouse systems.

(source: <https://www.toadworld.com/products/downloads?type=Freeware&download=toad-data-modeler>)



Fig. 4.39: QR code to the Toad modeler installation source

4.9.1 Environment settings

The process of the installation is straightforward, and it is not necessary to described it step-by-step. Then, after launching software and attempt to *create a new model, the target database* must be chosen. In our case, we will use the *Oracle database (version 19c)*, but generally, it can generate a script for any database system.

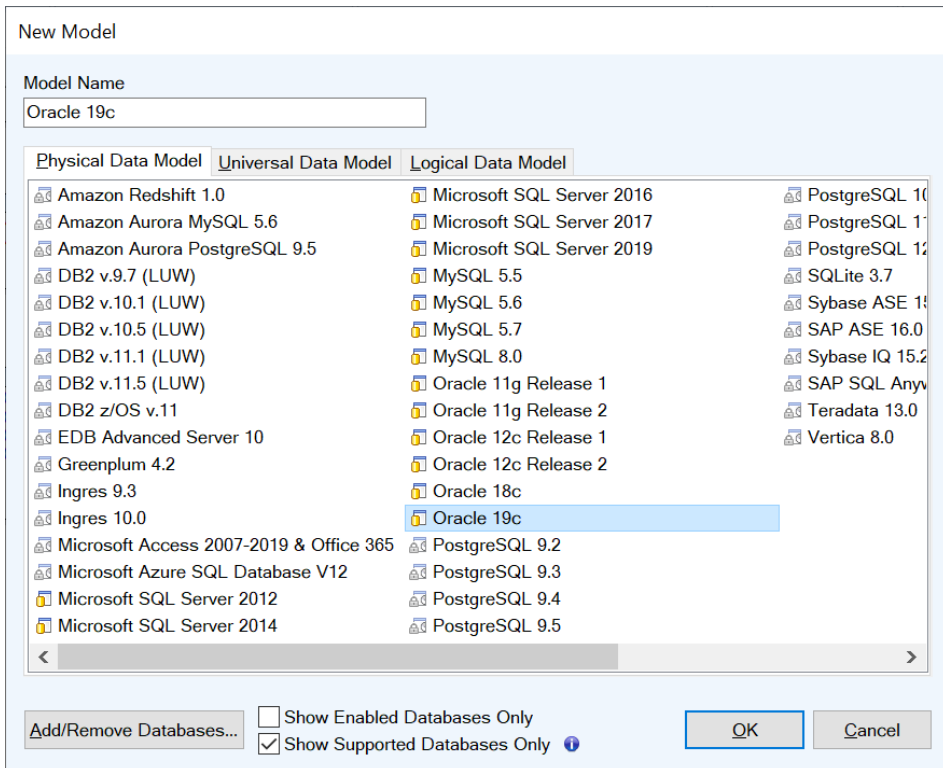


Fig. 4.40: Selecting target database

Then, the *drawing canvas* is created and enabled, which allows you to create the data model.

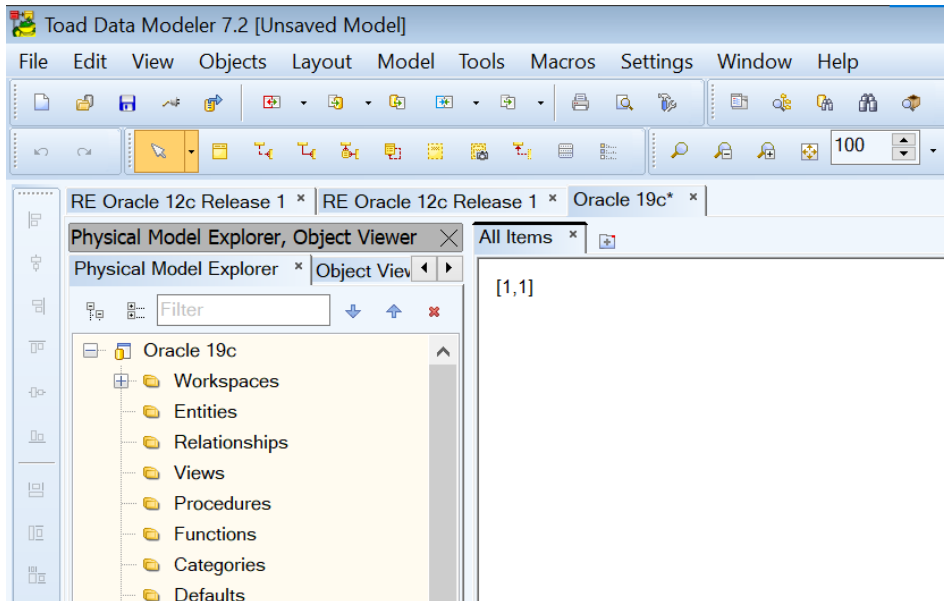


Fig. 4.41: Drawing canvas

The central part of the modeling management is just the *Model Objects* panel:



Fig. 4.42: ERD Objects

- (1) entity
- (2) non-identifying relationship
- (3) identifying relationship
- (4) M:N cardinality relationship (associative entity is created)
- (5) self-relationship

4.9.2 Entity management

After selecting *entity option* (1) and clicking on the *canvas*, *the new entity is created*. Individual *properties* can be changed after double-clicking on it – *attribute definitions with their constraints*. Each *entity* is directly mapped into the *table definition* and must have a *unique name*. For the naming, the first character must be a letter (The Unicode definition of letters includes Latin characters from *a* through *z*, from *A* through *Z*, and letter characters from other languages). Also, underscore (*_*), at sign (*@*), and hash sign (*#*) are allowed. Other characters can be a numeric value or dollar sign (*\$*). No special characters, supplementary characters, and spaces are allowed.

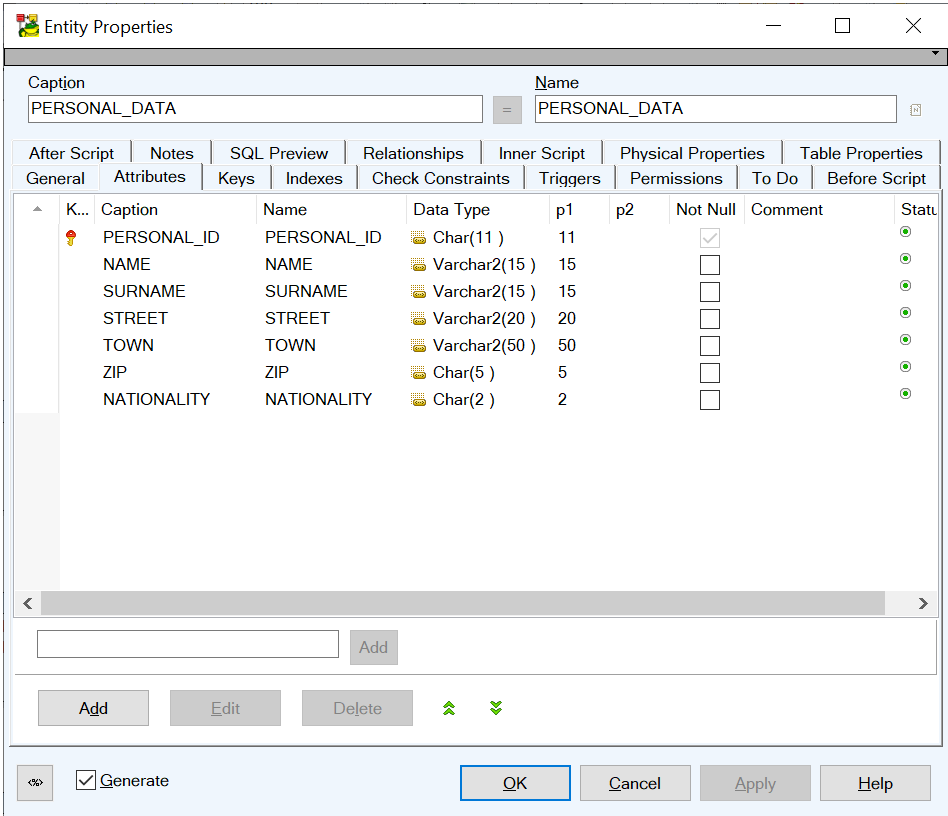


Fig. 4.43: Entity modeling

Each *entity* consists of at *least one attribute*, but generally, it has multiple attributes, also with a unique name (naming convention is the same as table name definition principle). Attributes can be added by clicking on the “**ADD**” button. It is also possible to “**EDIT**” the existing definition or to “**DELETE**” some attributes. Individual attribute definitions are in data grid consisting of these characteristics:

- Key.
- Name / Caption.
- Datatype + size demands.
- NOT NULL flag.
- Unique flag.
- Description (comment).

K...	Caption	Name	Data Type	p1	p2	Not Null	Comment	Status
	PERSONAL_ID	PERSONAL_ID	Char(11)	11		<input checked="" type="checkbox"/>		

Fig. 4.44: Attribute modeling

For the table *study_subjects*, the data grid looks like following:

K...	Caption	Name	Data Type	p1	p2	Not Null	Comment	Status
	SCHOOL_YEAR	SCHOOL_YEAR	Number(4,0)	4	0	<input checked="" type="checkbox"/>		
	STUDENT_ID	STUDENT_ID	Number(6,0)	6	0	<input checked="" type="checkbox"/>		
	SUBJECT_ID	SUBJECT_ID	Char(4)	4		<input checked="" type="checkbox"/>		
	LECTURER	LECTURER	Char(5)	5		<input checked="" type="checkbox"/>		
	RESULT	RESULT	Char(1)	1		<input type="checkbox"/>		
	EXAM_DATE	EXAM_DATE	Date			<input type="checkbox"/>		
	SIGN_DATE	SIGN_DATE	Date			<input type="checkbox"/>		
	ECTS	ECTS	Number(2,0)	2	0	<input type="checkbox"/>		

Fig. 4.45: *Study_subjects* table definition

When managing attributes, the following form will be available. Attribute definition adding or editing can be done using the first tab of the form:

Caption SCHOOL_YEAR		Name SCHOOL_YEAR	
General Check Constraints Foreign Keys Permissions Notes Identity Virtual Column Edition			
Data Type Number(x,y)		Domains 	
Precision 4	Scale 0		
Default Value <input type="text"/>		Default Rule -- None --	
<input checked="" type="checkbox"/> Primary Key <input checked="" type="checkbox"/> Not Null <input type="checkbox"/> Unique (New AK)			
Comment <div style="border: 1px solid #ccc; height: 60px;"></div>			

Fig. 4.46: Attribute definition

Each attribute must have its **unique name** (*ATTRIBUTE NAME* and *CAPTION*). Both mostly hold the same value. However, in some cases, they can differ. The difference of the values is mainly identified if the **foreign key** attribute is renamed or if several attributes would have the same values.

For the script generation, the relevant attribute parameter is just its **name**.

Moreover, each attribute must have an associated **data type** (several data types available with some differences between individual database system types depending on the dialect).

The primary data type categories are:

- string,
- numeric,
- date.

Characteristics are described in chapter [5.2 Data types](#).

For each attribute, three checkboxes are available:

- **Key** (should be selected if the attribute is part of the primary key) – each table must have no more than one primary key, which can also be composite (consists of several attributes).
- **NOT NULL** (should be selected if a no-undefined value can be used).
- **Unique** (should be selected if the particular attribute values must be unique).

☒ Primary Key ☐ Not Null ☐ Unique (New AK)

Fig. 4.47: Attribute definition

Notice that the primary key is always **UNIQUE**, but as the whole set, no individual attributes forming it, thus for *study_subjects* table, trinity {*school_year*, *subject_id*, *student_id*} is **UNIQUE**.

- If {*school_year*} was unique, it would cause that only one *subject* and only one *student* can register in a particular *school_year*.
- If {*subject_id*} was unique, it would cause that only one *student* can register for it, regardless of the *school_year*.
- If {*student_id*} would be unique, it would cause that he can register for only one *subject*, regardless of the *school_year*.
- If the pair {*school_year*, *subject_id*} was unique, it would cause that each *subject* in each *school_year* can be registered by only one *student*.
- If the pair {*student_id*, *subject_id*} was unique, it would cause that each *subject* can be registered by one *student* only once (he cannot repeat the *subject*).
- If the pair {*school_year*, *student_id*} was unique, it would cause that each *student* in each *school_year* can register for only one *subject*.

Thus, the correct solution is the **unique trinity**.

Attribute definition can also be enhanced by **default value** and **check constraint** (column, user integrity, see [Lab 11 – Relational integrity](#)). The **default** value is used if no value for the particular attribute is specified. Notice the difference between **NULL** values, it is not the same in this case. Principles are demonstrated in the following example.

Let's have a simple table **T1** consisting of two attributes – **ID**, **ID2**. Let's have attribute **ID2** enhanced by the **default value**.

```
create table T1(id integer, id2 integer default 1);
```

Then, insert two rows into the table and care about the real data stored in the database. As you can see, generally (across multiple database systems), **default value is used only if no data is used**. If a **NULL** value is explicitly defined, the default value is not used. The reason is that it has been user-specified, although it holds an undefined value.

```
insert into T1 values(3, null);
insert into T1(id) values (2);
```

```
select * from T1;
```

ID	ID2
3	(null)
2	1

In *DBS Oracle*, such an option was valid prior the version 12c. The **default** value would not be applied for the **NULL** value specification. By introducing 12c version, a new clause

has been introduced – *default on null*, extending the default section specification. In that case, also explicitly defined *NULL* values can be replaced by the default option specification.

```
drop table T1;
```

```
create table T1(id integer, id2 integer default ON NULL 1);
```

```
insert into T1 values(3, null);
insert into T1(id) values (2);
```

```
select * from T1;
```

ID	ID2
3	1
2	1

Check constraint reflects *user-defined domain* (data type sub-category) and is explained in chapter [11.7 Domain integrity](#).

4.9.3 User-defined domain

User domain definition borders the value set based on the defined data type. Generally, the *integer value* can also be *negative*. However, for *salary*, it is not suitable to use a *negative value*. Therefore, it is possible to define *own domain* by *limiting values*, which can particular attributes hold.

The domain itself is a set of scalar values of the same data type.

Toad modeler allows the user to define domain by selecting **Model => Model Items => Domains** from the *main menu*.

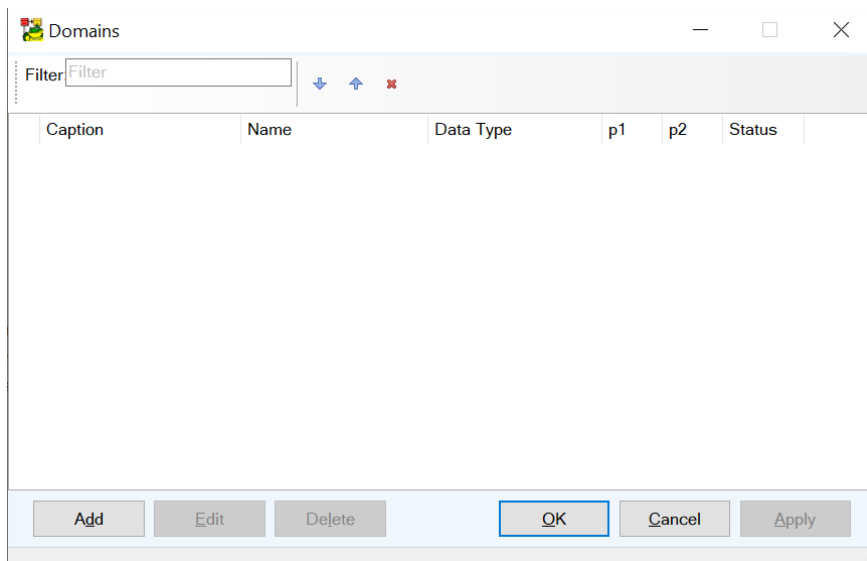


Fig. 4.48: User-defined type

Generally, the *domain* is similar to the core data type, but *check constraint* is defined for the possible value limitation. For demonstration purposes, let's create a new *domain* characterizing **price**. *It can hold any real value, which cannot be negative*. Let's name it "**price_domain**". Click on the add and specify the name and data type.

The image shows a 'Domain Properties - E' dialog box. At the top, there is a dropdown menu showing 'Domain1'. Below this are five tabs: 'General', 'Check Constraints', 'To Do', 'Used in', and 'Notes'. The 'General' tab is selected. It contains several fields: 'Caption' with the value 'price_domain', 'Name' with the value 'price_domain', 'Data Type' with a dropdown menu showing 'Number', 'Default' with an empty text box, 'Default Rule' with a dropdown menu showing '-- None --', and 'Encryption Specification' with an empty text box. At the bottom, there are buttons for '<>', 'OK', 'Ok+Add', 'Cancel', 'Apply', and 'Help'.

Fig. 4.49: Domain definition

The suitable data type is “**number**”. Value set definition is defined in the **Check Constraints** tab. Next, name the constraint (in the **General** tab) and navigate the **SQL** tab.

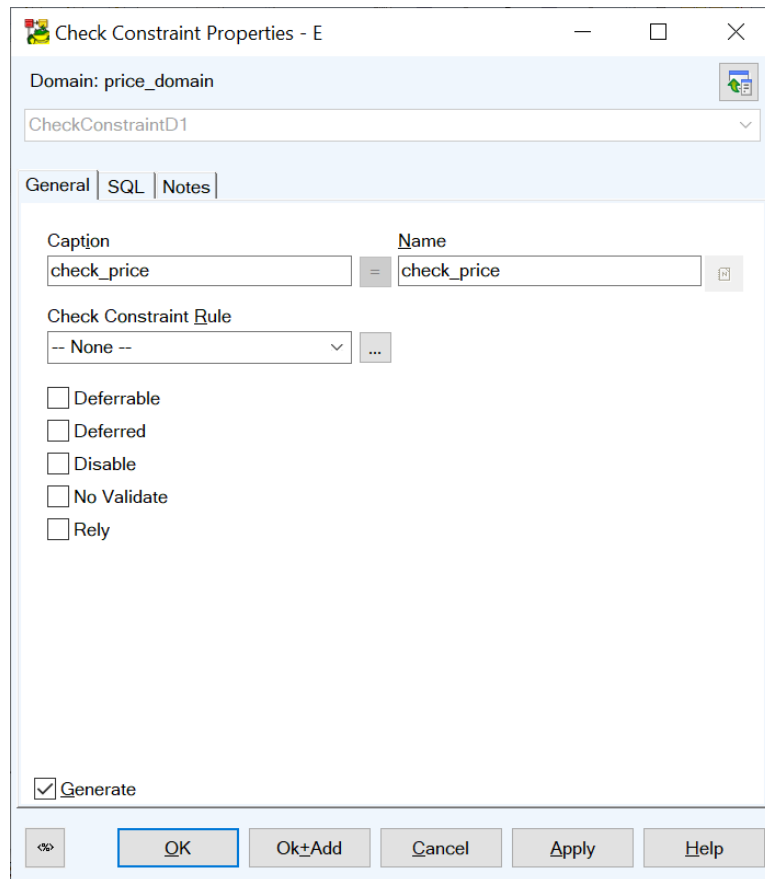


Fig. 4.50: Check constraint – General tab

DDL check constraint for the attribute looks like following:

```
create table Check_tab(price number check (price > 1));
```

Therefore, Toad modeler allows you to use its *internal macro* – **<%ColumnName%>**, which ensures that a particular value is replaced by the appropriate attribute name during the SQL script generation. Therefore, the *Check constraint* definition for the *price_domain* can look like the following. It is written to the *SQL* tab of the Check constraint definition.

```
<%ColumnName%> >= 0
```

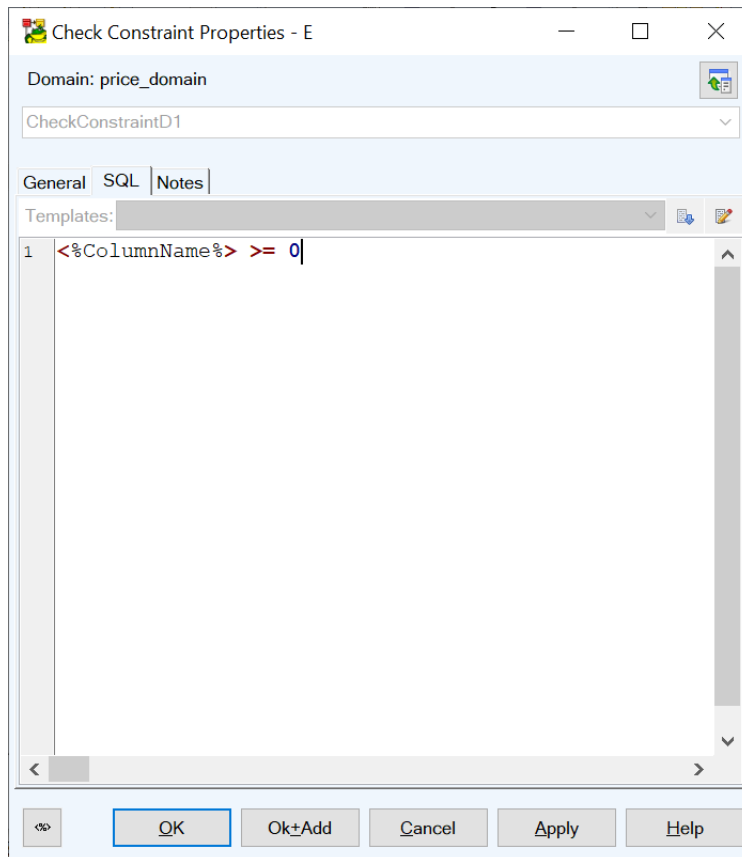


Fig. 4.51: Check constraint – SQL tab

Define the *check constraint* very carefully (no spaces can be used for macro) because the *Toad modeler* does not check syntactical correctness during the SQL script generating process. It can cause significant problems when using such a script in the database server.

Other examples of *check constraints* are the following. In principle, it can use whatever simple condition.

```
<%ColumnName%> in ('T', 't', 'F', 'f') --> boolean data type definition
```

```
<%ColumnName%> between 1 and 1000
```

```
substr(<%ColumnName%>,1,1) = upper(substr(<%ColumnName%>,1,1))
```

Then, the *user-defined domain* can be associated with the *attribute* by the *Domains Select* list.

Fig. 4.52: User-defined type

4.9.4 Relationship management

Relationship definition can be done using the (2), (3), (4), and (5) buttons.

Button (2) reflects *non-identifying* the *relationship*, button (3) delimits *identifying* relationship type, button (4) defines *M:N cardinality relationship*. Button (5) deals with *self-relationships*.



Fig. 4.53: Relationship management

Each *relationship* is **directional** oriented and *routed* from the *parent* table to the *child*. The foreign key is part of the *child* table.

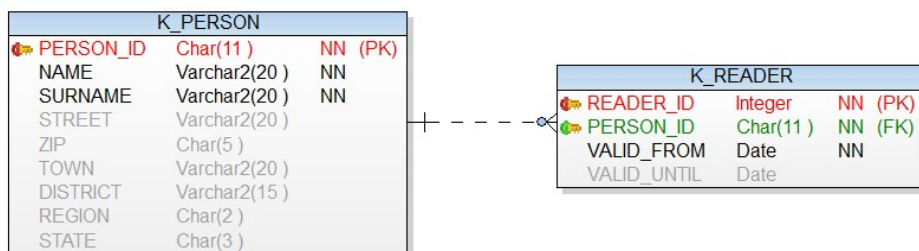


Fig. 4.54: Person, Reader table

Afterward, the *relationship* can be edited by double-clicking on it. The first (*General*) tab of the form is the most important. Each relationship can have a *name*, which will be, transformed into SQL script. A *relationship type* can be edited (from *identifying* to *non-identifying* or vice versa), *membership* (mandatory/optional in either *parent* or *child* entity), and also *cardinality* (1, N, or direct association count limitation). Many other properties can be set, like *referential integrity* management, *deferrable constraints*, etc.

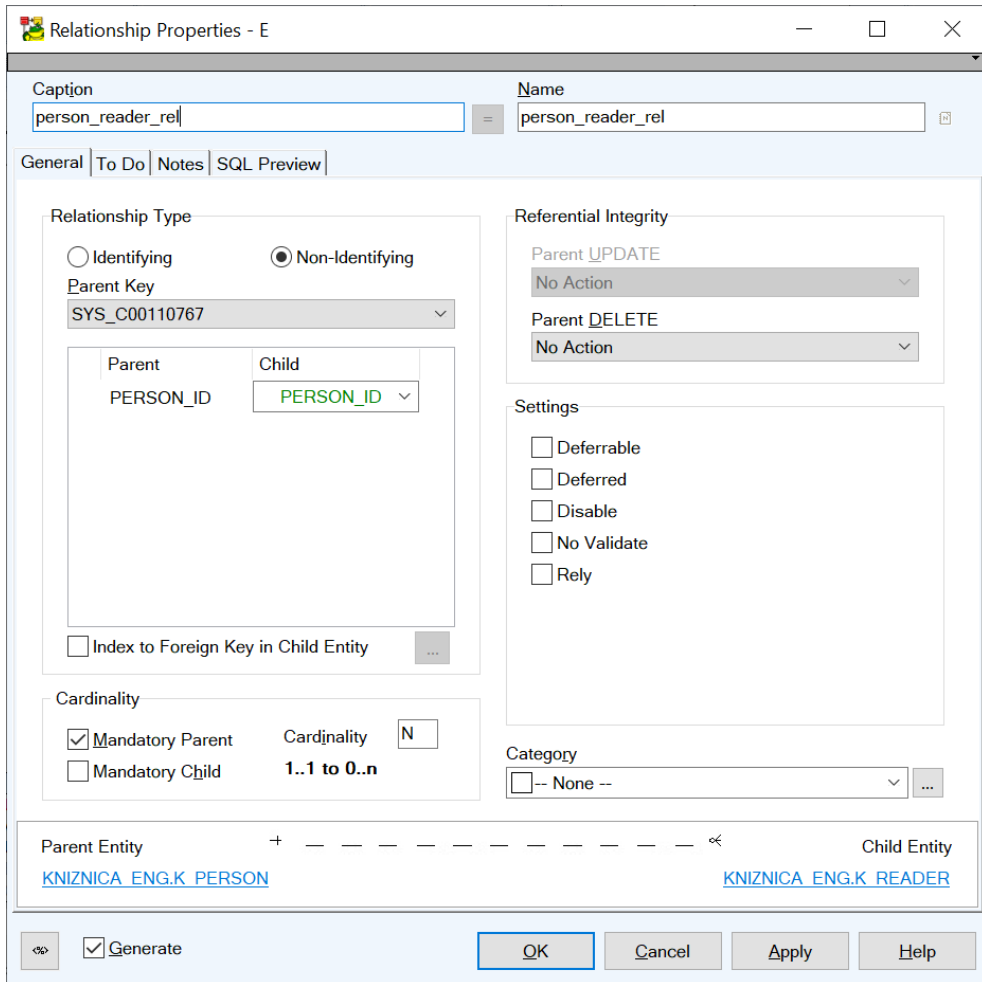


Fig. 4.55: Relationship properties

4.9.5 Generating SQL script

When *data modeling* is finished, *SQL script* can be *generated* and consequently *executed* on the server – *database objects are created*. Whereas the defined script is database system dependent, it is inevitable to choose the correct one. Suppose the different database system type is used compared to the selection at the beginning. In that case, the model can be converted to the particular system by selecting “**Convert Model => Run**” from the “**Model**” main menu tab.


SQL script itself can be generated by clicking on the “**Generate DDL Script**” button  in the “**Model**” panel:



Fig. 4.56: Model panel

In the main menu, it is located in the **Model** menu navigating to **Generate DDL Script => Run** or by using the F9 button shortcut.

The form for the script property definition is shown in the following figure.

DDL Script Generation of RE Oracle 12c Release 1

What to Generate | Detail Settings | Referential Integrity | Select List

Location of SQL File
C:\Users\Michal Kvet\Documents\Toad Data Modeler\GeneratedScripts\Generated.SQL

User / Schema
-- Not Specified --

Split Output File ☐ Append To File ☐

Property Name	Extended Value
<input type="checkbox"/> Model	
<input type="checkbox"/> After Script	
<input type="checkbox"/> Before Script	
<input type="checkbox"/> Directories	Create
<input type="checkbox"/> Editions	Create
<input checked="" type="checkbox"/> Entities	Create
<input type="checkbox"/> Functions	Create
<input type="checkbox"/> Java	Create
<input type="checkbox"/> Materialized Views	Create
<input type="checkbox"/> Packages	Create
<input type="checkbox"/> Permissions to Objects	
<input type="checkbox"/> Procedures	Create
<input checked="" type="checkbox"/> Relationships	Create
<input type="checkbox"/> Sequences	Create
<input type="checkbox"/> Synonyms	Create
<input type="checkbox"/> User Data Types	Create
<input type="checkbox"/> User Groups	
<input type="checkbox"/> Users	Create
<input type="checkbox"/> Views	Create
<input type="checkbox"/> Zone Maps	Create

☐ Show Preview

Save Action... Load Action Verify Show Log Generate Show Code Help

Fig. 4.57: Generating script

First, there are several options defined by checkboxes bordering objects for which script should be generated. For our purposes, we will use:

- **Entities** (DDL script for table definitions, primary keys, etc.).
- **Relationships** (creating relationships between tables).

The proposed tool can also generate many more script types, like user-defined *indexes*, *procedures*, *functions*, or *views*. Also, *referential integrity constraints* (*cascade*, *nullified*, *restrict*) can be defined, and management ensured by the *triggers*, generated automatically based on user selection (see [Lab 10 – Triggers](#)). The script can be generated for the whole model or for its subpart, which can be defined in the “**Select list**” tab:

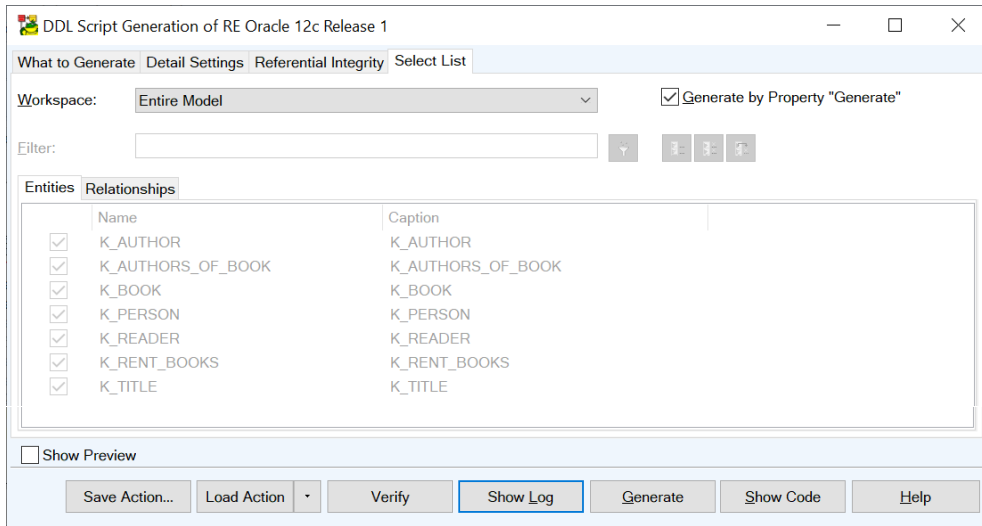


Fig. 4.58: Generating script

The important script option is in the “**Detail Settings**” tab. *Deselect* the option “**Use Quotation Marks**”.

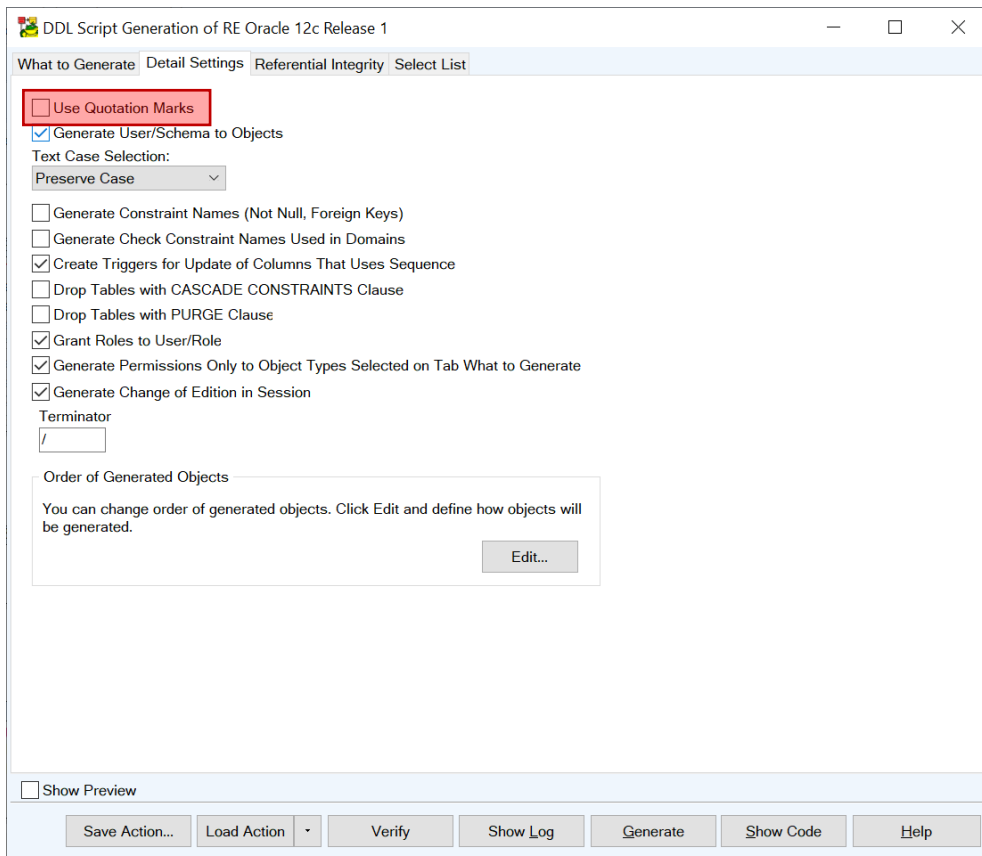


Fig. 4.59: Use Quotation Marks problem

Otherwise, the *generated script* would use **quotation marks** – each *object* and *attribute name* would be enclosed *by quotation marks* (which are not visible by querying *data dictionary views*, see [Lab 14 – Data dictionary views](#)), however, they should be used when coding scripts. Moreover, particular names would be *case-sensitive*. Generated code would look like the following:

```
Create table "Title"
(
    "title_id" Integer NOT NULL,
    "title_name" Varchar2 (50) NOT NULL,
    "genre" Varchar2 (8) NOT NULL,
    "publisher" Varchar2 (40),
    "year_of_issue" Integer,
    "isbn" Char (13),
    primary key ("title_id")
)
/

Alter table "Reader" add foreign key ("person_id")
                                references Person ("person_id")
/
```

In the **Detail Settings** tab, several parameters can be set, like *Cascade* operation in case of table dropping, *Purging* tables (after object dropping, it is not placed in the recycle bin), etc.

Default Terminator is “/”. Therefore, after the table, index, or relationship definition, slash is placed in a separate line like the terminator. In the preceding parts, we mainly used semicolons. However, the principles are the same.

After the definition and options specification, an SQL script can be generated. *Be strictly aware and execute the proposed script on the server (cloud) only if it has been generated without errors! If not, correct them and repeat the process.*

Generated script for the table *Reader* looks like the following – *table* and *primary key* are defined. Afterward, the *relationship* is added.

```
Create table Reader
(
    reader_id Integer NOT NULL,
    person_id Char (10) NOT NULL,
    valid_from Date NOT NULL,
    valid_until Date,
    primary key (reader_id)
)
/

Alter table Reader add foreign key (person_id)
                                references Person (person_id)
/
```

4.9.6 Executing script on the server

The script can be executed on the server based on your preferred software tools.

If the *SQL Developer* is used, the particular file is opened on the client site and can be executed on the server (the same principle as running whatever code). Either *desktop* or *web* version of the *SQL Developer* can be used.

Script stored in a file can be executed via the already described *SQL Client*, as well. In that case, a particular file should be located by pointing to the server. Set and locate the directory where the file resides (using the *cd* command). Finally, execute the script using *SQL*Plus* environment by using the *start* command (assuming that the file name is *script_library.sql*):

```
start script_library.sql
```

Seven tables should be created – *Author*, *Authors_of_book*, *Title*, *Book*, *Rent_books*, *Reader*, *Person*.

4.9.7 Working with directories and files

It is helpful to know some basic commands for dealing with directories and files in the file system and traverse using the tree structure. Note that these commands are associated with the *operating system*. Thus, if you want to call them from the *SQL*Plus (SQL Client)* environment, then they must be prefixed by the *host* command:

```
SQL> host pwd
```

```
$ pwd
```

Tab. 4.1: Commands for working with directories and files

Command	Explanation
<i>\$pwd</i>	Getting actual working directory <i>/home/kvetl</i>
<i>\$ls</i>	Listing the directories and files inside the actual directory
<i>\$ls -la</i>	Listing the directories and files inside the actual directory with more details (like access privileges, owner, ...)
<i>\$ls directory</i>	Listing the directories and files defined inside the directory parameter
<i>\$cd</i>	Moving to home directory <i>/home/kvetl</i>
<i>\$cd ..</i>	Moving to parent (direct superior) directory
<i>\$cd directory</i>	Moving to the defined directory (relative path)
<i>\$cd /path/directory</i>	Moving to the defined directory (absolute path) – starts with a slash (/)

Command	Explanation																		
<p><i>\$chmod value name</i></p> <p>User (owner) Group Others RWX / RWX / RWX R – read, W – write, X – execute $R = 2^2$ $W = 2^1$ $X = 2^0$</p> <table border="1"> <thead> <tr> <th>#</th><th>rwX</th></tr> </thead> <tbody> <tr><td>7</td><td>rwX</td></tr> <tr><td>6</td><td>rw-</td></tr> <tr><td>5</td><td>r-X</td></tr> <tr><td>4</td><td>r--</td></tr> <tr><td>3</td><td>-wX</td></tr> <tr><td>2</td><td>-w-</td></tr> <tr><td>1</td><td>--X</td></tr> <tr><td>0</td><td>----</td></tr> </tbody> </table> <p><i>\$chmod {u g o a} {+ = -} {r w x}</i> u – user, g – group, o – others, a – all</p>	#	rwX	7	rwX	6	rw-	5	r-X	4	r--	3	-wX	2	-w-	1	--X	0	----	<p>Changing access privilege of the defined directory/file (name) using parameter values (value)</p> <p><i>chmod 751 file.txt</i> owner => rwX privileges group => r-x privileges others => --x privileges</p> <p>Changing access privileges using access string</p> <p><i>chmod g+w file.txt</i> write privilege is added to the group</p>
#	rwX																		
7	rwX																		
6	rw-																		
5	r-X																		
4	r--																		
3	-wX																		
2	-w-																		
1	--X																		
0	----																		
<i>\$mkdir directory name</i>	Creating directory (make directory)																		
<i>\$rmdir directory_name</i>	Deleting directory (remove directory) – it must be empty																		
<i>\$cp from to</i>	Creating file copy, parameters <i>from</i> , and <i>to</i> are used to define the file's location. Using this command, it is also possible to rename the file.																		
<i>\$mv from to</i>	Moving the file, parameters <i>from</i> and <i>to</i> define the location of the file. Using this command, it is also possible to rename the file.																		
<i>\$rm file name</i>	Removing files from the file system.																		
<i>\$rm -r directory_name</i>	Removing the directory with all files and directories inside (be aware of using such a command).																		

New files can be created using any provided editor. During this lab, the “*joe*” editor will be used. However, feel free to use any you like.

To create a new file, use the following command. Whereas SQL code will be obviously written into the files, meet the concept of using “*.sql*” file extension.

```
$ joe file_name.sql
```

There are also multiple *joe* editor shortcuts, which can effectively improve data management. Some of them are in the following table:

Tab. 4.2: *Joe editor shortcuts*

Shortcut	Meaning
CTRL + K + B	The first (begin) point of the block definition
CTRL + K + K	The last (end) point of the block definition
CTRL + K + C	Copying defined block
CTRL + K + M	Moving defined block
CTRL + Y	Removing the whole row
CTRL + K + X	Saving and exit
CTRL + C	Exit without saving
CTRL + K + D	Saving file only
CTRL + Z	Previous word
CTRL + X	Following word
CTRL + A	The beginning of the line
CTRL + E	End of the line
CTRL + U	Previous screen, like <i>PgUp</i>
CTRL + V	Following screen, like <i>PgDn</i>
CTRL + K + U	Beginning of the file
CTRL + K + V	End of the file

4.10 Practice

1. Download preprepared *model* of the *library* from the USB medium, respectively server (*flight_part.dm2*).
2. Extend the model by adding *Rent_books* and *Book* table with appropriate relationships. Mind the correct direction, relationship type, cardinality, and membership.

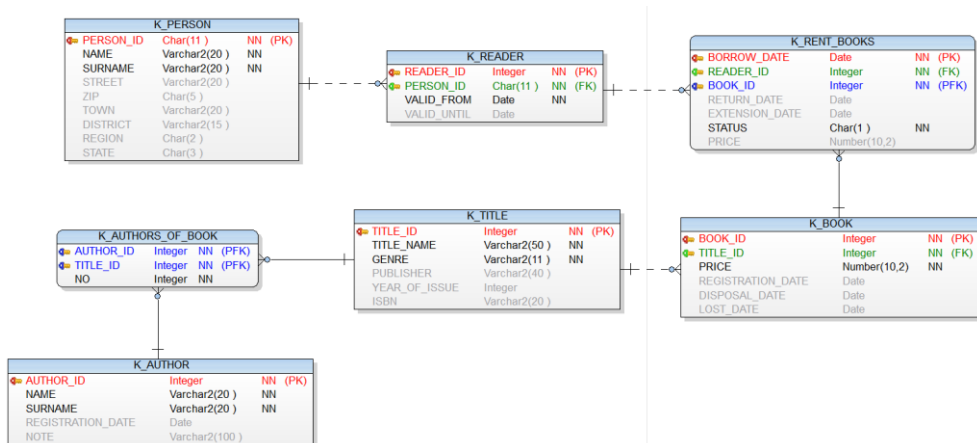


Fig. 4.60: *Library model*

3. Set the suitable *data types*, *NULL/NOT NULL flags*, *primary keys*, etc., for all attributes.
4. Which *data type* have you selected for attribute *price*? Is it possible to put there *negative value*?

5. Define the domain *price_domain*, which limits the value set of the domain (use macro `<%ColumnName%>`):

```
<%ColumnName%>  >= 0
```

6. Associate defined domain with all price data attributes.
7. Ensure that each *publication* can have no more than 6 *authors*. Thus, the order is delimited by the values 1, 2, 3, 4, 5, and 6. Define and associate user domain.
8. Ensure that the value of the *publisher* attribute will always hold uppercase values (defined explicitly by the user). Define and associate user domain.
9. Extend the table person, so you will also record the parents for the child (the book can be returned either by the person who borrowed that book or by the parent). Do not forget to rename foreign key attributes (*mother*, *father*). Use optional membership types.

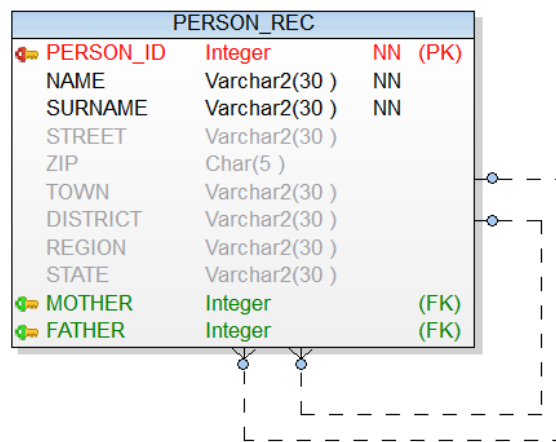


Fig. 4.61: Person, mother, father modeling

10. Record also an identifier of the *editor* and *illustrator* for each *title*. If the *title* has no images, a particular *illustrator* attribute value can hold an *undefined value*. Assume that each *title* has no more than one *editor* and *illustrator*. Reference the table *Author*.

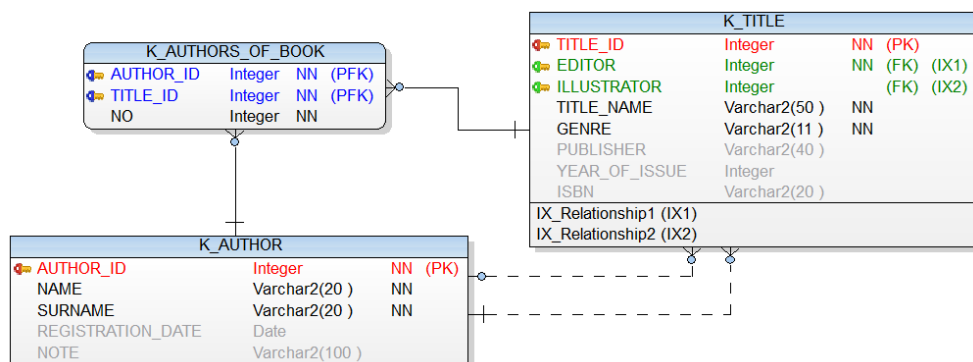


Fig. 4.62: Illustrator, editor modeling

11. *Generate SQL script for tables, primary keys, and referential integrity. **If no errors occurred, execute the script on the server** (otherwise, correct them and repeat the process).*
12. *Drop defined tables in the correct order.*

```
drop table table_name;
```


Lab 5 – Create, Alter and Drop commands

This lab deals with the Data Definition Language (DDL) formed by the Create, Alter and Drop commands. Compared to the DML statements, DDL changes the database structure, not the data themselves.

It offers the extended data type summary, user, and table management. Reader will learn the basics of the data retrieval process performance and indexes. Section 5.5 deals with the index types (B+tree, bitmap, hash), access methods, and addresses to the physical database – ROWID pointers.

5.1 Introduction

This lab will introduce and describe principles of database object definition, management, modifications, and remove operations on the object level definition. All commands are covered by the *Data Definition Language* (DDL) statements – *Create*, *Alter*, *Drop* and *Truncate*. Notice that by using these statements, object management is provided, not the data stored in those structures (e.g., table definition, not the data management inside the table). **Create** command is used for adding (creating) new database object (*table*, *index*, *sequence*, *view*, *procedure*, *function*, *package*, *trigger*, *user*, ...). The **Alter** command aims to modify the database object, **Drop** command removes the database object from the system. **Truncate** operation removes the pointers to the data blocks holding the data, resulting in removing all data rows from the particular table object. Before going deeper to individual operations, let's introduce and summarize available data types.

Moreover, such commands are usually managed internally by developed software tools. Database system Oracle consists of a small number of instances created by *Create Database* command, but mainly by *Database Assistant (DBCA)* tool. Each instance is delimited by its name – *SID* (do you remember it from the installation process as well as connecting to the database, don't you?). These instances are independent, and each of them consists of user accounts. For DBS Oracle, each user has an assigned *schema* (1:1 assignment) for storing particular objects, like *tables*, *views*, etc. These objects can also be accessible to other users if privileges are granted (privilege management is described in [Lab 7 – Managing privileges](#)). A complex description of the administration processes can be found in [1] [5] [13].

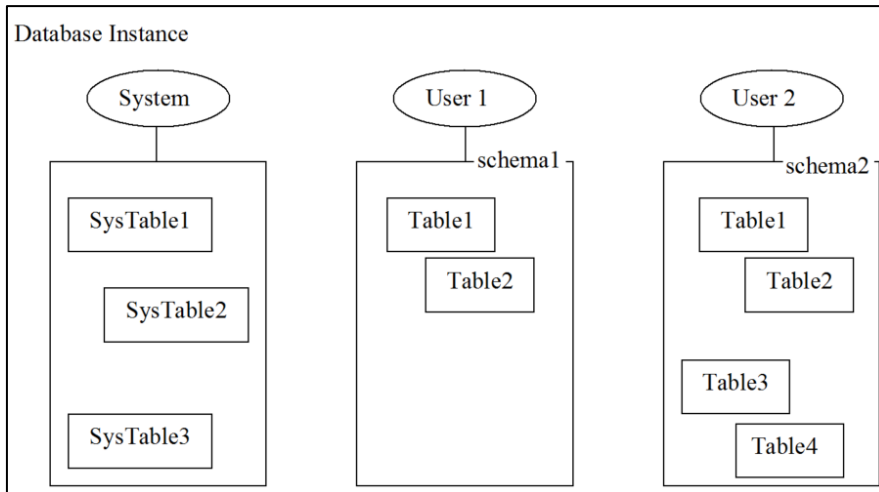


Fig. 5.1: Database instance model and user accounts

5.2 Data types

SQL supports multiple ranges of data types. The following table reflects the possibilities implemented in DBS Oracle in comparison with SQL norm:

Tab. 5.1: Data types

			<i>SQL norm</i>	<i>Oracle</i>
<i>Scalar data types</i>	<i>Strings</i>	<i>Fixed size</i>	CHARACTER (n)	CHAR(n)
			NATIONAL CHARACTER (n)	NCHAR(n)
		<i>Variable size</i>	CHARACTER VARYING (n)	VARCHAR(n), VARCHAR2(n)
	<i>Numeric values</i>	<i>Integer values</i>	SMALLINT	NUMBER(5)
			INTEGER	NUMBER(10)
				NUMBER(38)
		<i>Fixed decimal size</i>	DECIMAL(m, n)	NUMBER(m, n)
			NUMERIC(m, n)	X
		<i>Float decimal size</i>	SMALLFLOAT	FLOAT(63)
			FLOAT	FLOAT(126)
			DOUBLE PRECISION	NUMBER
	<i>Data and Time</i>		DATE	DATE
			TIME	X
			TIMESTAMP	TIMESTAMP
			INTERVAL	INTERVAL
			INTERVAL	
	<i>File</i>		CLOB	CLOB
			NCLOB	NCLOB
				LONG
			BLOB(n)	BLOB
				LONG RAW

Note, that there are two data types dealing with variable string definition – *varchar* and *varchar2*. Such situation occurs for historical reasons, whereas the original definition

(*varchar*) was replaced by the optimized version (*varchar2*). Currently, any format, you use, the optimized version is always used.

5.3 User management

User management covers the particular category of database objects as an interface between the database and user activities. *Users* are commonly managed (*created, altered, dropped*) by the database administrator. However, now, you are the supervisor of the whole cloud database instance, so you are responsible for user management, as well. For defining new users (schema), it is necessary to **Create user**. Therefore, we describe the principles more precisely in this section.

Each user is delimited by the **username** (*login*) and **password** (which can be managed locally or by external verification methods, like *LDAP*). Moreover, each of them must have assigned space for storing defined objects (*tables, views, procedures, etc.*). This space is called **tablespace**, and two types are distinguished – **default tablespace** (for storing persistent data objects) and **temporary tablespace** (space, where temporary tables, intermediate data, *Select* statements results, etc. resides. After processing, these objects are purged and space freed). Moreover, each user can have an assigned **profile** and **quota** for system resources. There are also another two keywords, which are suitable to be described. **Password expire** keywords ensures that created user will be forced to change his password immediately after his first successful login to the database. The **Account lock** keyword is used if you want to create a new user, however, such user will not be possible to login using it because such account is locked. It means that all defined objects still reside in the system, but it is not possible to access the system using such a user (locked user can also be caused by performing a suspicious activity, like too many incorrect login attempts or too old passwords without change). *Please notice that password of the user should start with a letter, no numeric value. Moreover, always define a strong password consisting of characters (lower and uppercase), numeric values, and special characters.*

The syntax of the create user command looks like the following. Only the first two rows of the script are necessary. The rest have their default values, which will be used, if not explicitly defined.

```
CREATE USER user_name
IDENTIFIED { BY password | EXTERNALLY | GLOBALLY AS 'CN=user' }
[ DEFAULT TABLESPACE tablespace ]
[ TEMPORARY TABLESPACE tablespace ]
[ QUOTA { number [K|M] | UNLIMITED } ON tablespace ]
[, QUOTA { number [K|M] | UNLIMITED } ON tablespace ]
[ PROFILE profile_name ]
[ PASSWORD EXPIRE ]
[ { ACCOUNT LOCK | ACCOUNT UNLOCK } ]
```

Connect as the *admin* user and *create a new user*. The solution can look like this. The created username will be *mk_user* and password *my_password*.

```
create user mk_user identified by my_password;
```

In this case, the rest values will use their default values. The following query can be used to get the default values for the tablespace definition (*data dictionary view* is used, principles are defined in [Lab 14 – Data dictionary views](#)). For now, use it as it is.

```
select *
from database_properties
where property_name like 'DEFAULT%TABLESPACE';
```

	PROPERTY_NAME	PROPERTY_VALUE	DESCRIPTION
1	DEFAULT_TEMP_TABLESPACE	TEMP	Name of default temporary tablespace
2	DEFAULT_PERMANENT_TABLESPACE	SYSTEM	Name of default permanent tablespace

All these characteristics can be later changed (using *Alter* command).

The command itself consists of the *Alter* keyword followed by the object type (in this case, “user” will be used) and object name (e.g., *kvet_eng*). Then, characteristics to be changed are defined. So, if you would like to change associated permanent tablespace (new value will be “system” tablespace), the script can look like this:

```
alter user kvet_eng default tablespace system;
```

If you want to freeze or unfreeze a user account, the following *Alter* command can be used:

```
alter user kvet_eng account lock;
```

```
alter user kvet_eng account unlock;
```

The particular category covers the principle of changing the user password. Although it can also be done using *Alter* command, it is not very suitable because the non-encrypted form of the password is visible on the screen (now, the password will be “new_password”).

```
alter user kvet_eng identified by new_password;
```

The password of the user can be changed anytime in two ways, in principle. The first solution covers the technique of changing user password when a particular user is logged on. In this case, he uses the *password* command. First, an existing password will be required, followed by a new password to be set. You will be prompted to write it twice for security reasons (avoiding typos).

```
password
```

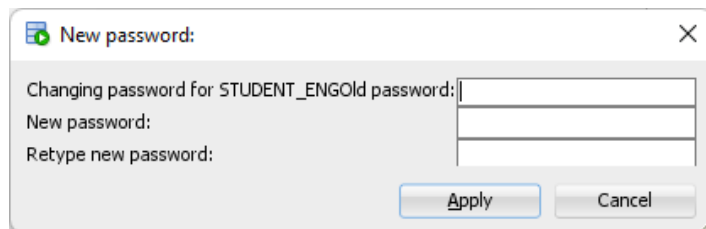


Fig. 5.2: Changing password

The second solution allows the *administrator* to change any user password. Naturally, this operation can also be done without knowing the actual password of the particular user. Mentioned command *password* is extended by the username definition (*system* user is changing the password of the *kvet_eng* user):

```
password kvet_eng
```

To remove the particular user from the system, the ***Drop user*** command should be used. If a particular user has some defined objects, the operation will fail. It is caused by security reasons, where there is no reverse operation for such activity.

```
drop user kvet_eng;
```

```
SQL Error: ORA-01922: " CASCADE must be specified to drop 'KVET_ENG'"
*Cause:      Cascade is required to remove this user from the system.
             The user own's object which will need to be dropped.
*Action:     Specify cascade.
```

A ***Cascade*** keyword must be used to force the system to drop user regardless of the defined objects. However, always think twice. Such an operation cannot be easily reversed.

```
drop user kvet_eng cascade;
```

5.4 Table management

The user's main activity is the data management and the object definition, in which the data will be stored. In the following chapters, we will describe the principles of data management and data modeling techniques. Now, we will deal with the database object definition itself. The following table shows the object types, which can be managed in the DBS Oracle.

Tab. 5.2: Object types

	<i>Oracle type</i>
<i>System objects</i>	DATABASE
	USER
	SCHEMA
	ROLE
	PROFILE
<i>Basic objects</i>	TABLE
	UNIQUE INDEX
	INDEX
	SYNONYM
	SEQUENCE
<i>Derived objects</i>	VIEW
	MATERIALIZED VIEW
	SNAPSHOT
<i>Automatic action management</i>	TRIGGER
<i>Stored methods</i>	PROCEDURE
	FUNCTION
	PACKAGE

For now, we will describe the principles of table definition; other object management will be described later (in a particular chapter defining such objects).

The syntax of the creating table consists of several parts and looks like this. It can be considered complicated at first sight. However, we will describe each keyword principle separately using multiple examples.

5.4.1 Create command

```
CREATE TABLE [schema_name.]table_name
[
    ( { column_name datatype [DEFAULT expr] { [column_constraint] } [...]
      |
      table_constraint
    } [...]
    )
]

column_constraint ::=
[CONSTRAINT constraint_name]
{
    [NOT] NULL
    |
    { UNIQUE | PRIMARY KEY }
    |
    REFERENCES [schema_name.]table_name [ ( column_name ) ]
    [ ON DELETE CASCADE ]
    |
    CHECK (condition)
}

table_constraint ::=
[CONSTRAINT constraint_name]
{
    { UNIQUE | PRIMARY KEY } ( { column_name } [, ...] )
    |
    FOREIGN KEY ( { column_name } [, ...] ) REFERENCES
        [schema_name.]table_name
        [
            ( { column_name } [, ...] )
        ] [ ON DELETE CASCADE ]
    |
    CHECK (condition)
}
```

As you can see, each table must be defined by its unique name. By default, the table is created in the logged-in user's schema but can also be defined in another schema. *Naturally, particular privileges to access another schema and create a new object must be granted.* In that case, the name of the table would be extended by the *schema* (username) or a particular *user* (owner of the object).

```
Create table kvet_eng.person ...
```

Then, individual attributes are listed with their names, data types, and constraints. So, now, let's create a simple table (*person*) consisting of three attributes (*personal_id*, *name*, *surname*). *Each table must have at least one attribute.*

```
Create table person
(
    personal_id    char(11),
    name           varchar2(15),
    surname        varchar2(15)
);
```

Naturally, some attributes cannot be *NULL*, so if the table definition requires *personal_id* value as *NOT NULL*, the solution will look like the following. Notice that table must be dropped before changing its definition by the new *Create* command. Changing the structure and constraints of an existing table can be provided using *Alter* command described a bit later.

```
Create table person
(  personal_id  char(11)          NOT NULL,
   name        varchar2(15),
   surname     varchar2(15)
);
```

As we can see, by default, each attribute is listed as *NULL*. As evident, we can get the table schema using already known command *desc*.

Name	Null	Type
PERSONAL_ID	NOT NULL	CHAR(11)
NAME		VARCHAR2(15)
SURNAME		VARCHAR2(15)

The primary key is a significant part of each table, allowing the user to access the particular row of the table directly. If the primary key is simple (consists of only one attribute), two possibilities are available for the definition. The first principle is based on using *primary key* keyword after the particular attribute definition (*column constraint*):

```
Create table person
(  personal_id char(11) primary key,
   name varchar2(15),
   surname varchar2(15)
);
```

The second one uses the primary key definition after the attribute listing (*table constraint*):

```
Create table person
(  personal_id char(11),
   name varchar2(15),
   surname varchar2(15),
   primary key(personal_id)
);
```

Notice that the primary key is automatically *NOT NULL* from the definition. It is not necessary to define it explicitly.

Only the second solution is available in the case of composite primary key definition (multiple attributes covering primary key).

To demonstrate the solutions and principles, let's create another table *Employee* consisting of information about the person's employment contract in the particular company.

How would you define the structure of the table? Which attributes are necessary? What about the primary key definition? The natural solution defines composite primary key:

```
Create table employee
(  personal_id char(11),
    employer_id integer,
    date_from date,
    date_to date,
    primary key(personal_id, employer_id, date_from)
);
```

Notice that composite primary key definition directly in the **Create table** command can be done only with the previously defined principles. It is not possible to write a primary key keyword after multiple attributes forming column constraint primary key because it would be evaluated as an attempt to create multiple primary keys for one table resulting in exception raising.

```
Create table employee
(  personal_id char(11) primary key,
    employer_id integer primary key,
    date_from date primary key,
    date_to date
);
```

```
Error report -
ORA-02260: table can have only one primary key
02260. 00000 - "table can have only one primary key"
*Cause:      Self-evident.
*Action:     Remove the extra primary key.
```

Foreign key

These two tables (*person*, *employee*) can be linked together, forming a *relationship*. Foreign key references the primary key of the second table (to be honest, it can also reference any unique index). To form the relationship, it can be done by using the **references** keyword either in **Create** or **Alter** command. Notice that the attribute names must be enclosed in the parentheses. Moreover, this command only adds the reference. The particular attribute must already be part of the table.

```
alter table employee add foreign key (personal_id)
references person(personal_id);
```

Whereas the name of the attributes in the table *employee* and *person* to be referenced are the same, the name of the referenced attribute in the *person* table can be omitted.

```
alter table employee add foreign key (personal_id)
references person;
```

The relationship has been created. However, what about the relationship type (identifying / non-identifying)? Sure, if the foreign key is part of the primary key, the identifying relationship must be created.

What about cardinality? (*1:1*, *1:N*, *M:N*)? This is *1:N* cardinality, whereas the foreign key attribute is part of the composite primary key. Thus, one *person* can be listed in table *employee* multiple times. Vice versa, each *employee* references exactly one *person*.

Finally, what about the membership type (obligatory/optional)? Why? From the *person* to the *employee* table, it can be an optional relationship – a person does not need to be employed at all. However, there must be an obligatory relationship from the *employee* to *person* table because the foreign key value is part of the primary key. Thus, it cannot contain a *NULL* value at all.

More about the foreign key definition, management, and described principles are in [Lab 4 – Data modeling](#).

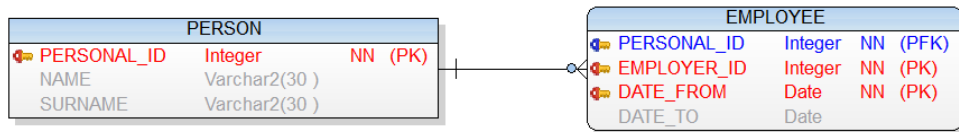


Fig. 5.3: Foreign key definition – Person, Employee table

Domain definition (check constraint)

The attribute value is characterized by the data type it belongs to (like *integer*, *varchar*, *date*, ...). Such data type can also be limited to particular values forming the user-defined *domain*. It can be done using the **check** constraint of the attribute. To describe the solution, add the attribute **job_type** to the *employee* table definition expressing the type of employment (*full time*, *part-time*). From the definition, the value must be in string format. However, another *check* constraint should be defined to ensure that only specified values can be inserted (or updated). The following code shows the table definition, therefore, *drop* the existing table and create a new one.

```
Create table employee
(
  personal_id char(11),
  employer_id integer,
  date_from date,
  date_to date,
  job_type char(9) check (job_type in ('full time', 'part time')),
  primary key(personal_id, employer_id, date_from)
);
```

If the definition is completed, one question arises, whether such solution is correct. The answer is easy – sure, it is. However, it is not practical. For each *employee*, it is necessary to store at least 9 characters, but the word “time” is always present. Therefore, it can be shortened to “full” and “part”. To get an effective solution, only one character is adequate. Thus, the size of the attribute can be lowered nine times. Much better, isn’t it? Imagine the complex system consisting of thousands of employees or even portal managing all employment in the country. Size effectivity is considerable. Therefore, always deal with the efficiency of the system.

```
Create table employee
(
  personal_id char(11),
  employer_id integer,
  date_from date,
  date_to date,
  job_type char(1) check (job_type in ('f', 'p')),
  primary key(personal_id, employer_id, date_from)
);
```

Default value

The *default value* can be optionally extended attribute definition. Thus, if no value is inserted, it will be automatically replaced by the defined default value. In the past, there was a significant difference between no value (not listed in the *Insert* statement) and *NULL* value listed explicitly. If any value were defined explicitly, no *default value* would be used. Thus, *NULL* was not replaced by the *default value* at all. In Oracle 12c version, a new clause – *default on null* – was introduced. Thus, if the value is undefined or not specified, it will be replaced by the default value. Consequently, a *NULL* value is replaced, as well.

Principles are described using a *job_type* attribute of the *employee* table. Let's assume that in a standard environment, we deal with the *full-time* job. Thus, the default value can look like the following. Notice that the *default* value must be syntactically defined before the *check* constraint:

```
Create table employee
(  personal_id char(11),
   employer_id integer,
   date_from date,
   date_to date,
   job_type char(1) default on null 'f' check (job_type in ('f', 'p')),
   primary key(personal_id, employer_id, date_from)
);
```

Constraint naming

Each constraint (*primary key*, *foreign key*, *unique*) can be optionally named using user-defined naming notation. We strongly recommend using your name due to later management. You will be clear about the meaning of the constraint, if necessary, to remove it. Otherwise, a system-generated name will be used.

```
Create table employee
(  personal_id char(11),
   employer_id integer,
   date_from date,
   date_to date,
   job_type char(1) default 'f'
   check (job_type in ('f', 'p')),
   constraint emp_pk primary key (personal_id, employer_id, date_from),
   constraint emp_fk_per foreign key(personal_id)
   references person
);
```

Create table as Select

Special opportunity for table definition can be provided by the *Create table as Select*. In that case, a new table is created based on the defined *Select* statement. One more time, it is inevitable to use a column alias for each attribute formed using the function. The rest attributes can be renamed using aliases optionally. Thus, the result of the function *nvl* and subtracting operation will be named as *duration*.

```
Create table employee_deposit as
select name, surname, personal_id, employer_id,
       nvl(date_to, sysdate) - date_from as duration
from person join employee using(personal_id);
```

Name	Null	Type
-----	----	-----
NAME		VARCHAR2 (15)
SURNAME		VARCHAR2 (15)
PERSONAL_ID		CHAR (11)
EMPLOYER_ID		NUMBER (38)
DURATION		NUMBER

If you add the condition to the *Select* statement, which will never be valid (e.g. primary key consisting of *NULL* values, which can never occur), the only structure will be defined, but the table will be empty.

```
Create table employee_deposit2 as
  select name, surname, personal_id, employer_id,
         nvl(date_to, sysdate) - date_from duration
  from person join employee using (personal_id)
  where personal_id is null;
```

Notice the constraints defined using such a command (*Create table as Select*). No *primary keys*, not *check* constraints, no *default* values are copied. Thus, if you create a deposit for the *employee* table, you can insert any character into the *job_type* attribute. The following *Insert* statements are valid. Whereas there is no primary key definition, the following *insert* statement can be executed several times without raising an error.

```
Create table employee_deposit3 as
  (select * from employee);
```

```
insert into employee_deposit3(personal_id, employer_id, date_from,
                             date_to, job_type)
values('000101/1234', 1, sysdate, null, 'x');
```

```
insert into employee_deposit3(personal_id, employer_id, date_from,
                             date_to, job_type)
values(null, 1, sysdate, null, 'x');
```

Thus, no *primary key* definition, no *check constraint* is copied. However, what about the *NULL* definition? It is a bit tricky. Sometimes it is valid, sometimes invalid. So, how it works? We will describe the principles using two simple tables (*T1*, *T2*) consisting of only one attribute – *ID* defined as a primary key. In the first case, *NOT NULL* is specified explicitly.

```
create table T1
  (id integer not null primary key);
```

```
create table T2
  (id integer primary key);
```

Now, create another two tables (*TT1* and *TT2*) using *Create table as Select* command based on tables *T1* and *T2*.

```
create table TT1 as
  (select * from T1);
```

```
create table TT2 as
  (select * from T2);
```

Get the schema of the tables.

```
desc TT1
```

Name	Null	Type
----	-----	-----
ID	NOT NULL	NUMBER(38)

```
desc TT2
```

Name	Null	Type
----	-----	-----
ID		NUMBER(38)

Try to insert *NULL* values into the newly defined tables. Is it possible?

```
insert into TT1 values (null);
```

```
Error report -
ORA-01400: cannot insert NULL into
  ("KVET_ENG"."TT1"."ID")
01400. 00000 - "cannot insert NULL into (%s)"
*Cause:      An attempt was made to insert NULL into previously listed
              objects.
*Action:     These objects cannot accept NULL values.
```

```
insert into TT2 values (null);
```

```
1 row inserted.
```

To conclude the *NULL* value management, it is necessary to highlight the definition of the attribute itself. If the *NOT NULL* constraint of the attribute is defined explicitly, it will be copied using *Create table as Select* command. Vice versa, if there is the only *primary key* definition (but there is no *NOT NULL* explicit definition), even though the primary key must always be *NOT NULL*, such constraint is not evaluated copied to the newly created table.

5.4.2 Alter command

As already partially described, each table definition can be later changed using the *Alter* command. Naturally, removing the table and creating a new one would be unsuitable (references, complex management, existing applications, etc.). Therefore, if there is a necessity to change the structure, the following notation can be used.

Alter table command has three primary variants of usage:

- *Add* – extending the table definition by another attribute or constraint.
- *Modify* – changing the column specification.
- *Drop* – removing the column or constraint.

Supplementary settings – *Rename*.

Add option

```
alter table table_name {add | modify | drop} ...
```

Following notations show the example of the main *Alter table* commands.

Adding new attribute – *passport number*. Notice that the defined value is noted as unique:

```
alter table person add passport_num varchar2(20) unique;
```

Adding primary key definition. Assume that there is no primary key definition of the table:

```
alter table person add primary key (personal_id);
```

Adding foreign key definition:

```
alter table employee add  
foreign key (personal_id) references person;
```

Modify option

The existing definition of the attribute can be changed using the *Alter table ... modify* commands:

```
alter table table_name {add | modify | drop} ...
```

Changing data type: increasing the size of the attribute is no problem at all. However, an attempt to decrease the size can raise an *exception* if the existing data have a bigger size than the limit to be set (one *Insert* statement is executed to highlight the limitations).

```
alter table person modify name varchar2(50);
```

```
insert into person(personal_id, name, surname)  
values('851210/1234', 'Michael', 'Flower');
```

```
alter table person modify name varchar2(10);
```

```
Table altered.
```

```
alter table person modify name varchar2(3);
```

```
Error report -  
ORA-01441: cannot decrease column length because some value is too big  
01441. 00000 - "cannot decrease column length because some value  
is too big"
```

Changing NULL/NOT NULL definition. Also, notice the previously described limitation. Thus, the *NOT NULL* definition can be added only if existing data do not contain *NULL* values in the particular attribute.

```
alter table person modify name not null;
```

```
alter table person modify name null;
```

Naturally, multiple definitions based on one attribute can be grouped.

```
alter table person modify name varchar2(30) not null;
```

Removing default value. The default value is not named constraint. It is necessary to remember that a *NULL* value will be set if no explicit default value for the attribute is defined. Thus, removing defined *default value* actually means replacing user-defined *default value* with *NULL*.

```
alter table employee modify job_type default NULL;
```

Drop option

```
alter table table_name {add | modify | drop} ...
```

Let's repeat the *employee* table definition with user-defined constraint names. Using explicit constraint naming is suitable if there is a necessity to remove the defined constraint. If a system-generated name is used, the particular value must be obtained by querying data dictionary views ([Lab 14 – Data dictionary views](#)). Notice that the name of the constraint must be unique.

```
create table employee
(
  personal_id char(11),
  employer_id integer,
  date_from date,
  date_to date,
  job_type char(1) default 'f',
  constraint check_job_type check (job_type in ('f', 'p')),
  constraint emp_pk primary key(personal_id, employer_id, date_from),
  constraint emp_fk_per foreign key(personal_id)
                        references person
);
```

Removing attribute:

```
alter table employee drop column date_to;
```

Removing primary key constraint:

```
alter table employee drop constraint emp_pk;
```

Removing foreign key constraint:

```
alter table employee drop constraint emp_fk_per;
```

Removing check constraint. Principles of removing *default value* have been proposed sooner. A different situation arises if the check constraint needs to be dropped. The solution is similar, based on using named *constraint*. Removing *check* constraint cannot be done using *Alter table ... modify* command.

```
alter table employee drop constraint check_job_type;
```

Table renaming

The table can also be renamed using the *Alter table* command. In the following example, the original table *person* is renamed to *person_tab*. The first part defines the syntax. The second one is an example of usage.

```
alter table table_name rename to new_name;
```

```
alter table person rename to person_tab;
```

However, also particular attribute can be renamed by using *rename column* keyword. In the following example, the attribute *surname* of the table *person* is renamed to *family_name*. The first part defines the syntax. The second one is an example of usage.

```
alter table table_name rename column orig_name to new_name;
```

```
alter table person rename column surname to family_name;
```

5.4.3 Drop command

If the database object is not necessary to be handled later, it can be removed from the system using the last DDL command type – **Drop**.

Can you drop the *student* table, now? If not, why? Look at the model. The answer resides in the referential integrity definition.

```
drop table student;
```

```
Error report -  
ORA-02449: unique/primary keys in table referenced by foreign keys  
02449. 00000 - "unique/primary keys in table referenced by foreign keys"  
*Cause:      An attempt was made to drop a table with unique or  
              primary keys referenced by foreign keys in another table.  
*Action:     Before performing the above operations the table, drop the  
              foreign key constraints in other tables. You can see what  
              constraints are referencing a table by issuing the following  
              command:  
              SELECT * FROM USER_CONSTRAINTS WHERE TABLE_NAME = "tabnam";
```

Can you *drop* the *study_subjects* table now? Yes, it is possible (but do not do it now, data will be necessary for future work).

```
drop table study_subjects;
```

```
Table dropped.
```

Now, it is possible to *drop* table *student* (but do not do it now, data will be necessary for future work).

```
drop table student;
```

```
Table dropped.
```

If you want to force the system to **Drop table** irrespective of the referential integrity, the keyword **Cascade constraints** can be used. However, it is not recommended to use it like this because it influences existing table definitions. (Do not do it now, data will be necessary for future work).

```
drop table personal_data cascade constraints;
```

Drop table *personal_data* using **Cascade constraints** keyword would influence the structure of *student* table, whereas it references *personal_data* table (using *personal_id* attribute). Foreign key based on the *personal_id* attribute is removed. Thus it can hold any value meeting other constraints.

Recycle bin

Let's create the table *person* as a copy of the *personal_data* table. Drop newly created table.

```
create table person as select * from personal_data;
```

```
drop table person;
```

Although the object (table *person*) has been removed from the system, it is still possible to reverse the operation. Executing the *Drop* command in its pure form reflects only the movement of the database object to another repository – *recycle bin*, from which it can be resumed (if sufficient disc space is allocated for recycle bin). The original table *Person* has been renamed to “*BIN\$\$QUdkj2ohEngUMGeOpRYw==\$\$0*”.

The content of the recycle bin can be obtained using one of the following commands (the second command will provide deeper characteristics):

```
show recyclebin
```

```
select * from recyclebin;
```

ORIGINAL NAME	RECYCLEBIN NAME	OBJECT TYPE	DROP TIME
PERSON	BIN\$\$QUdkj2ohEngUMGeOpRYw==\$\$0	TABLE	2017-02-21:06:54:32

Notice that original queries can be used if the original table name is replaced by *recycle bin* name. However, the table cannot be modified if it resides in *recycle bin*.

```
select name, surname, personal_id, employer_id,
       nvl(date to, sysdate)-date_from duration
from BIN$$QUdkj2ohEngUMGeOpRYw==$$0 join employee using(personal_id)
where employer_id = 1;
```

To restore the table from the recycle bin, a *flashback* command can be used.

```
flashback table person to before drop;
```

Optionally, such a table can be renamed using the *flashback* command.

```
flashback table person to before drop rename to person_renew;
```

However, notice, that not all *constraints* are resumed, but the data are. If you have multiple tables in the *recycle bin* with the same original names, the *LIFO* approach is used – the last object added to the *recycle bin* is taken back (renewed) as the first.

If you *drop* a database object using the *purge* keyword, it is not moved to the *recycle bin* but removed totally – there is no possibility to reverse the action. Do it very carefully.

```
drop table employee purge;
```

```
select * from recyclebin;
```

```
no rows selected.
```

Recycle bin can be flushed entirely using *purge recyclebin*, or only a particular object can be flushed.

```
purge recyclebin;
```

```
purge table table_name;
```

```
purge table person;
```

Notice that the cleaning process automatically removes database objects from the *recycle bin* if another object must be placed there and no free space is located.

5.5 Index

Oracle defines an index as an optional structure associated with a table or table cluster to *speed data access*. By creating an index on one or more columns of a table, you gain the ability to retrieve a small set of randomly distributed rows from the table quickly. During individual destructive *DML* statement execution, the index is built, respectively reconstructed. It contains locators to the physical structure on the leaf layer – pointers to the physical files – *ROWIDs*.

Fig. 5.4 shows the index structure.

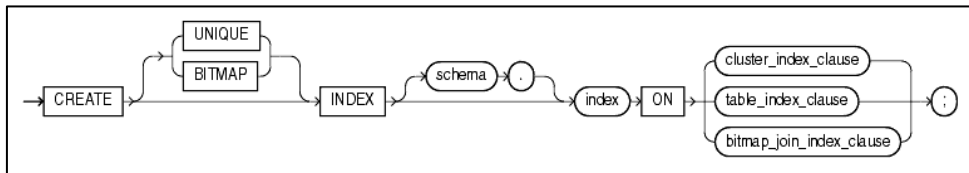


Fig. 5.4: Index definition; source: docs.oracle.com

The purpose of the database index is similar to an index in the back of a book (it associates a topic with a page number):

- + topic appears on a few pages
- usefulness decreases with an increase in the number of times a topic appears in a book.

An index creates an interlayer forming logical and physical independence of associated data. By using it, I/O disk operations are reduced, consequencing in better performance. The index can be created either implicitly (unique constraint of the attribute, primary key) or explicitly by using the *create index* command. The main advantages of indexes are following:

- improving SQL statement performance,
- enforcing uniqueness of the primary key and unique key constraints,
- reducing locking issues with parent and child tables associated via primary and foreign keys.

Notice that for primary keys, indexes are created automatically. Vice versa, foreign keys are not associated with the index. Instead, they only use the unique index in the referenced table. Generally, it is useful to create explicit indexes for foreign keys due to table joining and access reduction to the referenced table.

5.5.1 ROWID

The *ROWID* pseudo column is associated with each row in the database and returns the physical address of the row. It contains all information necessary to locate a row stored in 10 bytes):

- **The data object number** (1–32 bits)
- **Data file** in which the row resides (the first file is 1; file number is relative to tablespace) (33–44 bits)
- **Data block** in the data file in which the row resides (45–64 bits)
- **The position of the row** in the data block (the first row is 0) (65–80 bits)

ROWID values have several important uses:

- they are the fastest way to access a single row,
- they can show you how the rows in a table are stored,

- they are unique identifiers for rows in a table.

Although it provides unique value within a table, do not use it as the primary key for several reasons. First of all, it reflects the *ROWID* data type and requires 10 bytes. Moreover, these values can be changed over time (e.g., by using *import*, *export* functionality, *flashback*, *shrinking* space, *moving* data, etc.).

5.5.2 Index management

Access approach to the data during the retrieval is an automatic process controlled by a database optimizer. The decision, whether the index will be used or not (and which one, if several are defined), is based on *statistics* and an optional *SQL profile*. Therefore, it is inevitable to have correct and actual statistics to reach (sub-)optimal performance. Notice that by default, statistics are generated and calculated during maintenance windows in the weak workload of the database (usually at night).

Several indexes can include a particular column. A critical component is just the index type and order of the attributes (if the composite index is defined). These factors significantly influence performance. Therefore, it is also necessary to take care of it during primary key definition or when the composite index is created. Attribute order forming index is essential. The most often used attribute in the *Where* clause of the query should be listed first, likewise others. When looking at the primary key of the table *study_subjects*, the following importance list is assumed (top is *student_id* followed by *subject_id* and *school_year*). Thus, most query conditions should be based on *student_id*.

```
primary key (student_id, subject_id, school_year)
```

Notice that incorrect order of the attributes forming index can cause significant performance degradation.

5.5.3 Types of indexes

B+ tree index type

B+ tree index is a default type used in databases. Table row identifier (*ROWID*) and associated column values are stored within index blocks in a *balanced tree structure*. An essential property of such an index is the fact that it cannot manage *NULL* values at all. It is formed by the *root node*, *internal nodes*, and *leaf nodes* consisting of *ROWID* pointers. Data on the *leaf layer* are ordered and connected via the double-directional linked list. Fig. 5.5 shows its architecture.

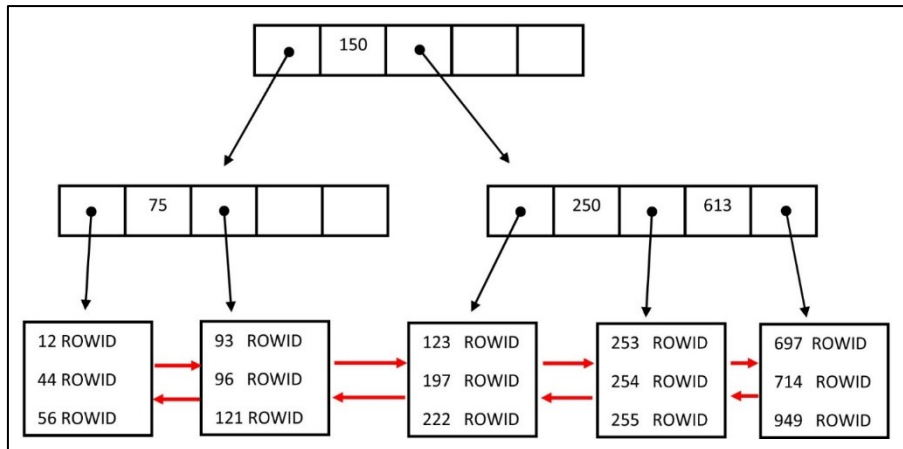


Fig. 5.5: *B+ tree index*

This is the syntax and example:

```
Create index ind_name on table_name (atr1 [, atr2, ...]);
```

```
Create index ind_ns on personal_data(surname, name);
```

The **reverse B+ tree** index approach is a specific type of *B+ tree*. It stores index entries with their bytes reversed. The problem of standard *B+ tree* index is just consecutive values – sequences and necessity to index block reconstruction – balancing (*B+ tree* structure is balanced) when data are inserted or updated. Thus, the **reverse B+ tree** index requires smaller server sources and provides performance benefits (for destructive *DML*). But, on the other hand, data on the leaf layer are not ordered, which can degrade performance if the condition is based on a range (non-equality).

The following figure shows the structure of the **reverse B+ tree** index.

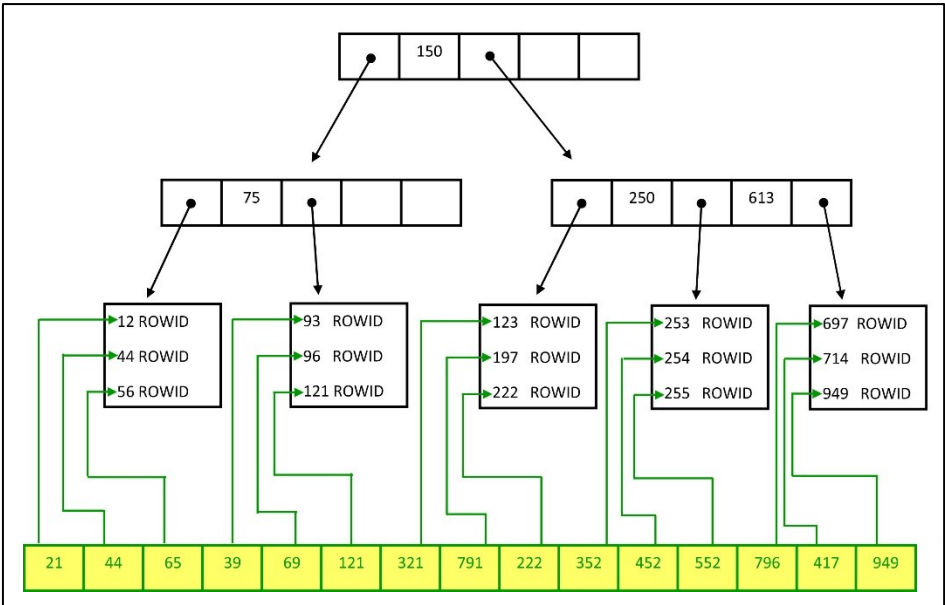


Fig. 5.6: Reverse key B+ tree index

Tab. 5.3: Original and indexed value

Original value	Indexed value
12345	54321
3489	9843
FRI	IRF

This is the syntax and example:

```
Create index ind_name on table_name (atr1 [, atr2, ...]) reverse;
```

```
Create index ind_st on student(student_id, st_group) reverse;
```

Another *B+ tree* index type is based on *functions*. It does not cover direct column values, but it is created based on SQL functions or expressions. Notice that if you use the user-defined function, it must be *deterministic*.

This is the syntax and example:

```
Create index ind_name
on table_name (func_name(param_list) [, ...]);
Create index ind_func_st
on personal_data(func_gender(personal_id), name, surname);
```

```
Create or replace function Func_gender(p_id char)
return char deterministic is
begin
case
when substr(p_id, 3,1) in (5,6) then return 'female';
when substr(p_id,3,1) in (0,1) then return 'male';
else return 'unknown';
end case;
end func_gender;
/
```

Multiple functions can form a function-based index. There can be a combination of direct attributes and functions, as well. For function definition, reference chapter Lab 9 – Procedures, functions and packages.

Bitmap index

A *bitmap* index is primarily suited for data warehouses and decision support systems – many rows, low-value variability. It is based on *star schema* – central fact table and number of related dimension tables. The aim is to monitor values in multiple dimensions over time.

Fig. 5.7 is based on the billing process and invoicing. *Billing_fact* as a core table, dimension tables are formed by *Billing_Date_dimension*, *Time_dimension*, *Geography_dimension*, and *Product_dimension*.

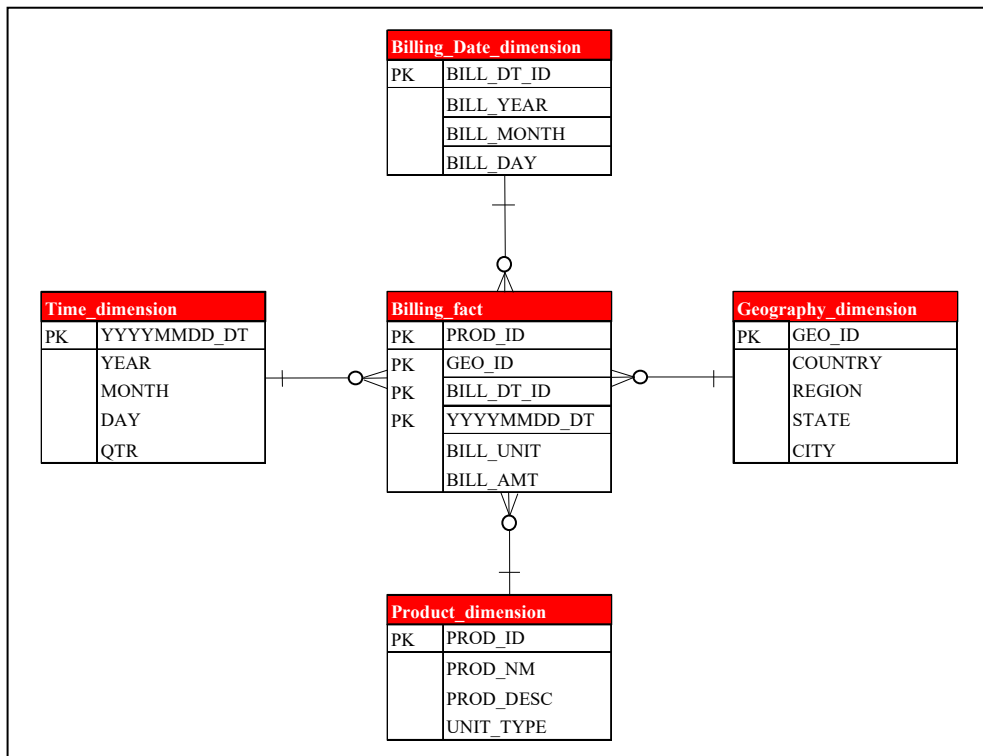


Fig. 5.7: Bitmap index

Guidelines for using *bitmap* index:

- It should generally be used on low cardinality columns.
- It manages *NULL* values automatically.
- It is suitable for many queries that join or filter on indexed columns.
- It is suitable for no (or very low amount) destructive DML activity.

Drop the bitmap indexes before updating tables and *recreate* them after the DML operations on tables are complete.

For OLTP, *bitmap* indexes are not appropriate (many DML operations and row locking) => significant performance degradation.

This is the syntax and example (assuming, that *personal_data* table contains also *gender* attribute. If not so, a particular value can be obtained by using the function):

```
Create bitmap index ind_name
on table_name (atr1 [, atr2, ...]);
```

```
Create bitmap index ind_b_pd on personal_data(gender);
```

Index organized table

Index organized table is physically stored like Oracle *B+ tree* index – all of the data are stored within the index. Therefore, it does not cover physical *ROWIDs* at all.

There are physical differences in comparison with the standard heap organized table supported by the B+ tree index. On the other hand, access is the same as any other Oracle table.

It is typically defined for:

- thin tables (without too many columns),
- multiple column primary key.

Notice that an *index-organized table* is created based on the primary key, which must be present. Otherwise, an error will be raised:

```
ORA-25175: no PRIMARY KEY constraint found
```

The syntax of the *index-organized table* is based on adding the *organization index* keyword to the end of the table definition.

```
Create table personal_data
(
    personal_id    Char(11)          NOT NULL,
    name           Varchar2(15),
    surname        Varchar2(15),
    street         Varchar2(20),
    town           Varchar2(50),
    zip            Char(5),
    nationality     Char(2),
    primary key (personal_id)
) organization index;
```

5.5.4 Access methods

Access path selection is one of the essential parts of the optimizer decision. It significantly influences the principles and performance of data *retrieval*.

Generally, two basic types of access paths are defined:

- **Full Table Scans (Table access full)** – all blocks (rows) of the table are scanned.

It is mainly used when:

- a large portion of the table's data is required,
- the accessed table is small, consisting of few blocks,
- no suitable index is defined.

- **Index Access Paths** – index is used for accessing particular data.

Selecting the index access method is the task of the optimizer, and its decision is based on the index definition itself and statistics, defined query, etc. The *hints* can partially influence it.

Several categories can be distinguished – *index unique scan*, *index range scan*, *index skip scan*, *index full scan*, *fast full index scan*, etc. Their characteristics, properties, and limitations can be found in [48] [49].

5.6 Practice

1. Create table *T1* with a primary key *ID*. Then, add another *string* attribute (*varchar2* / *char* data type) and at least one more attribute (*Integer* data type).
2. Create table *T2* with a *composite primary key* containing attribute *ID* and *valid_from* (data type *Date*). Be sure that the data type of the attribute *ID* is the same as the attribute *ID* in the *T1* table.
3. Create a *relationship* between tables *T1* and *T2* (based on *ID*). The cardinality of the relationship should be *1:N*. Will it be *identifying* or *non-identifying*? Why? How can you influence it?
4. Add another attribute, “*note*” to the *T2* table (choose the appropriate data type).
5. Create table *T3*. The primary key of the table should be *ID* with data type *integer*. Ensure that only even values can be inserted.

```
alter table T3 modify id check(mod(id,2)=0);
```

6. Create *M:N relationship* between *T1* and *T3*.

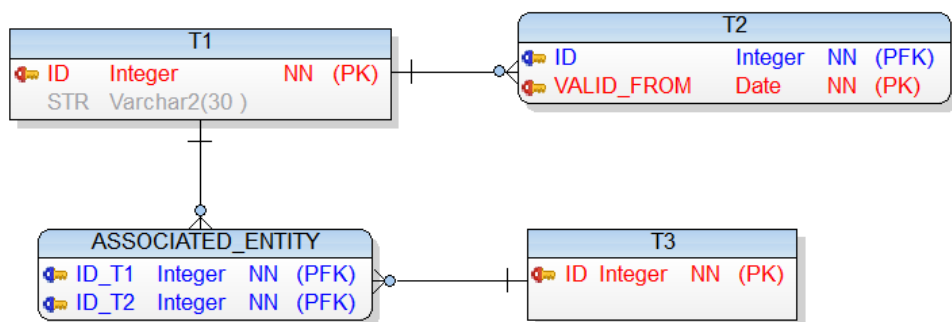


Fig. 5.8: Model for practice

7. Drop all created tables (*T1*, *T2*, *T3*, *associated_entity*) respecting the correct operation order.
8. Create another table *T1* with different attributes and drop it consequently.
9. Renew the tables from the *recycle bin*. Check the renewed constraints. What about table *T1*? Is it possible to restore the older one?
10. Create an index based on the *name* and *surname* of the person (table *personal_data*).
11. Try to create an index based on the *personal_id* of the person (table *personal_data*). Is it possible? Why not?
12. Try to create an index based on *student_id*, *subject_id*, *school_year* (in that order) of the *study_subjects* table. Is it possible? Notice that the primary key of the *study_subjects* table is following:

```
primary key (student_id, subject_id, school_year)
```

13. Try to create an index based on *subject_id*, *student_id*, *school_year* (in that order) of the *study_subjects* table. Is it possible? Be aware the order of attributes is significant.

Lab 6 – Data loading

*Data loading is essential during the data layer migration or by moving a huge data set to the database. Three techniques are proposed and discussed. **SQL Loader** is a general solution mostly referencing other systems by managing data in the textual form input (TXT, CSV files, etc.). Oracle database import and export can be done on the client or server layer. The Oracle directory mapping object operates server file system accessibility. Although server processes are now more preferred due to the performance, we also mention the principles of client site data management.*

In this lab, we will drive the reader through the process of data loading to the cloud environment. It requires access to the object storage and data containers (buckets).

6.1 Introduction

Data loading is a complex process of copying and loading data sets from the external data source – file or application to the database. Individual database systems provide various technologies for data loading, from generating *Insert* statements up to binary copies of the data tuples. For this subject, we will describe three techniques. The first one is based on the *SQL Loader* tool. Using this approach, several data file structures like *CSV* can be used, which is very useful when imported from third-party systems. The second and third approaches are based on *import* functionality and *data pump*. *Imp* is an older approach, which is subsequently replaced by newer technology. The *Imp* approach is based on the client site, which generates data to be imported as conventional *Insert* statements. Oracle 10g introduced a *data pump* facility, which significantly extends the possibilities and speed of the processing, whereas such solution is server site oriented. Thus, these two solutions (*imp* and *data pump import*) have entirely different architecture and cannot be combined.

For this lab, it is inevitable to follow all instructions. Correct data loading is necessary for subsequent database activities and queries. In this lab, the theoretical introduction is directly linked to the examples and activities you perform.

6.2 SQL Loader

SQL Loader allows you to *insert* data into the database using multiple format types. Associated *control file* delimits the structure. Thanks to that, it provides a sophisticated tool to convert and *insert* data from other systems. It is a user process, which inserts data using **conventional way** (*insert* statements are generated with regards to the *UNDO* and *REDO* data logs) or **direct path** (in this case, the *buffer cache* is bypassed, and data are loaded directly to the data files). No *UNDO* is generated. Moreover, it is possible to disable also *REDO* logging for this operation). Thus, the *Direct path* is far faster than the conventional method, but some negatives must be mentioned. *Referential integrity* control mechanisms must be *disabled* or *dropped*. Whereas the operation is not standard *Insert* statement execution, particular *Insert triggers* do not fire. Moreover, the processed table is *locked* against *DML* statements executed by other sessions. Vice versa, *primary key* and also *NULL* value constraints are managed consistently.

We have prepared a **library data model** with data filled to perform data loading operations using *SQL Loader*. So, follow the instructions:

1. Download the file archive from your USB media, respectively server (*SQL_load_library.zip*). It consists of the data necessary to be loaded into the database. It contains three file types, which can be differentiated by the extensions:

```
*.sql    SQL file - DDL statements for creating database schema
         (tables, relationships, ...),
*.unl    data files with the values to be loaded into the database,
*.ctl    control files containing instructions, how to load UNL data
         to the database (format, delimitation, etc.)
```

2. Create data model schema objects using the **SQL file** (copy the file to the server and launch its execution – file *library.sql*). In SQL developer, such file can be directly opened, and script launched.

```
start library.sql
```

3. Create missing **control files** for correct data loading (*person.ctl*, *author.ctl*). Be aware of the *schema of the table*, but also appropriate attribute order in particular **UNL file**.

The easiest way to creating a missing *control file* is based on copying another existing file. First, it is necessary to modify the *name* of the *control file*, the *table name* to which we would like to *load* appropriate data, and a *list of attributes* (columns).

The order of columns in the *control file* must reflect data in the *UNL file*, not the order in the *schema definition*. In our case, the order in the schema and the data UNL file is the same, but it generally does not need to be like that.

For correct *Date* attribute loading, it is necessary to set a suitable input data format. In our case, the order is *month/day/year*.

```
ACCEPTANCE_DATE DATE 'DD.MM.YYYY'
```

Thus, essential data files necessary for successful loading are the following. Notice that the crossed file names are missing and must be created by you.

Tab. 6.1: SQL Loader files

<i>Table_name</i>	<i>Data file</i>	<i>Control file</i>
K_person	Person.unl	Person.ctl
K_reader	Reader.unl	Reader.ctl
K_book	Book.unl	Book.ctl
K_title	Title.unl	Title.ctl
K_rent_books	Rent_books.unl	Rent_books.ctl
K_authors_of_book	Authors_of_book.unl	Authors_of_book.ctl
K_author	Author.unl	Author.ctl

The *Control file* structure looks like the following example. The first part deals with data *location* (**INFILE 'book.unl'**) and *table*, to which data should be loaded (**INTO TABLE book**). The individual attribute value must be delimited in some way. In our case, the delimiter is a pipe (|) – **FIELDS TERMINATED BY '|'**. Afterward, the data structure definition is proposed – order of the data represented in *.UNL file. Do not forget to define a format for *Date* data type attributes.

```

LOAD DATA
INFILE 'book.unl'
INTO TABLE book
FIELDS TERMINATED BY '|'
(
  BOOK_ID,
  TITLE_ID,
  PRICE,
  REGISTRATION_DATE DATE 'MM/DD/YYYY',
  DISPOSAL_DATE DATE 'MM/DD/YYYY',
  LOST_DATE DATE 'MM/DD/YYYY'
)

```

DISPOSAL_DATE Data inside the *UNL* file looks like this (for table book):

```

279|17|9|09/23/2002|08/02/2014|12/29/2014
280|42|2|02/06/2012|08/17/2014|10/19/2014
281|81|3|12/13/2001|06/15/2014|12/17/2014

```

4. Load the data into *tables* using defined *control files* and *data files*. **Do not forget to use the correct order of operations** (table reflecting another table *primary key* must be loaded later, whereas *foreign key* value must refer to existing data) – the operation order is the same as *Insert* statement order. The loading process can be done using the following command (*Linux*).

```
$ sqlldr login@connect_string control='control_file_name.ctl'
```

We will launch the SQL Loader tool from the Instant client, so the steps are following:

- Start the *SQL*Plus (SQL Client)* application and connect to the *libraryDB* cloud database.
- Provide the credentials (admin and connect identifier or connect string, respectively).
- After successful login, the first letters of the row should be “*SQL>*”.
- As stated, *SQL Loader* is an external tool. Invoking it from the *SQL*Plus* environment requires you to call operating system activity, so the “*host*” command will enclose the original statement:

```
host sqlldr login@connect_string control='control_file_name.ctl'
```

```

Enter user-name: admin@librarydb_high
Enter password:
Last Successful login time: Tue Mar 16 2021 10:22:28 +01:00

Connected to:
Oracle Database 21c Enterprise Edition Release 21.0.0.0.0 - Production
Version 21.2.0.0.0

Ahoj Michal :)

PL/SQL procedure successfully completed.

SQL> host sqlldr admin@librarydb_high control='title.ctl'
Password:

SQL*Loader: Release 19.0.0.0.0 - Production on Tue Mar 16 15:34:27 2021
Version 19.8.0.0.0

Copyright (c) 1982, 2020, Oracle and/or its affiliates. All rights reserved.

Path used:          Conventional
Commit point reached - logical record count 100

Table K_TITLE:
 100 Rows successfully loaded.

```

Fig. 6.1: SQL Instant client – SQL Loader tool

5. **Check the correctness immediately after the loading. Results shown on the screen do not reflect the number of inserted data, but only the number of rows read from the *.UNL data file. If there is any problem, solve it before continuing (errors are mainly based on integrity constraints violation).**

Appropriate information about the execution process can be found after starting *control file* execution. The file has the same name as the *control file* but contains the extension “*.log”, e.g.:

```
person.ctl --> person.log
```

The log file will consist of *error* information about the *refused* rows to be loaded (*refused rows are directly stored in the file with BAD extension*) and also information about a number of successfully read (from the file) and loaded (into database) rows.

Example of the *LOG* file:

```

SQL*Loader: Release 21.0.0.0.0 - Production on Mon Mar 21 09:54:28 2022
Version 21.3.0.0.0

Copyright (c) 1982, 2021, Oracle and/or its affiliates. All rights reserved.

Control File:          person.ctl
Data File:             person.unl
Bad File:              person.bad
Discard File:          none specified

(Allow all discards)

Number to load:        ALL
Number to skip:        0
Errors allowed:        50
Bind array:            64 rows, maximum of 256000 bytes
Continuation:          none specified

```

```

Path used:          Conventional

Table K_PERSON, loaded from every logical record.
Insert option in effect for this table: INSERT

Column Name          Position      Len  Term      Encl Datatype
-----
NAME                 FIRST          *   |      CHARACTER
SURNAME              NEXT           *   |      CHARACTER
PERSON_ID            NEXT           *   |      CHARACTER
STREET               NEXT           *   |      CHARACTER
ZIP                  NEXT           *   |      CHARACTER
TOWN                  NEXT           *   |      CHARACTER
DISTRICT             NEXT           *   |      CHARACTER
REGION               NEXT           *   |      CHARACTER
STATE                 NEXT           *   |      CHARACTER

Table K_PERSON:
100 Rows successfully loaded.
  0 Rows not loaded due to data errors.
  0 Rows not loaded because all WHEN clauses were failed.
  0 Rows not loaded because all fields were null.

Space allocated for bind array:      148608 bytes (64 rows)
Read buffer bytes:                  1048576

Total logical records skipped:      0
Total logical records read:      100
Total logical records rejected:      0
Total logical records discarded:      0

Run began on Ne Sep 17 18:26:39 2017
Run ended on Ne Sep 17 18:26:42 2017

Elapsed time was:                    00:00:02.50
CPU time was:                        00:00:00.03

```

The principle of creating *UNL* file is expressed by the following example written in SQL. Moreover, explicit *line feed* can be added using *chr(10)*.

```

set echo off newpage 0 space 0 pagesize 0 feed off
spool author.unl
  select trim(name) || '|' ||
         trim(surname) || '|' ||
         trim(author_id) || '|' ||
         to_char(registration_date, 'MM/DD/YYYY') || '|' ||
         trim(note) || '|' from author;
spool off

```

SQL Data Loader can be invoked from the *SQL Developer client* (desktop) environment, as well. It is provided by the wizard, so the steps are a bit easier and maybe more user-friendly.

Connect to the cloud instance of the library database. Expand the list of tables in the left panel. By right-clicking on the particular table name, select the “*Import Data...*” option.

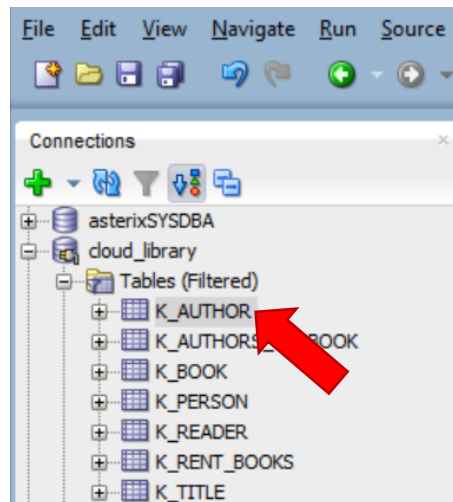


Fig. 6.2: SQL Developer – Import data (SQL Loader variant) (1)

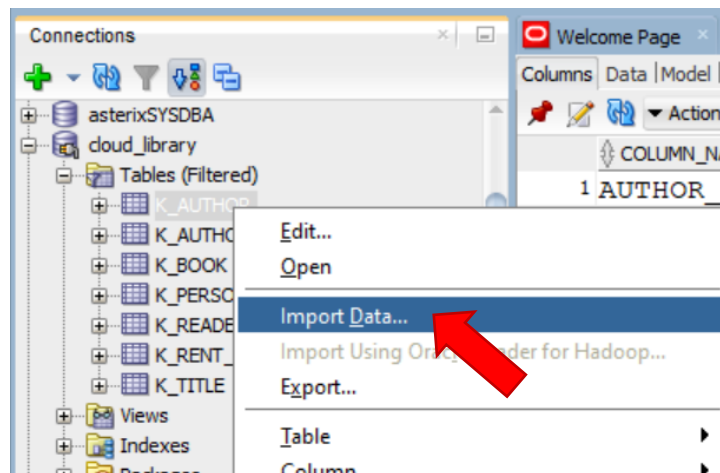


Fig. 6.3: SQL Developer – Import data (SQL Loader variant) (2)

Navigation wizard will be launched consisting of five steps. The first step defines data source and structure. Select the source file and file format. In our case, data are not enclosed by the special characters. Individual values are delimited by the pipe (|), whereby the rule is that one line in the source file corresponds to one inserted record.

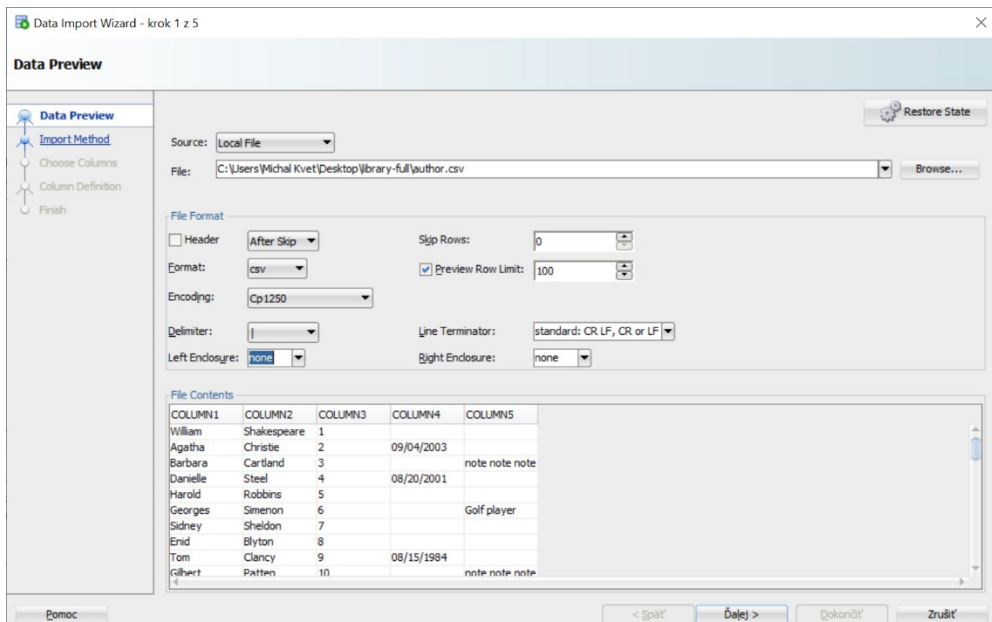


Fig. 6.4: SQL Developer – Import data wizard (1)

The next part consists of the definition of the *Insert type* – either *Insert*, *Insert script*, and *SQL Loader*. We will use the *SQL Loader Utility* option with no limit.

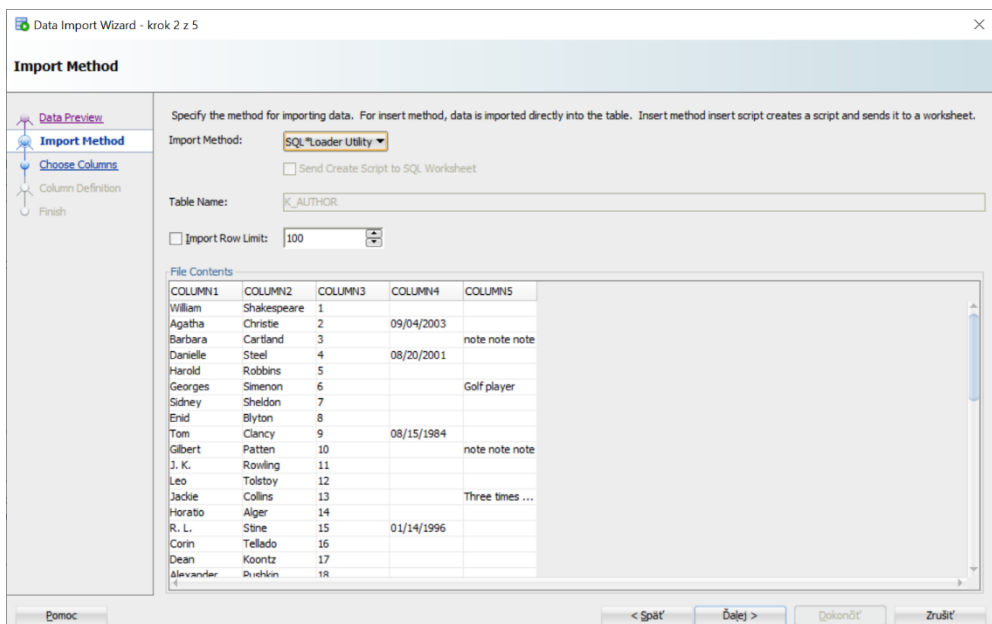


Fig. 6.5: SQL Developer – Import data wizard (2)

In the third phase, *column mapping* must be done. The original data source does not have headers, so columns are named sequentially. Mapping is in the right part. Combine individual source columns to the table attributes. The order must correspond to the order of columns

in the source file. In our case, it reflects *name*, *surname*, *author_id*, *registration_date*, and *note*.

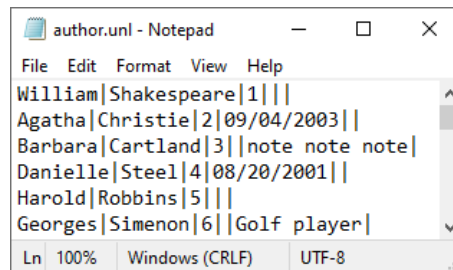


Fig. 6.6: Data source file

Always take emphasis on the *Date* or *Timestamp* data types. It is necessary to specify the element format to ensure proper loading. In our case, the *Date* is delimited by the *MM/DD/YYYY* format.

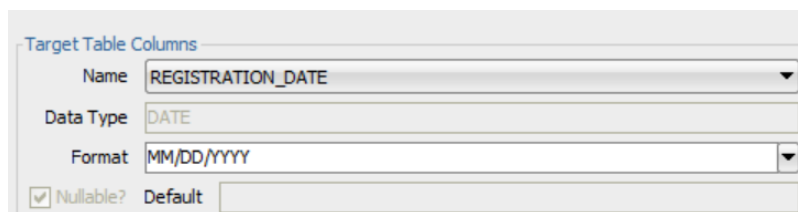


Fig. 6.7: Table column property definition

Optionally, you can specify *default values*, which will be used, if no data value is provided (step 3).

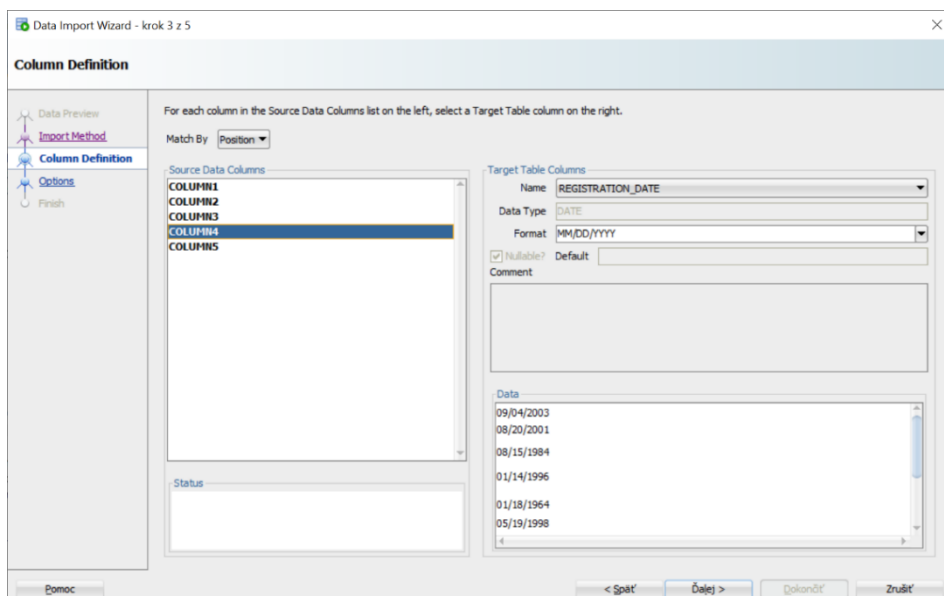


Fig. 6.8: SQL Developer – Import data wizard (3)

Navigate the wizard to the fourth step by clicking on the *Next* button. There, output files are specified. As already stated, the *BAD* file consists of the source file lines, which were

not loaded. *Log file* consists of the process monitoring and result description, list of raised exceptions is there.

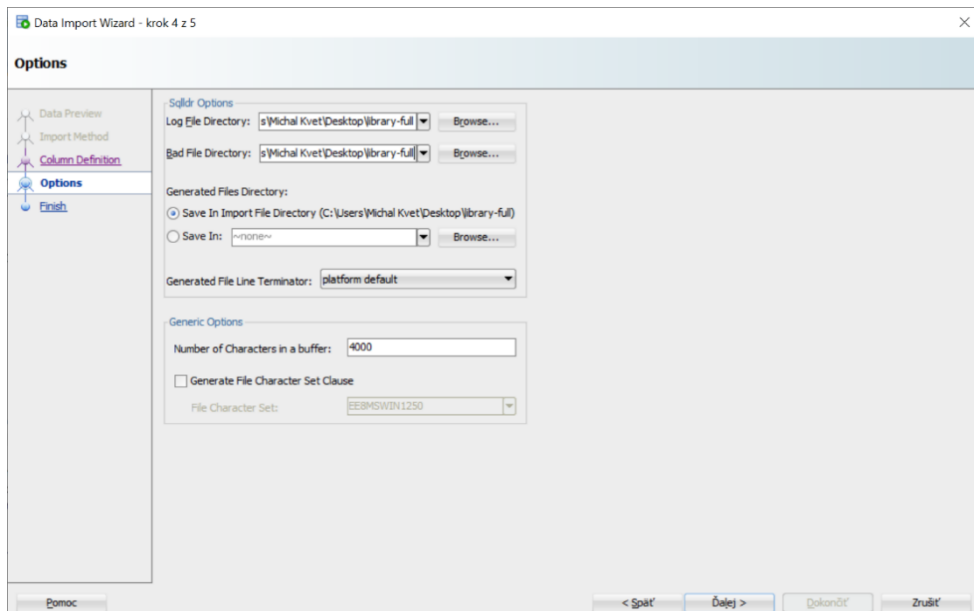


Fig. 6.9: SQL Developer – Import data wizard (4)

Finally, the summary is provided, and the whole process can be launched:

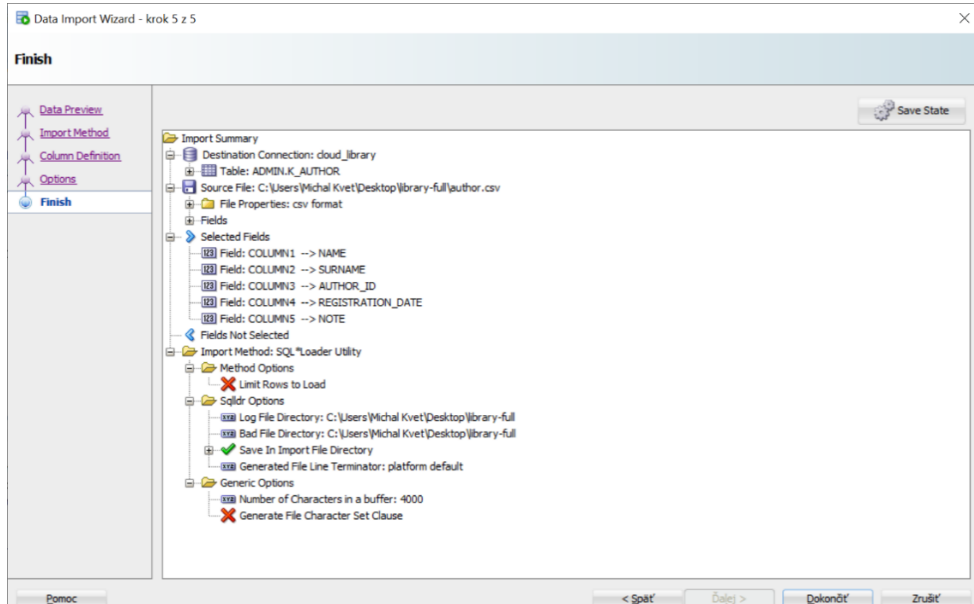


Fig. 6.10: SQL Developer – Import data wizard (5)

After launching the process, wait a bit for loading to be finished. In the repository, a log file and a bad file are created if any issue occurs. In our case, no problem should be identified.

Like the *SQL Developer desktop* version, *SQL loader* can be launched in the *SQL Developer Web* using the *Data loading* tab. The process and wizard are analogous. We will, therefore, skip the step-by-step definition.

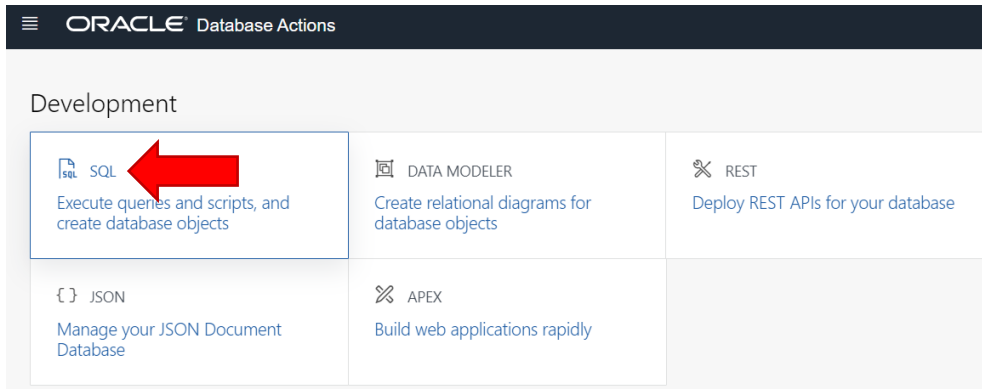


Fig. 6.11: *SQL Developer Web – Import data (1)*

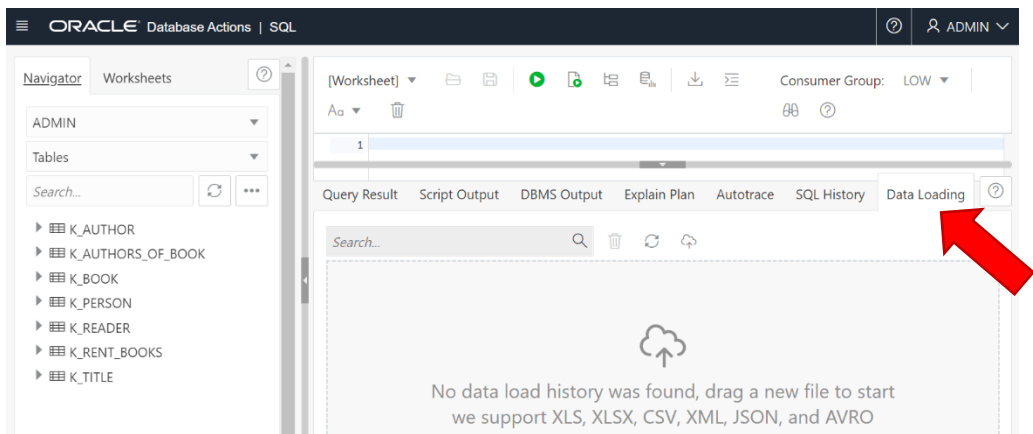


Fig. 6.12: *SQL Developer Web – Import data (2)*

6.3 EXP / IMP utility

1. If all previous operations ended successfully, backup the table structures and data using the *exp* client. It should be executed in the OS environment. The command consists of three parts – *credentials to the database* (password will be requested by the system. If you want also to write password explicitly, use the second command) followed by the *list of tables* to be exported (keyword *tables*) and *file*, to which data should be exported. Notice that the table names are delimited by the space or by commas.

```
$ exp login@connect_string tables='list_of_tables' file='file_name.exp'
```

```
$ exp login/password@connect_string  
    tables='k_person k_reader' file='library.exp'
```

If invoked from the *SQL Instant client* environment, operating system tool is launched by using host prefix.

```
host exp login@connect_string
      tables='k_person k_reader' file='library.exp'
```

2. If the export process is done successfully, then *drop* all the exported tables. Notice that the correct order must be used (based on referential integrity) – in principles, reverse to loading.

```
drop table table_name;
```

3. Then, use the prepared export and load data back into the database. When requested, use your credentials.

```
$ imp login@connect_string file='filename.exp'
```

In case the export file has not been created by you, but by another user, you must code it explicitly by adding the *fromuser* clause.

```
$ imp login@connect_string fromuser=old_login file='filename.exp'
```

Standard users can import data only to their schemas. However, the user with *DBA* privileges can import data to any user schema using the following command. Clause *touser* defines the schema name to which data should be loaded. (If you use the localhost database, the *SYSTEM* user has *DBA privileges*, so you can try it).

```
$ imp login@connect_string fromuser=old_login touser=new_login
      file='filename.exp'
```

Whereas it is invoked from the *Instant client* environment, the operating system tool is launched using the host prefix.

```
host imp login@connect_string file='filename='library.exp'
```

6.4 Creating import/export using dump files

Data pump (DP) import/export has been introduced in version *Oracle 10g*. At the same time, the original approach (*imp*, *exp*) has been marked as deprecated. Although future versions will not support old export functionality, the import will still be available for compatibility with older versions. Compared to a previously described solution, the *Data pump* is a *server process*, not a *user process*, and is managed by *Data Pump Master Process* and *Workers*. It can also generate SQL files.

Note: Please distinguish between the operating system directory and the Oracle directory in the next section. The whole process is multi-step. Operate carefully to reach the results.

Login expression in the next section expresses your real login to Oracle cloud and should be replaced in your code.

6.4.1 Import using data pump

As already stated, data pump functionality is, in comparison with *exp* or *imp* functionality, executed on the server-side. It is, therefore, necessary to copy the export file to the cloud storage and make it available for the database and management processes. We will

need *Object storage* and *Credentials* to access the objects by the database and to allow you to manage the import process using the locally installed *SQL developer* tool. Access to a cloud account using *SQL developer* and *Wallet* has already been shown.

Object storage

Oracle Cloud Object Storage is high-performance cloud storage – reliable, resistant, and cost-efficient data repository. Object storage repository can cover an unlimited number of files with any structure. *Always Free* version is limited to 20 GB of the capacity. *Oracle Cloud Object Storage* data can be easily, safely, and securely managed and retrieved by the internet or by using a cloud platform. It is not associated with a specific *compute instance*. The core element is the region itself.

Object storage can be accessed from the left panel of the *Cloud management* by selecting the *Storage* option.

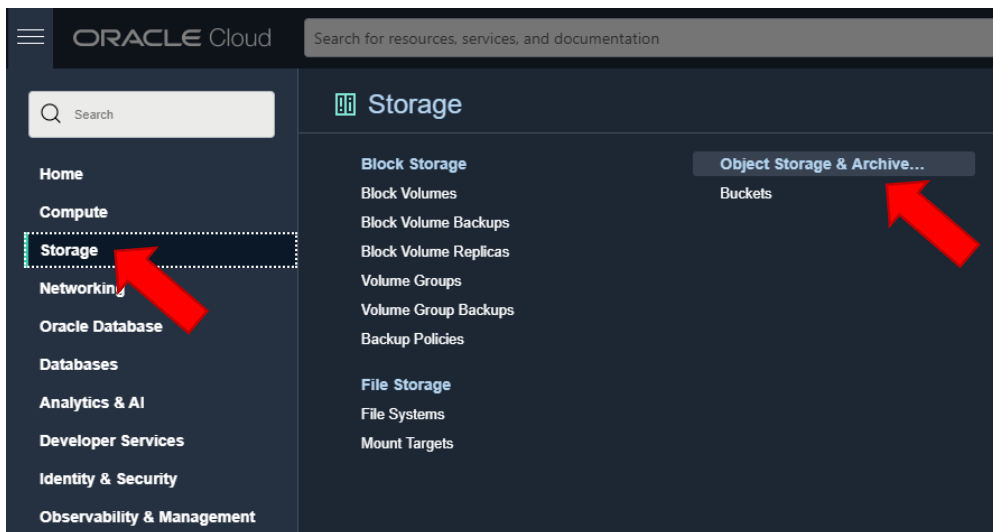


Fig. 6.13: Object storage

Object storage consists of the *buckets* holding data files themselves.

Bucket

The *bucket* is a logical container for storing data files. Each bucket is created and associated with the *compartment* delimited by the policies limiting the actions, which can be done there. We will use the general term “*object*” as a file of any data structure and format for cloud storage. The *object* is defined by its representation and metadata. Each *object* is stored within the *bucket*.

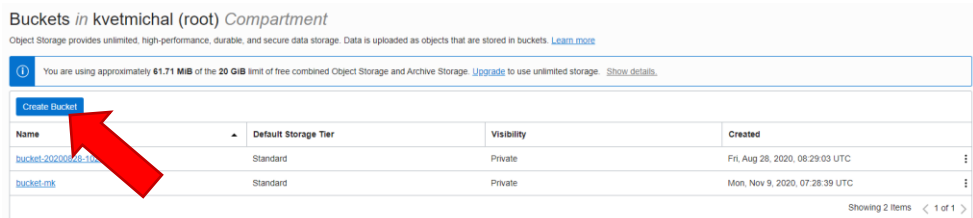


Fig. 6.14: Bucket creation (1)

To create a new *bucket*, the following parameters are defined:

- *Bucket name*.
- *Bucket storage tier* – *standard* or *archive*.
- *Object auto-tiering* option (*enabled* or *disabled*) allowing the system to move infrequently accessed objects to the less expensive storage repository (if the paid option is used).
- *Object versioning* option (*enabled* or *disabled*) storing all versions of the data object in case of creating and uploading new object version or by deleting and overwriting object, respectively.
- *Emit object events* – automation of the state changes (for the *object* of the whole *bucket*) using pre-defined events (like user notifications) – CRUD (create, read, update, delete) operations
- *Encryption type* using either *Oracle managed keys* or *customer-managed keys*.
- *Tags* – metadata, by which the resources can be categorized and tracked inside the *tenancy*. *Tags* consist of pair – *key* and *value*.

Create a new bucket for the dump file repository. Name it *bucket_library*, whereas it will hold relevant import, export, and log files for data pump operations. Let the *bucket type* set as standard. Object changes do not need to be monitored or versioned. Let *encryption* be done and managed by the Oracle.

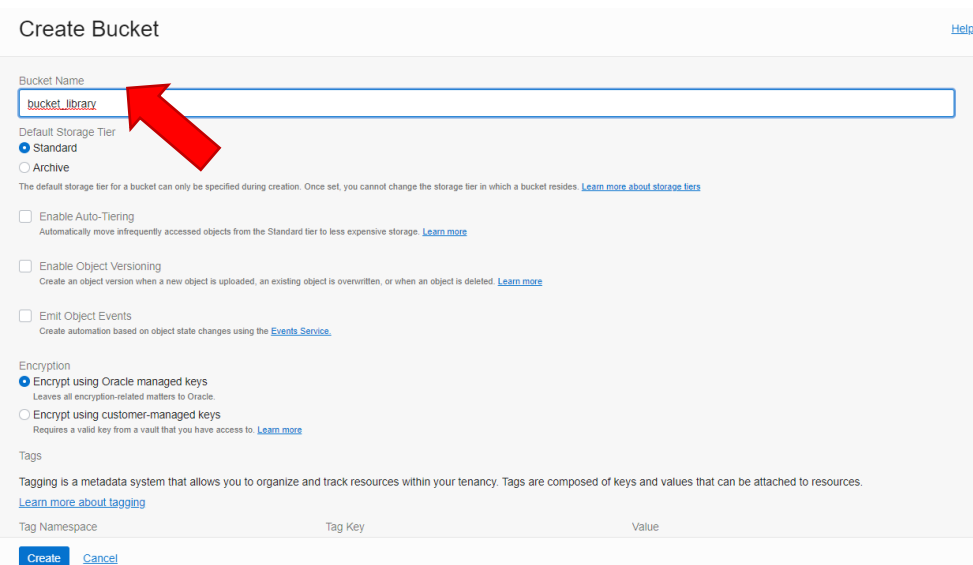


Fig. 6.15: Bucket creation (2)

Press the *Create* button and wait for the system to create the bucket. After the creation, a new bucket will be part of the list.

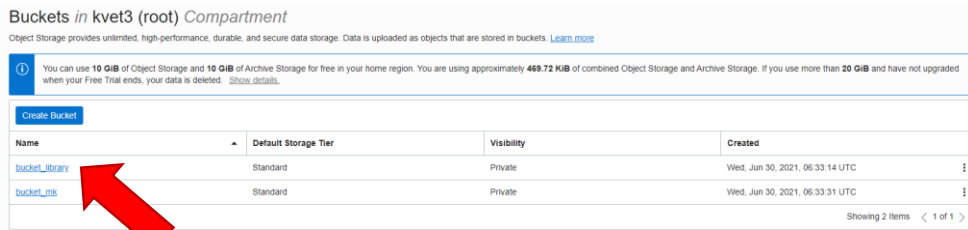


Fig. 6.16: Created bucket

By clicking on the *bucket name*, its definition is present, followed by the parameters, availability, and list of objects stored there. For now, the bucket does not hold any data.

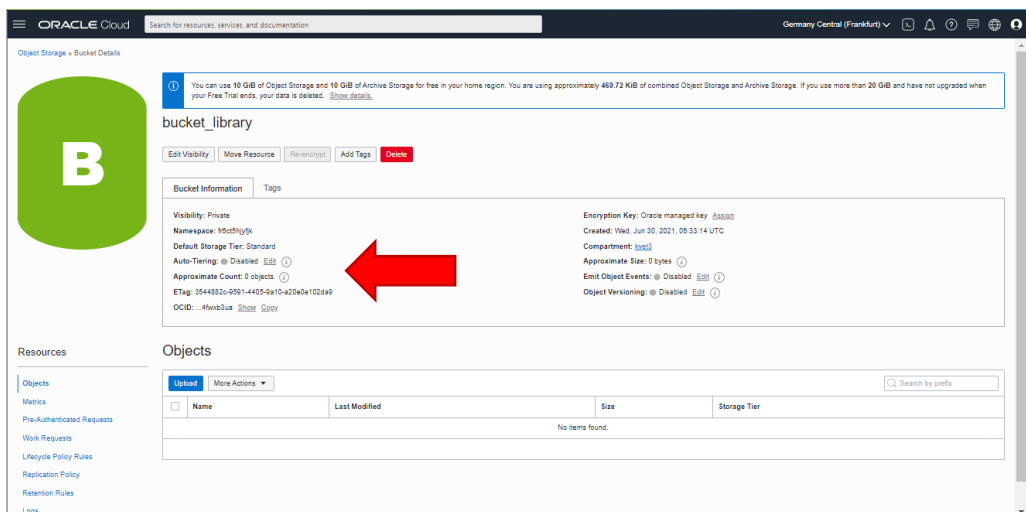


Fig. 6.17: Created bucket characteristics

Now, it is time to upload the exported dump file to the bucket (export file is available in the file repository of the book – *expdp_library.dmp*). Click on the *Upload* button under the *Object definition*.



Fig. 6.18: Object upload into the bucket

Object Prefix Name is optional and will be treated as the left-most part of the original object name extension. The original name is self-explanatory so that the input field can remain empty.

After upload operation, a particular file is available in the *Object list* of the *Bucket*, meaning that the file is (can be) accessible via the internet or various interfaces.



Fig. 6.19: Uploaded file

At this stage, the file has been uploaded into *Cloud Object storage*. However, it is not accessible from the database. It is even not accessible through the local *SQL Developer*. The solution is to create *Credentials* by invoking the *Create_credentials* procedure of the *Dbms_cloud* package.

Create_credentials procedure

Dbms_credentials is a package supervising the authentication process. It provides an interface for authenticating and impersonating *EXTPROC* callout functions and external or remote jobs and file watchers from the *Scheduler*.

Credentials are database objects holding *username* and *password* pairs. They are created by invoking the *Create_credentials* procedure of the defined package. The syntax of the method consists of seven parameters. The first three ones are mandatory. The others have default clauses.

```
DBMS_CREDENTIAL.CREATE_CREDENTIAL (
  credential_name  IN  VARCHAR2,
  username         IN  VARCHAR2,
  password         IN  VARCHAR2,
  database_role    IN  VARCHAR2  DEFAULT NULL,
  windows_domain   IN  VARCHAR2  DEFAULT NULL,
  comments         IN  VARCHAR2  DEFAULT NULL,
  enabled          IN  BOOLEAN    DEFAULT TRUE);
```

Credential_name is a unique name used for reference. It cannot be undefined (*NULL*) and is automatically converted to the uppercase unless specified in the double quotes ("). *Username* is a definition of the connection to the cloud database – tenancy. *Password* is provided by the *authentication token*. *Database_role* parameter delimits the administration privileges (*SYSDBA*, *SYSDBG*, *SYSADMIN*, or *SYSBACKUP*). By default, the connection is made via standard user privileges delimited by the *NULL* value. *Enabled* parameter defaults to *True*, limiting the availability of the *Credentials*.

The following code expresses the required parts. *Credential_name* is a unique, *username* represents *tenancy name* and *password* is delimited by the *authentication token* (do not execute the code now, it is just an example of the structure):

```
BEGIN
  DBMS_CLOUD.CREATE_CREDENTIAL(
    credential_name => 'CREDENTIAL_NAME',
    username       => 'tenancy_name',
    password       => 'authentication_token');
END;
/
```

For the creation, we will need the *credential name* (use any you want, but without the spaces). Note that it must be unique among the *tenancy*. *Tenancy name* has been specified

and provided to you during the registration process to the cloud. Its value can be obtained by clicking on the profile inside the cloud.

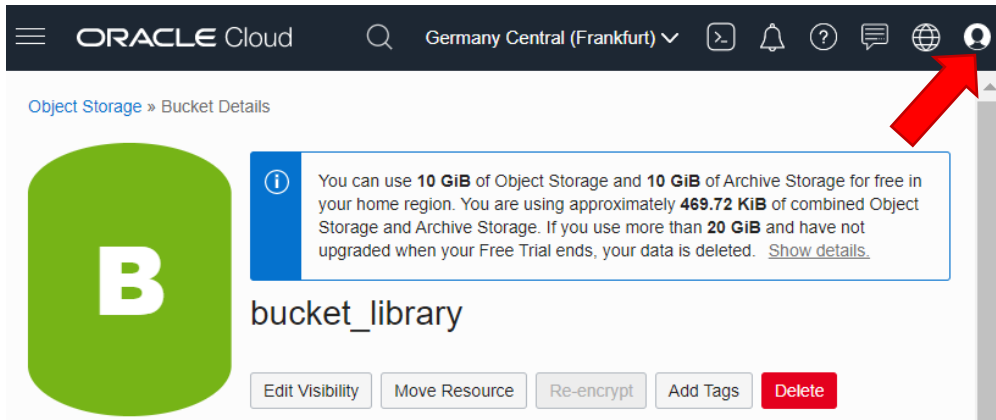


Fig. 6.20: Getting tenancy name (1)

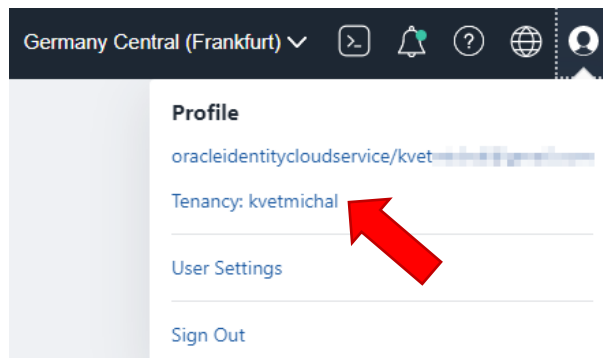


Fig. 6.21: Getting tenancy name (2)

In my case, the tenancy name is “*kvetmichal*”.

Authentication token

To define new *Credentials*, it is necessary to obtain an *Authentication token*. There are, in principle, two ways, how to reach them. The first solution is based on clicking the *Profile* and selecting username (in my case: `oracleidentitycloudservice/kvet***@*****.com`):

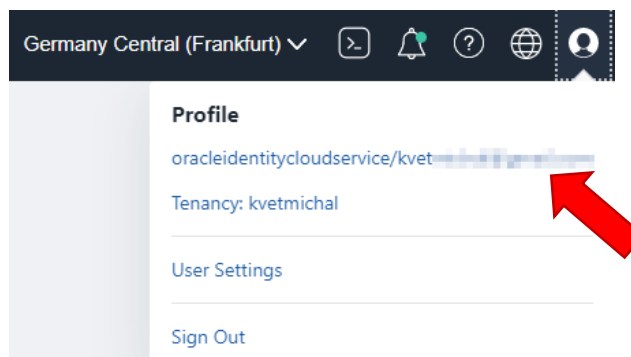


Fig. 6.22: Profile

The second solution is based on accessing it using the left navigation panel menu: *Identity & Security* => *Users*.

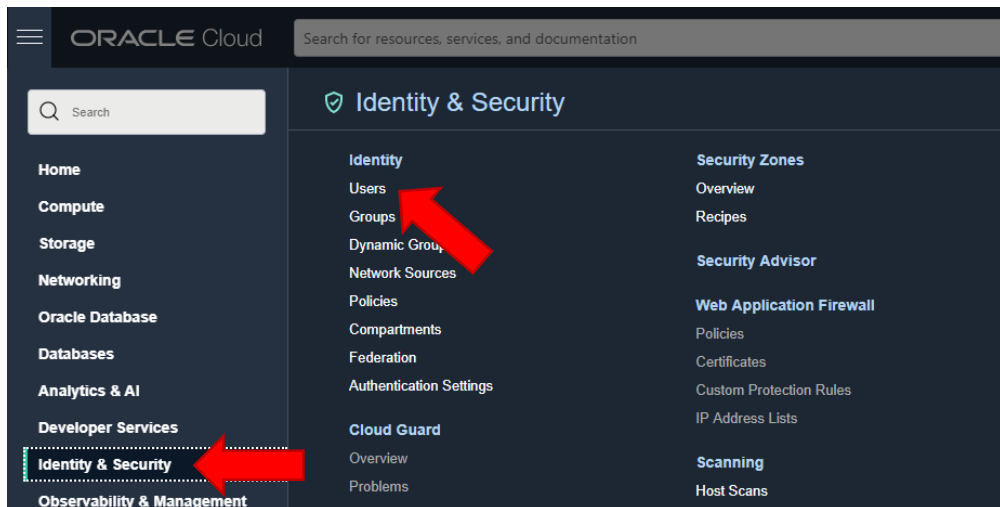


Fig. 6.23: User identity

By clicking on the user, available resources are listed in the left part of the screen.

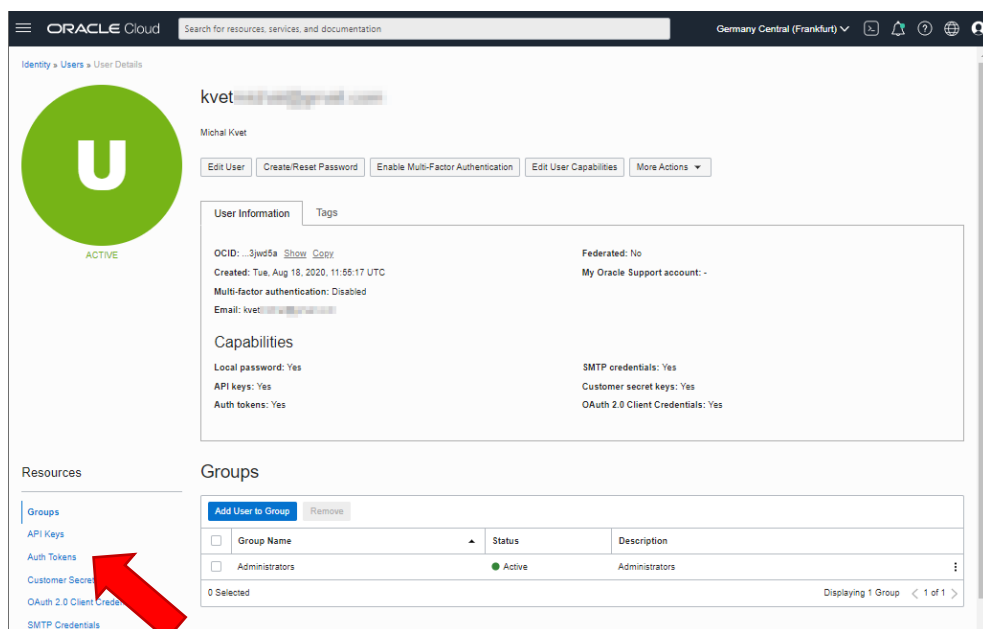


Fig. 6.24: Authentication token (1)

Select *Auth Tokens* and *Generate token*.

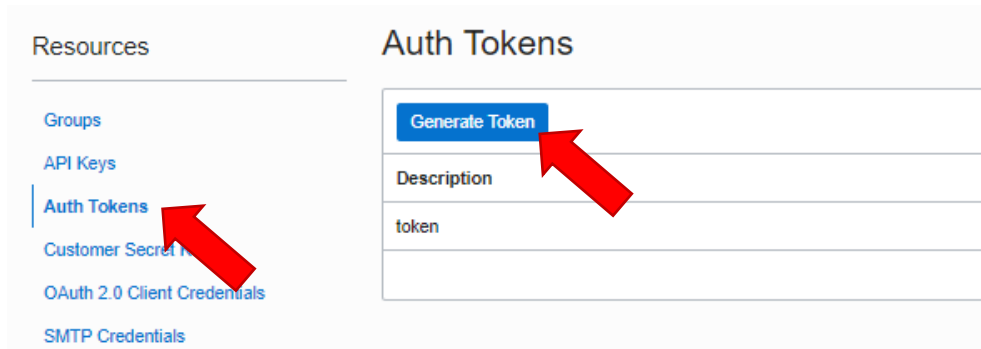


Fig. 6.25: Authentication token (2)

Provide some explanatory description.

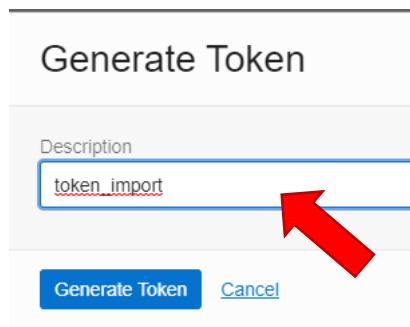


Fig. 6.26: Authentication token (3)

Copy the generated token to the clipboard. Note that it is impossible to get it afterward. It would be necessary to remove the token and create a new one.

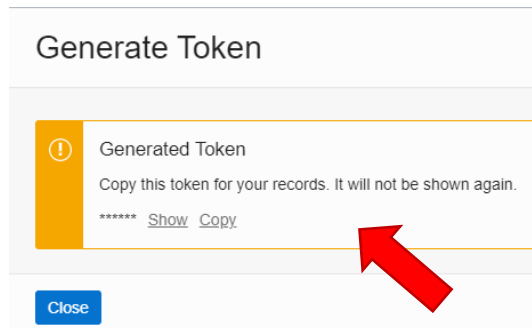


Fig. 6.27: Authentication token (4)

Now, you have all the required values to create *Credentials*, so let's return to the defined syntax. Connect to the cloud instance as the administrator user (*admin*) via *SQL Developer desktop*, *web*, or *Instant client*. Execute the following code. The process of getting a username and authentication tokens has already been specified.

```
BEGIN
DBMS_CLOUD.CREATE_CREDENTIAL(
  credential_name => 'ATP_CREDENTIAL_MK',
  username => 'kvetmichal',
  password => '*****'); -- replace the value with the generated value
END;
/
```

The result provided by *the SQL Developer Web*:

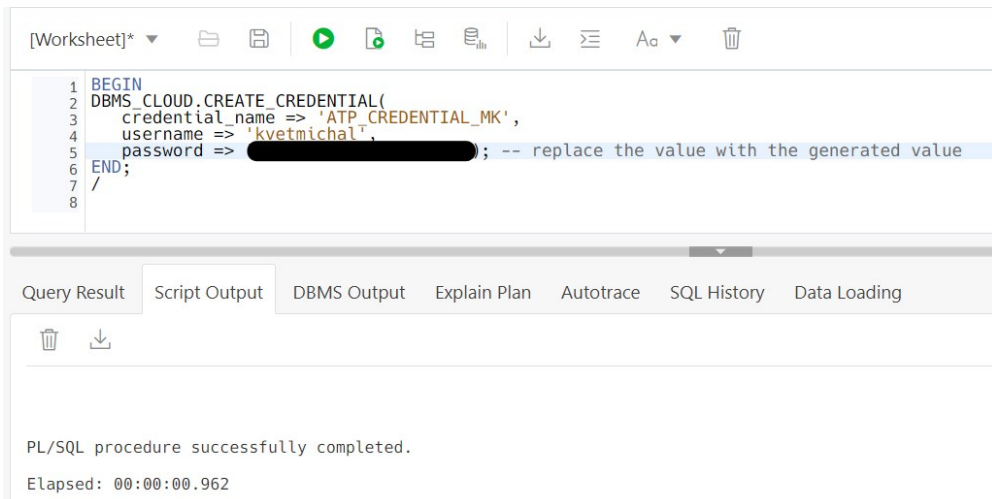


Fig. 6.28: Create credentials – result

Check the created *Credentials* by retrieving data from the *all_credentials* data dictionary:

```
SELECT credential_name, username
FROM all_credentials
ORDER BY credential_name;
```

	credential_name	username
1	ATP_CREDENTIAL_MK	kvetmichal

Fig. 6.29: List of created credentials

All prerequisites for the *data pump* operations are passed at this moment. Before we start, *create a new user account* to cover the data import. Grant him at least *connect*, *resource*, and *tablespace limit privileges* (it can be done by using *admin* user in the *SQL Developer* or *Instant client*).

```
create user library_user identified by *****;
grant connect, resource, unlimited tablespace to library_user;
```

Replace the “*****” with the actual password (at least 8 characters with upper and lower case and numeric value).

In the next part, import using the data dump (*impdp*) will be managed in *the SQL Developer desktop*. Navigate to the *View => DBA*.

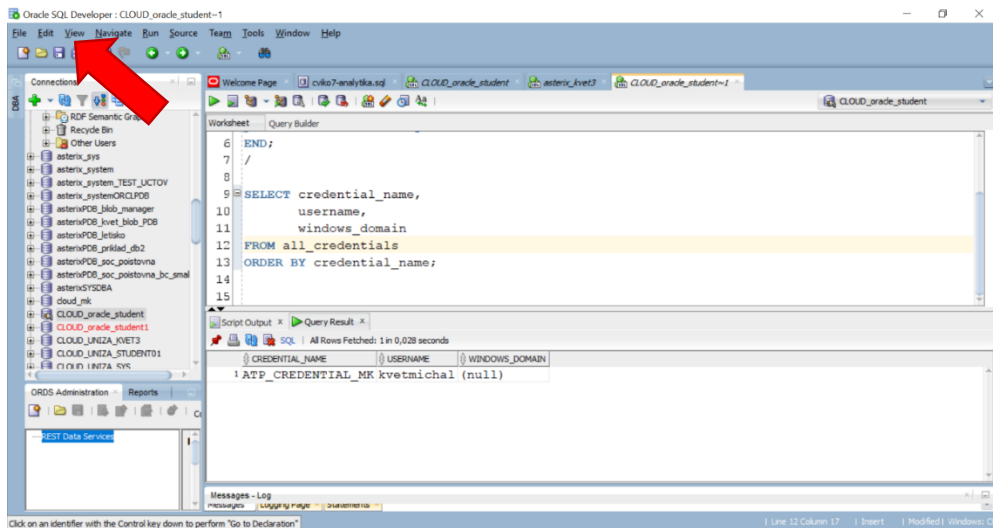


Fig. 6.30: Enabling DBA menu (1)

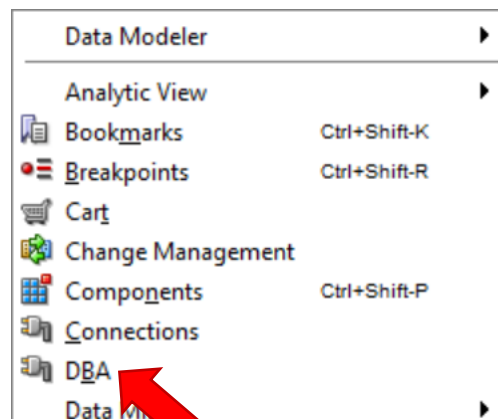


Fig. 6.31: Enabling DBA menu (2)

Specify a new *DBA* connection by clicking on the green plus symbol (+) in the *DBA* group:

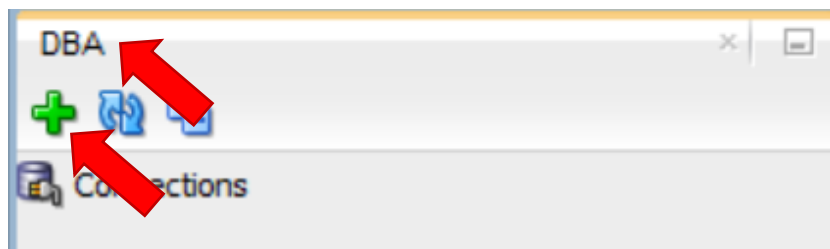


Fig. 6.32: DBA connection specification

The connection is delimited by the *admin* account to the cloud database. Let's expand it. It consists of several administration tools and performance monitoring. Navigate to the *Data Pump* section. Right-click on the *Data Pump*. There are two options – wizard for the *Data Pump Import* and *Export*.

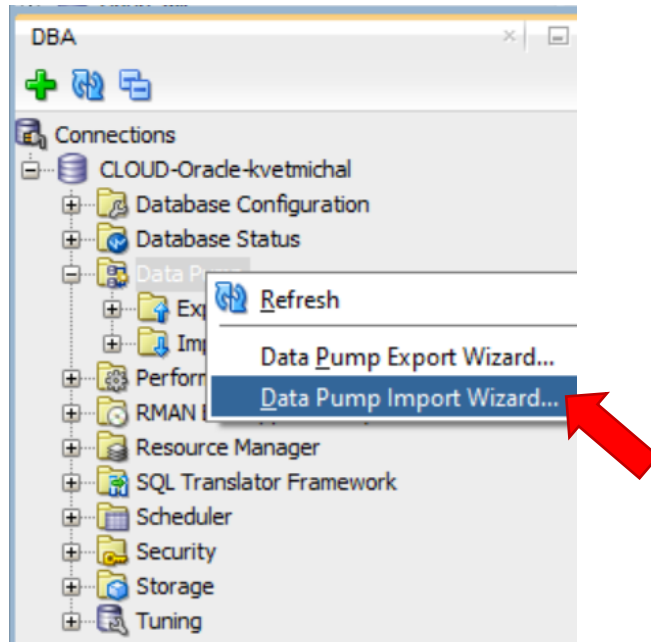


Fig. 6.33: Data Pump import

Data Pump Import Wizard

Impdp Wizard consists of the six steps to be treated. All activity is done on the *server*-side; a client is just treated as the supervisor process.

Namely, specify the *Job Name* by which the process can be monitored. *Type of objects* to be imported – either the structure itself or extended by the data loading into the tables. We will load the *table structure (DDL)* and the data, so the option “*Data and DDL*” will be selected. The type of import can deal with various granularity levels. We will use table precision reflection. The input source demands are credential definition and link to the data file. Credentials have been created using in the previous step by using the *Create_credentials* procedure:

```
BEGIN
DBMS_CLOUD.CREATE_CREDENTIAL(
  credential_name => 'ATP_CREDENTIAL_MK',
  username => 'kvetmichal',
  password => '*****'); -- replace the value with the generated value
END;
/
```

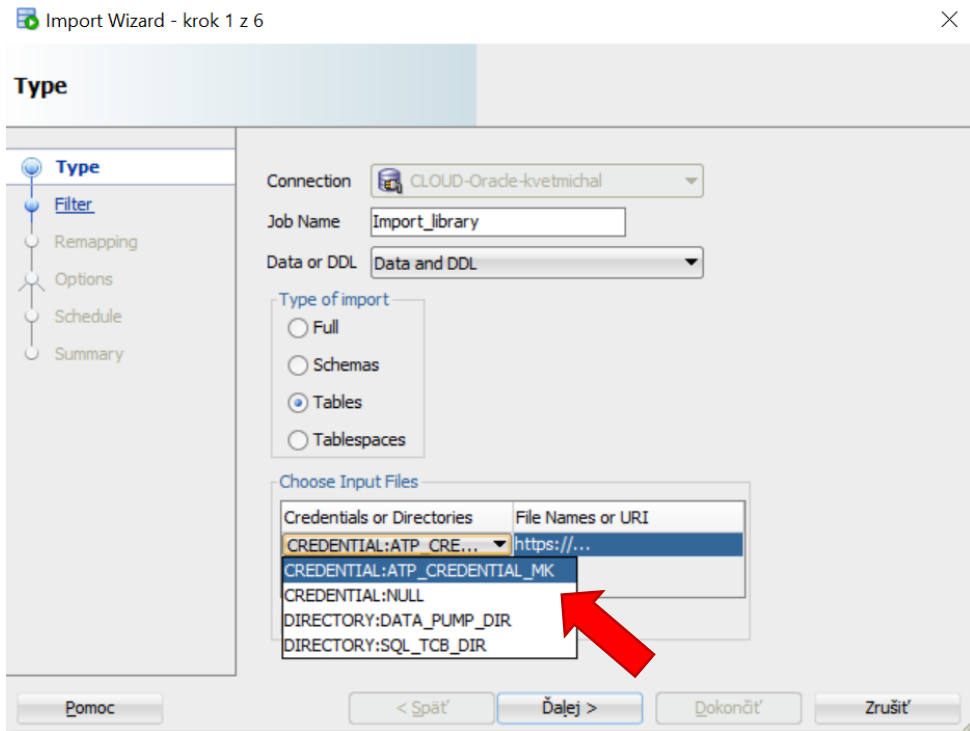


Fig. 6.34: Data Pump import wizard (1)

We still need the path to the import file, which is stored in the *Object storage* of the cloud. It is done by the *pre-authenticated request* defined in the Oracle Cloud console.

Return to the cloud, navigate to the *Object storage*, select the *Bucket* and relevant file inside (scroll down to the *Objects* section).

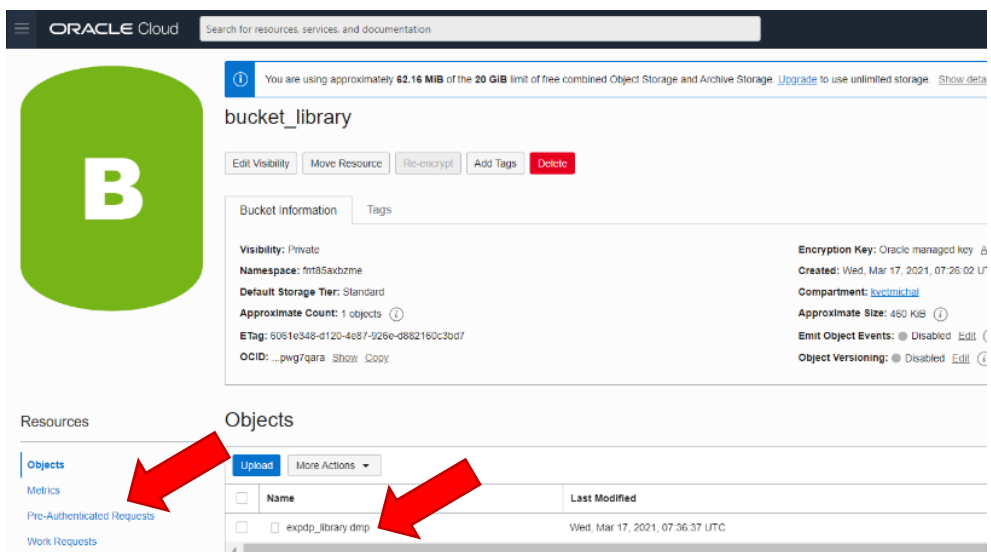


Fig. 6.35: Pre-authenticated request definition

In the *Resources* list of the left part of the screen – *Pre-Authenticated Requests* are present. Click there and create a new element. Specify the name and target – either the whole *bucket* or the *object* itself. Specify the *access privileges* and *expiration*, as well.

Create Pre-Authenticated Request [Help](#)

Name
preauth_bucket_library

Pre-Authenticated Request Target

Bucket
 Create a pre-authenticated request that applies to all objects in the bucket. ✓

Object
 Create a pre-authenticated request that applies to a specific object.

Objects with prefix
 Create a pre-authenticated request that applies to all objects with a specific prefix.

Access Type

☒ Permit object reads
☐ Permit object writes
☐ Permit object reads and writes

☐ Enable Object Listing
 Let users list the objects in the bucket.

Expiration
Jul 7, 2021 07:16 UTC

Create Pre-Authenticated Request Cancel

Fig. 6.36: Create a pre-authenticated request for the whole bucket

Copy the provided URL address. Note that it will not be visible later.

Pre-Authenticated Request Details

Name Read-Only
preauth_bucket_library

Pre-Authenticated Request URL Read-Only
https://objectstorage.eu-frankfurt-1.oraclecloud.com/p/Z-b3JdtYUS1Tbb3SVBsaX3Hl

⚠ Copy this URL for your records. It will not be shown again.

Close

Fig. 6.37: Pre-authenticated request result

We have created a *pre-authentication request* for the whole *bucket*.

Similarly, the *pre-authentication request* can be created just for the *individual object* (*file*), as well. In that case, it is done either by selecting *Object type* in the definition. The name of the object inside the *bucket* needs to be specified explicitly, as well as *access rules* (*read*, *write* privileges).

Create Pre-Authenticated Request [Help](#)

Name
preauth_export_library

Pre-Authenticated Request Target

Bucket

Create a pre-authenticated request that applies to all objects in the bucket.

Object

Create a pre-authenticated request that applies to a specific object. ✓

Objects with prefix

Create a pre-authenticated request that applies to all objects with a specific prefix.

Object Name
expdp_library.dmp

Access Type

☒ Permit object reads

☐ Permit object writes

☐ Permit object reads and writes

Expiration
Jul 7, 2021 07:20 UTC

[Create Pre-Authenticated Request](#) [Cancel](#)

Fig. 6.38: Pre-authenticated request for the specific object (1)

A more straightforward solution can be reached by defining pre-authentication request directly for the file in the *bucket* object list. Navigate to the *Object storage*, select the relevant *bucket* by which the objects inside will be listed.

ORACLE Cloud Search for resources, services, and documentation Germany Central (Frankfurt)

bucket_library

Edit Visibility Move Resource Re-encrypt Add Tags Delete

Bucket Information Tags

Visibility: Private
Namespace: tr6ct5hlyrk
Default Storage Tier: Standard
Auto-Tiering: @ Disabled Edit
Approximate Count: 1 objects
ETag: 3544882c-9591-4405-9a10-a20e0e102da9
OCID: ...4fwb3ua Show Copy

Encryption Key: Oracle managed key Assign
Created: Wed, Jun 30, 2021, 06:33:14 UTC
Compartment: lvet
Approximate Size: 124 KB
Emit Object Events: @ Disabled Edit
Object Versioning: @ Disabled Edit

Resources

Objects

Upload More Actions Search by prefix

	Name	Last Modified	Size	Storage Tier
<input type="checkbox"/>	expdp_library.dmp	Wed, Jun 30, 2021, 06:39:16 UTC	124 KB	Standard

Fig. 6.39: Pre-authenticated request for the specific object (2)

Click on the *three dots* at the end of the file property list and select *Create Pre-authenticated Request*.

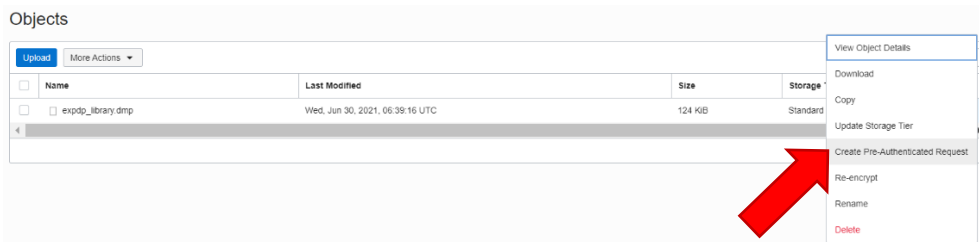


Fig. 6.40: Pre-authenticated request for the specific object (3)

In that case, the *Object name* will be filled automatically.

Create Pre-Authenticated Request [Help](#)

Name
par-object-20210630-0922

Pre-Authenticated Request Target

Bucket
Create a pre-authenticated request that applies to all objects in the bucket.

Object
Create a pre-authenticated request that applies to a specific object. ✓

Objects with prefix
Create a pre-authenticated request that applies to all objects with a specific prefix.

Object Name
expdp_library.dmp

Access Type
☒ Permit object reads
☐ Permit object writes
☐ Permit object reads and writes

Expiration
Jul 7, 2021 07:22 UTC

[Create Pre-Authenticated Request](#) [Cancel](#)

Fig. 6.41: Pre-authenticated request for the specific object (4)

For now, *read privilege* is suitable.

Copy the link to the clipboard and return to the import wizard inside the *SQL Developer* and use the copied link to the object (not the whole bucket) to the *File Name* position.

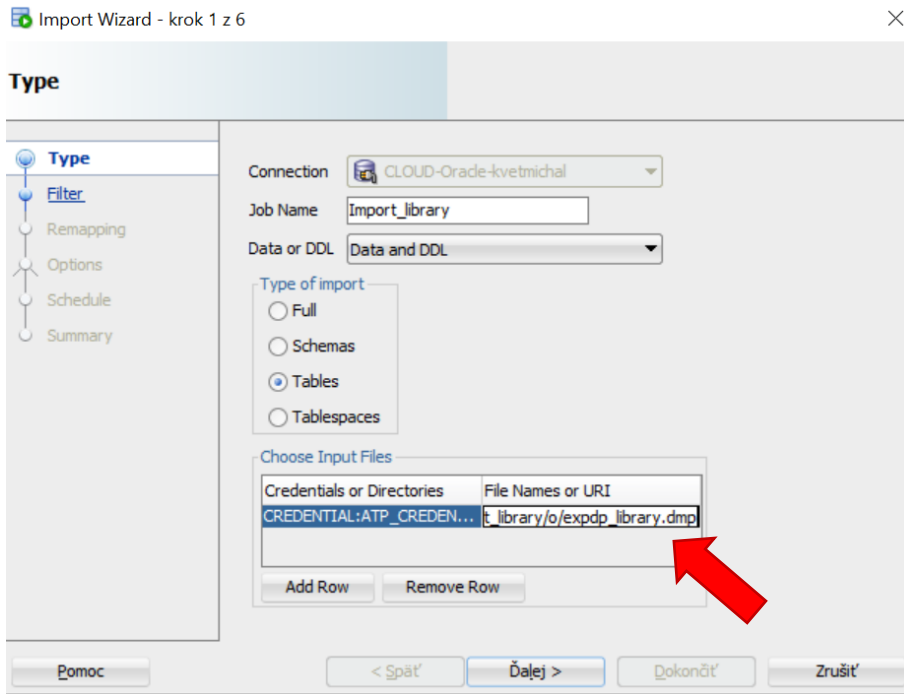


Fig. 6.42: Data Pump import wizard (1)

Navigate to the *Next* and wait to get the list of the available tables inside the *export dump file*. Select the *set of the tables* to be imported. In my case, I will import all of them.

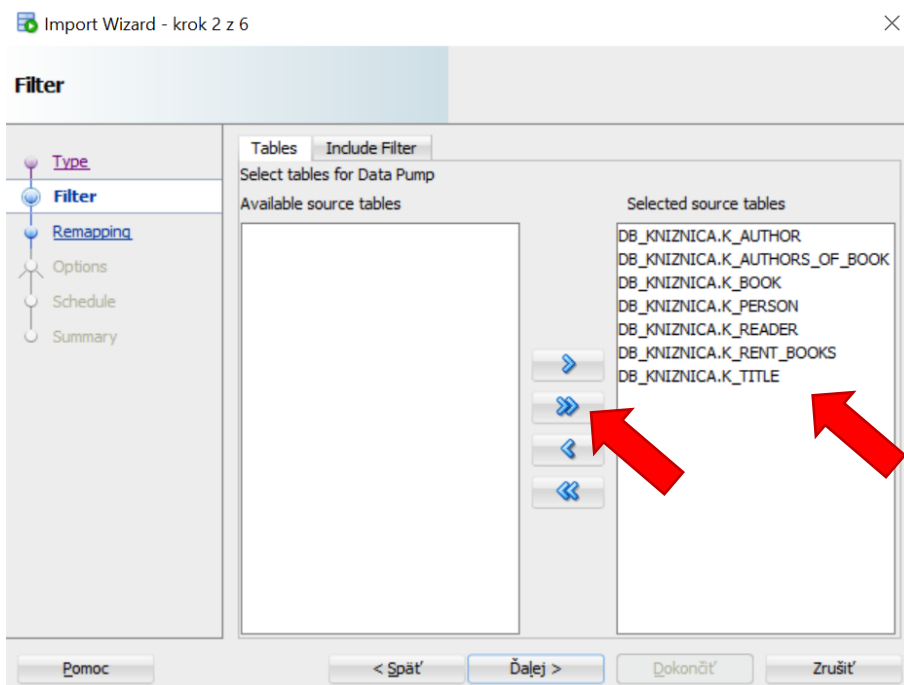


Fig. 6.43: Data Pump import wizard (2)

As stated, we have created one extra user as the import destination. His username is then specified in the third step – *Re-Map Schemas*. Source *username* will be obtained automatically, whereas the *export file* consists of only one user data. The *destination* user has been created. In my case, the username is “*library_user*”. *Tablespaces* can remain original. However, if necessary, they can be remapped similarly.

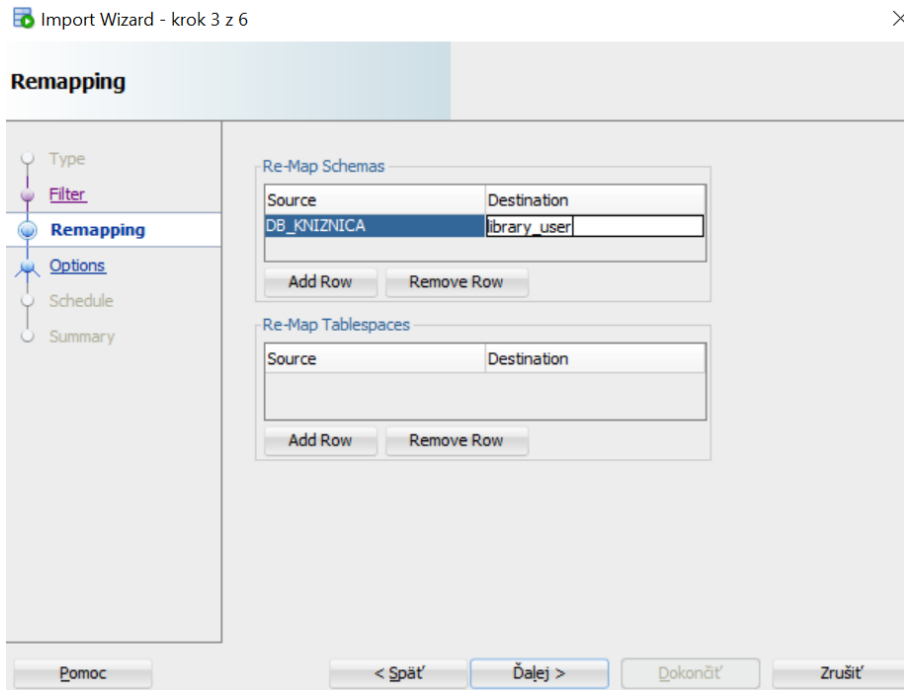


Fig. 6.44: Data Pump import wizard (3)

Step 4 defines the parameters of the import process – *number of threads*, *log file destination*, *action on the table*, if a particular table already exists, etc. For the definition, the *Logging* section is the most relevant.

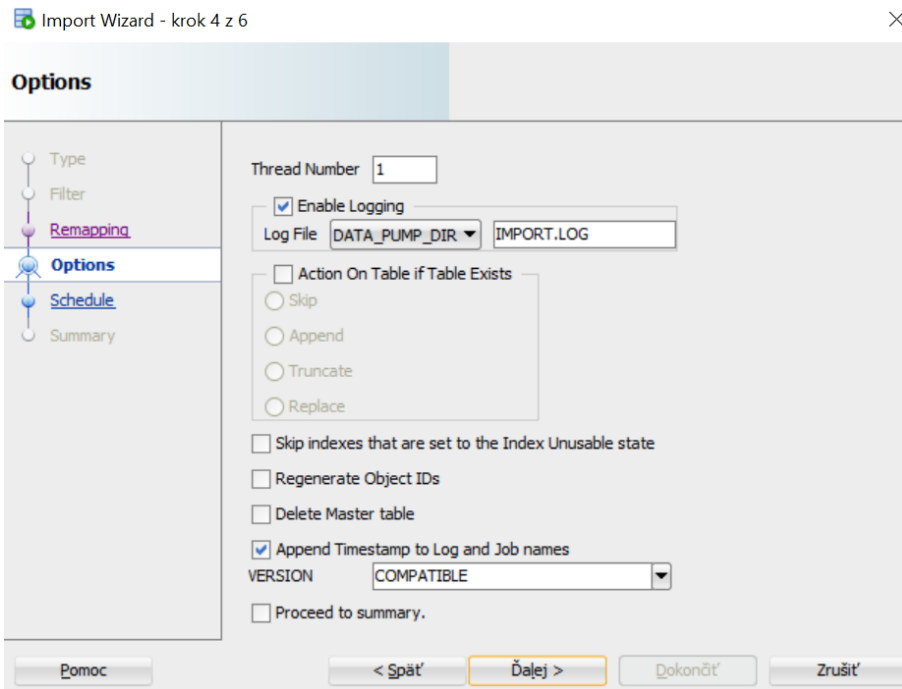


Fig. 6.45: Data Pump import wizard (4)

The name of the *Oracle directory mapper* is *DATA_PUMP_DIR*, which is always created for these *data pump* activities. The name of the log file is suitable, as well. However, it is required. So, name it whatever you want.

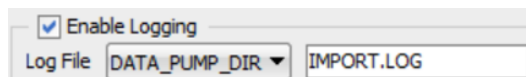


Fig. 6.46: Log output specification

Logging *import* and *export* activities (via *data pump*) is inevitable to identify the issues during the process, to get the information about the results, status, etc.

Optionally, you can define a job by the *time* when the *action will start*. Otherwise, it will be executed immediately.

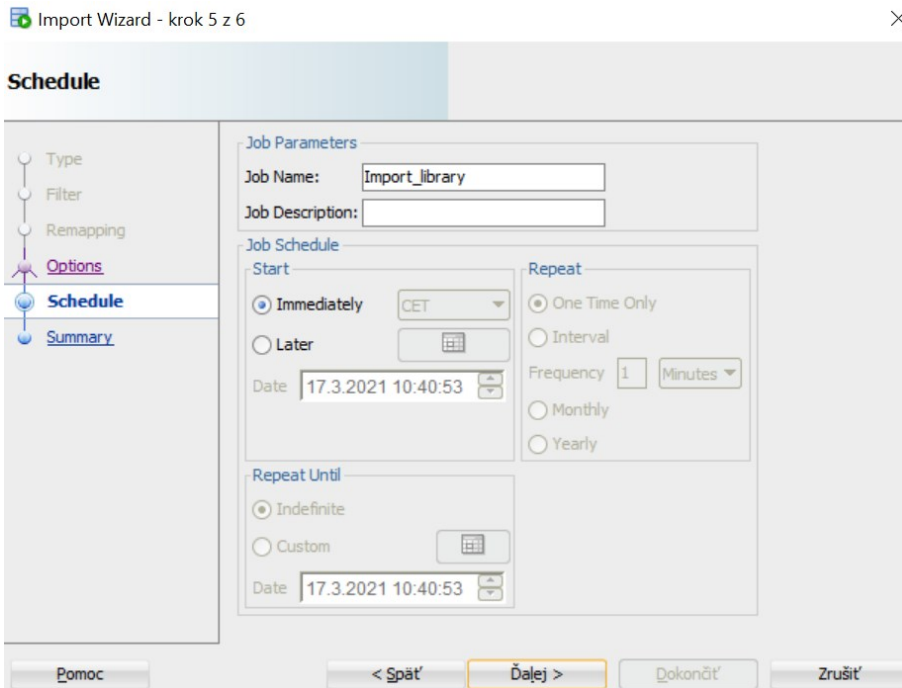


Fig. 6.47: Data Pump import wizard (5)

Proceed to the summary, check it, and start the process.

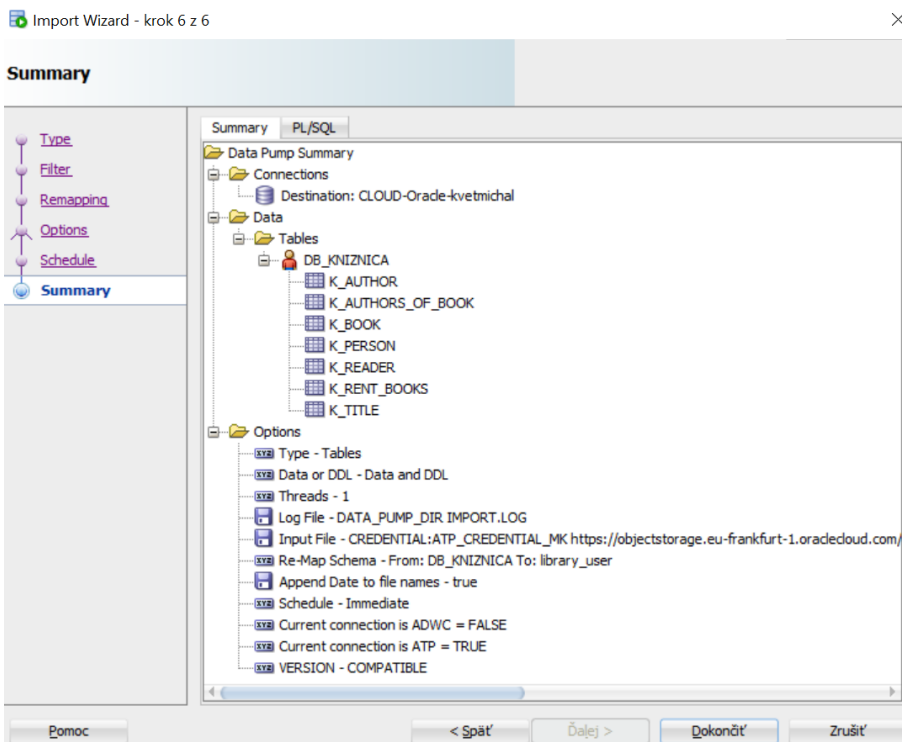


Fig. 6.48: Data Pump import wizard (6)

Now, the process is to be started.

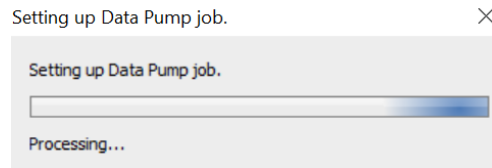


Fig. 6.49: Setting data import job

Execution can be monitored using the *SQL Developer*, as well.

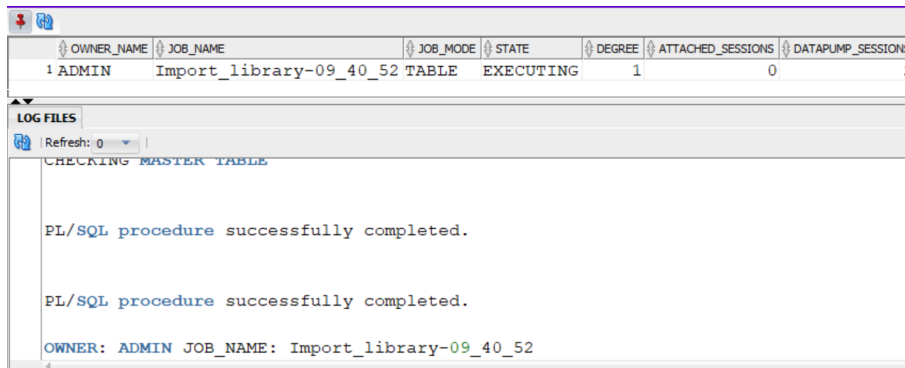


Fig. 6.50: Data Pump import execution monitoring

The import process has been done. *IMPDP*, as a *data pump import* process, has generated a *log file* describing the import process. But where to find it?

The *log file* is not accessible directly. It is available, located on the cloud by using the *Oracle directory* specified during the 4th step of the import process (*DATA_PUMP_DIR*). It is not part of the *bucket*, so the user cannot get it directly. Looking at the import process description of the *SQL Developer* desktop, the defined file name is specified:

OPENING: MK_EXP_KNIZNICA_DIR: expdp_kniznica.log

Fig. 6.51: Log file location

To get the log file accessible, it must be copied to the *bucket* of the *Object storage*. It can be ensured through the procedure *Put_object* of the *DBMS_CLOUD* package.

We will ensure this through a procedure *DBMS_CLOUD.PUT_OBJECT*. It copies a file from *Autonomous Database* to the *Cloud Object Storage*. The maximum allowed file size is 50 GB.

Syntax of the *Put_object* procedure:

```
DBMS_CLOUD.PUT_OBJECT (
  credential_name      IN VARCHAR2,
  object_uri           IN VARCHAR2,
  directory_name       IN VARCHAR2,
  file_name            IN VARCHAR2);
```

- *Credential_name* has already been created, covered by a definition of the *username* and *password* provided by the *authentication request* procedure to the *Object Storage*.
- *Object_uri* represents the *URL link* to the *bucket* or *object* itself.

- *Directory_name* is an existing *Oracle directory* present in the *Autonomous Database*.
- *File_name* – the *name of the file* located and accessible via a defined *Oracle directory* (*directory_name* specification). It represents the *log file* of the *data pump import*, and its name was specified either during the import definition in the wizard or is obtainable using *SQL Developer* (as already stated).

Let's define the *Put_object* parameters and execute it. Let's take emphasis on the *Object_uri* parameter value.

We have created two *pre-authenticated requests*, one for the whole *bucket*, and the second is associated with the *object* itself. Let's evaluate their structures. *N* references namespace, *B* covers the bucket, and *O* expresses the object.

Bucket

```
https://objectstorage.eu-frankfurt-1.oraclecloud.com
/p/O8mTAKHDvT3qypjLlazUDoylyt-KOemYlcQdl9DhpcMiRS6BzM68vSd5EAi3OCd7
/n/frrt85axbzme
/b/bucket_library
/o/
```

Object

```
https://objectstorage.eu-frankfurt-1.oraclecloud.com
/p/j4Ub7Rtqeb9ElqRaVdPsQooOPJ8XqUpAim-eiXsMhUPN2ziSDv1JAhcsm7Zv5Gq
/n/frrt85axbzme
/b/bucket_library
/o/expdp_library.dmp
```

The *URL address* always consists of the *cloud address*, reflection to the *Object storage*, *Bucket*, and optionally *object (file) name*.

So, if you use the whole *bucket* as the *object_uri* parameter, it must be extended by the name of the destination file:

```
https://objectstorage.eu-frankfurt-1.oraclecloud.com
/p/O8mTAKHDvT3qypjLlazUDoylyt-KOemYlcQdl9DhpcMiRS6BzM68vSd5EAi3OCd7
/n/frrt85axbzme
/b/bucket_library
/o/IMPORT_DP_library.LOG
```

The definition of the *Put_object* procedure can look like following (replace the values with your defined structures and links):

```
BEGIN
  DBMS_CLOUD.PUT_OBJECT (
    credential_name => 'ATP_CREDENTIAL_MK',
    object_uri => 'https://objectstorage.eu-frankfurt-
                  1.oraclecloud.com/p/O8mTAKHDvT3qypjLlazUDoylyt-
                  KOemYlcQdl9DhpcMiRS6BzM68vSd5EAi3OCd7
                  /n/frrt85axbzme/b/bucket_library
                  /o/IMPORT_DP_library.LOG',
    directory_name => 'DATA_PUMP_DIR',
    file_name => 'IMPORT-10_31_28.LOG');
END;
/
```

Look at the *Cloud repository*, navigate to the *Object storage* and particular *bucket*. Now the file is visible there.

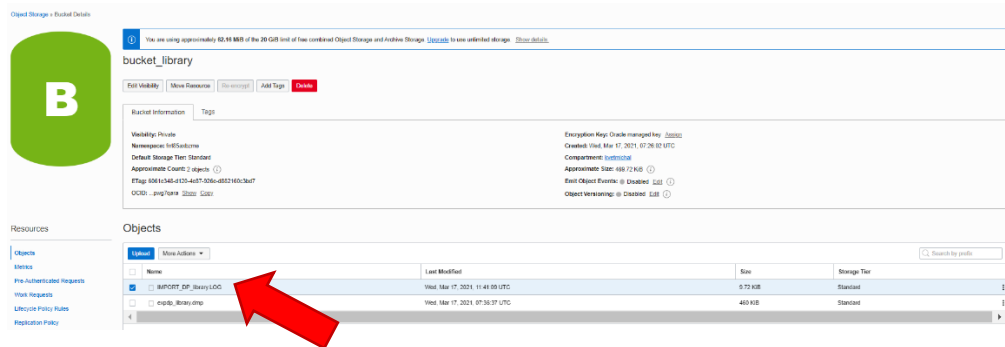


Fig. 6.52: Access to the log via Object storage

You can download it locally:

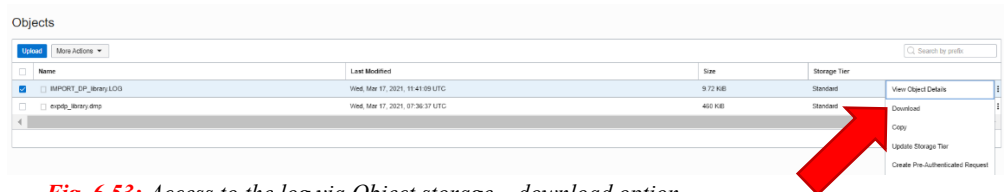


Fig. 6.53: Access to the log via Object storage – download option

The log consists of the summary of the activity, as well as the error description. The first part deals with the total number of imported rows to each table.

```
Processing object type SCHEMA_EXPORT/PRE_SCHEMA/PROCACT_SCHEMA
Processing object type SCHEMA_EXPORT/TABLE/TABLE
Processing object type SCHEMA_EXPORT/TABLE/TABLE_DATA
. . imported "KNIZNICA_ENG"."K_RENT_BOOKS"          40.46 KB    1000 rows
. . imported "KNIZNICA_ENG"."K_BOOK"                 21.75 KB     500 rows
. . imported "KNIZNICA_ENG"."K_PERSON"                15.58 KB     100 rows
. . imported "KNIZNICA_ENG"."K_TITLE"                 12.38 KB     100 rows
. . imported "KNIZNICA_ENG"."K_READER"                11.01 KB     150 rows
. . imported "KNIZNICA_ENG"."K_AUTHOR"                 8.062 KB      50 rows
. . imported "KNIZNICA_ENG"."K_AUTHORS_OF_BOOK"       6.593 KB      50 rows
Processing object type SCHEMA_EXPORT/TABLE/GRANT/OWNER_GRANT/OBJECT_GRANT
Processing object type SCHEMA_EXPORT/TABLE/CONSTRAINT/CONSTRAINT
Processing object type
SCHEMA_EXPORT/TABLE/INDEX/STATISTICS/INDEX_STATISTICS
Processing object type SCHEMA_EXPORT/TABLE/CONSTRAINT/REF_CONSTRAINT
Processing object type SCHEMA_EXPORT/TABLE/STATISTICS/TABLE_STATISTICS
Processing object type SCHEMA_EXPORT/STATISTICS/MARKER
Job "KNIZNICA_ENG"."SYS_IMPORT_FULL_01" successfully completed at Mon Mar
21 14:21:11 2022 elapsed 0 00:00:35
```

Fig. 6.54: Log file content

Note that if any error occurs, it is necessary to analyze and evaluate it. In my case, there was a problem with the statistics. However, it can be ignored – the system will calculate new statistics on demand.


```
ORA-39083: Object type INDEX_STATISTICS failed to create with error:  
ORA-01403: no data found  
ORA-01403: no data found
```

Fig. 6.55: Errors inside the log

6.4.2 ExpDp

ExpDp is a new, more flexible, and faster *server-side* alternative to the “*exp*”. *ExpDp* functionality can be executed on various levels – either for the whole *database*, *schema* (*user*), or specific *tables*.

The process of the export is analogous to the *impdp* already described. In the DBA section, expand the connection and use the option *Data Pump Export Wizard* of the *Data Pump* element.

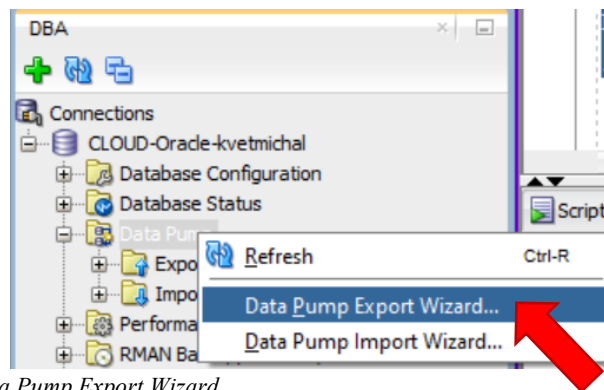


Fig. 6.56: Data Pump Export Wizard

Note, that for explanatory reasons, I have created one extra user called “*student_user*” consisting of the student model by using older *imp* version (launched in *Instant client*):

```
host imp student_user@studentdb_high  
fromuser=kvet1 touser=student_user  
file=exp_student.exp
```

Data pump export (ExpDp) is delimited by the eight-stage process. First of all, the result set structure is defined, consisting of either the *data structure definitions (DDL)* or data that can be present in the output, as well.

Exported *Data Pump* granularity can be the whole *database*, *tablespace*, *schema*, or *table*. In this example, we will export all tables of the created user *student_user*:

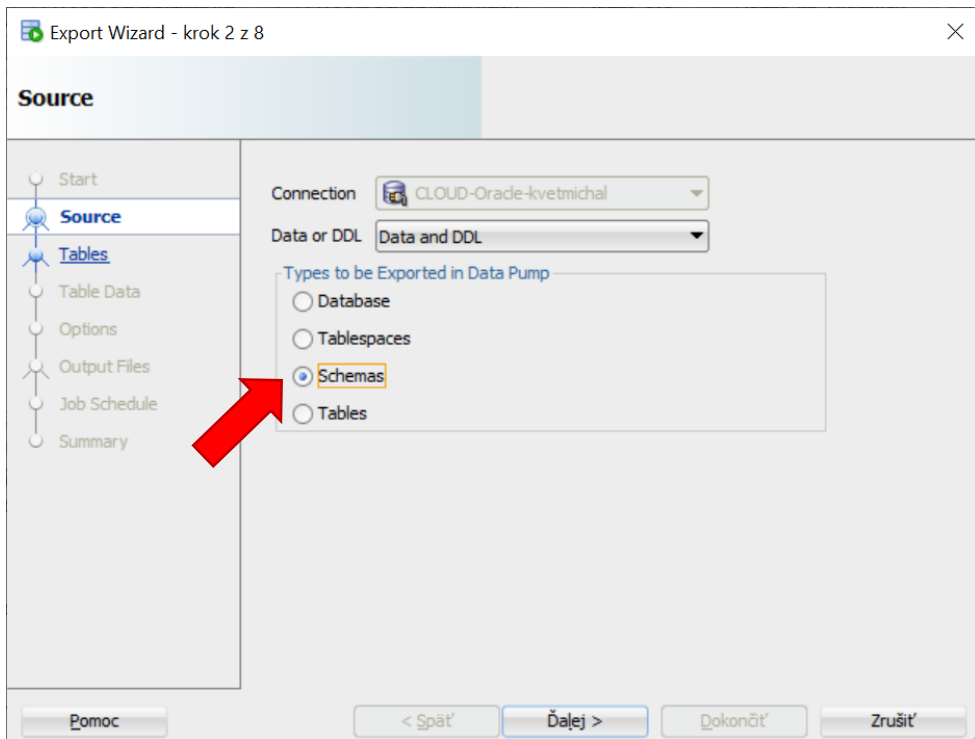


Fig. 6.57: Data Pump Export – step 2

In the third step, a list of usernames is present, which can be filtered out. Select the *student_user* and move it to the right part (*Selected source schemas*).

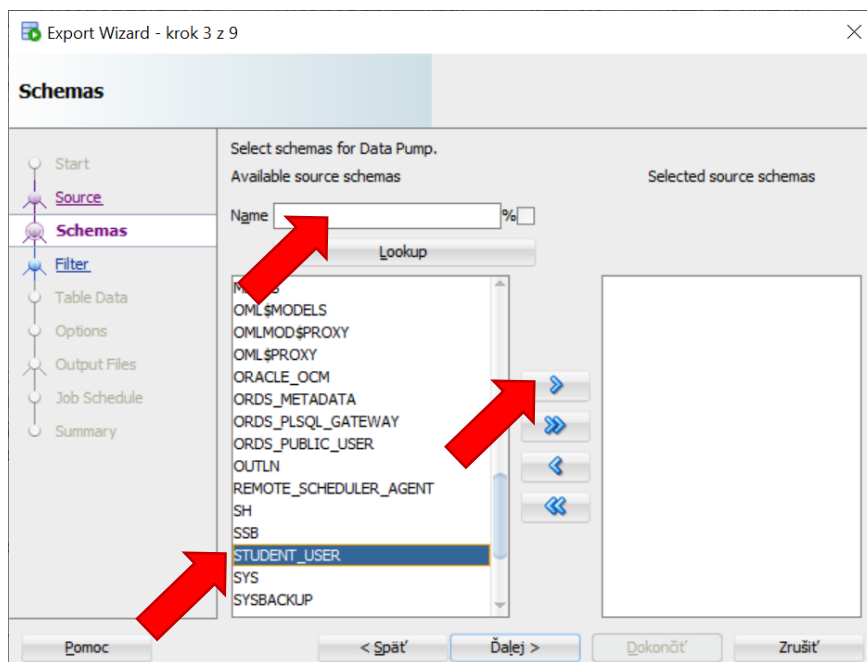


Fig. 6.58: Data Pump Export – step 3 (1)

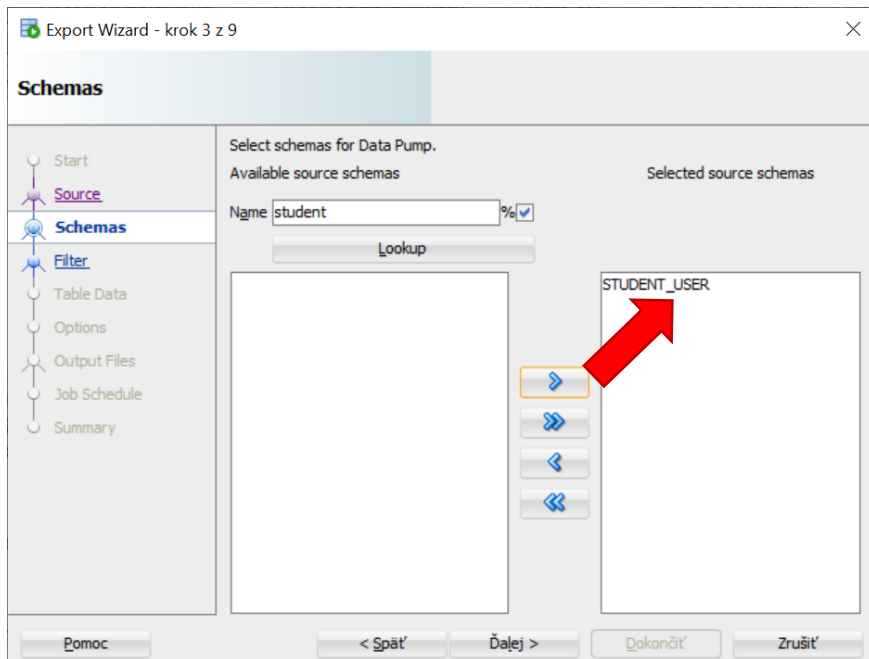


Fig. 6.59: Data Pump Export – step 3 (2)

In the next phase, filters can be optionally applied. We will do not use any filtering option.

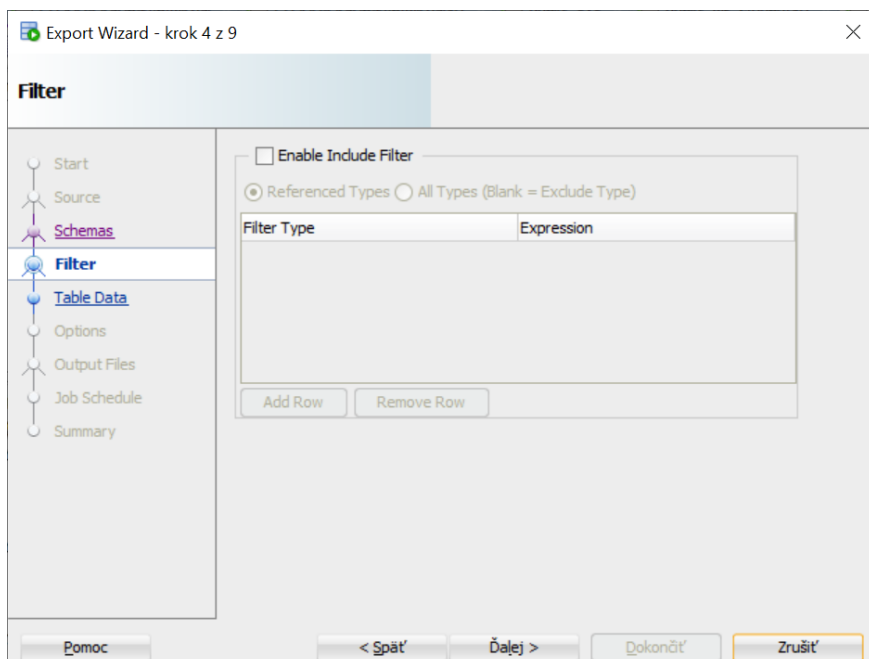


Fig. 6.60: Data Pump Export – step 4

Now, click on the *Lookup* button, by which the defined schema will be analyzed, list of tables will be loaded.

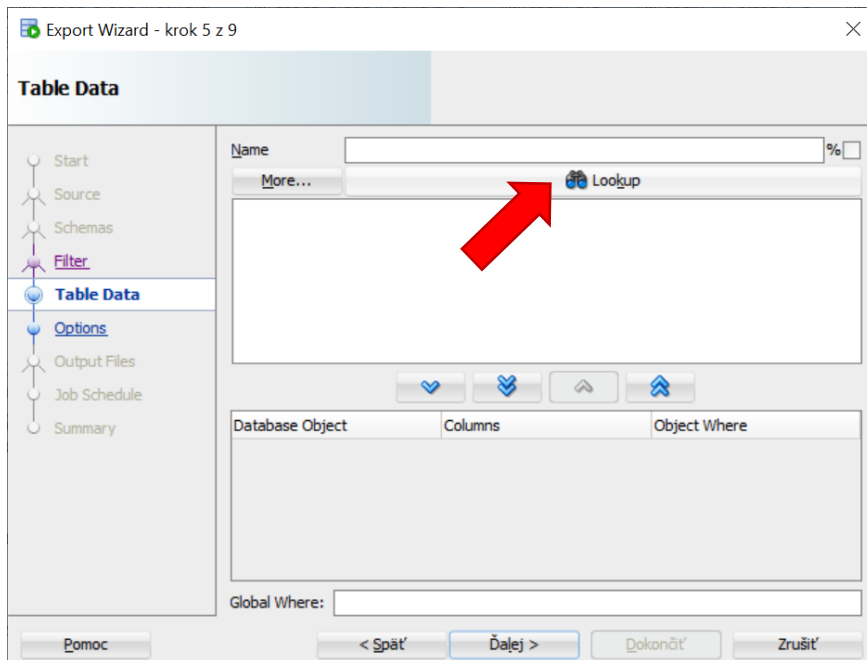


Fig. 6.61: Data Pump Export – step 5 (1)

Copy tables, which should be exported to the above list. I will export all the tables.

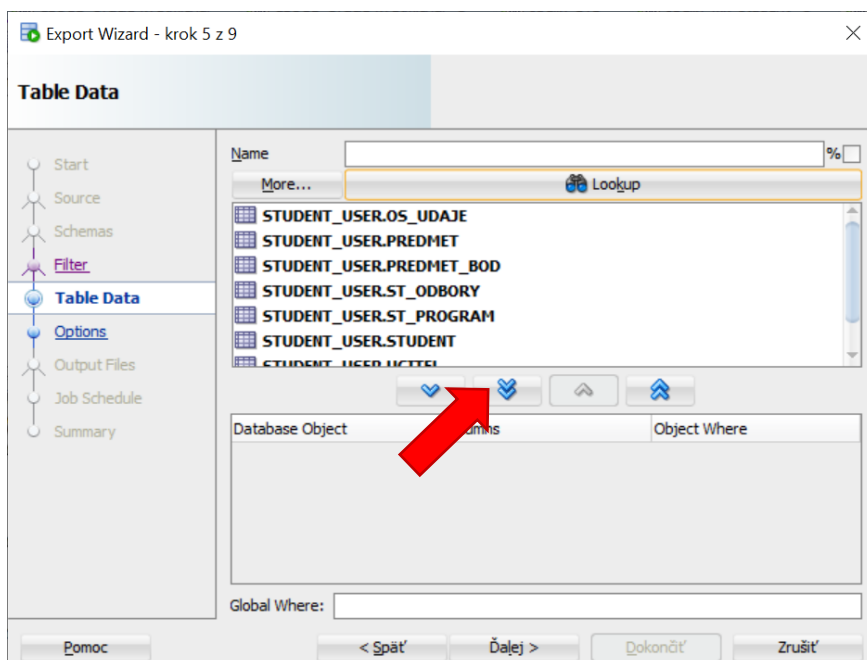


Fig. 6.62: Data Pump Export – step 5 (2)

Then, the options are defined. I recommend forcing the system to create a *log file* covering the data pump export process. It is maintained by the *DATA_PUMP_DIR* Oracle directory. Specify the name of the log file created.

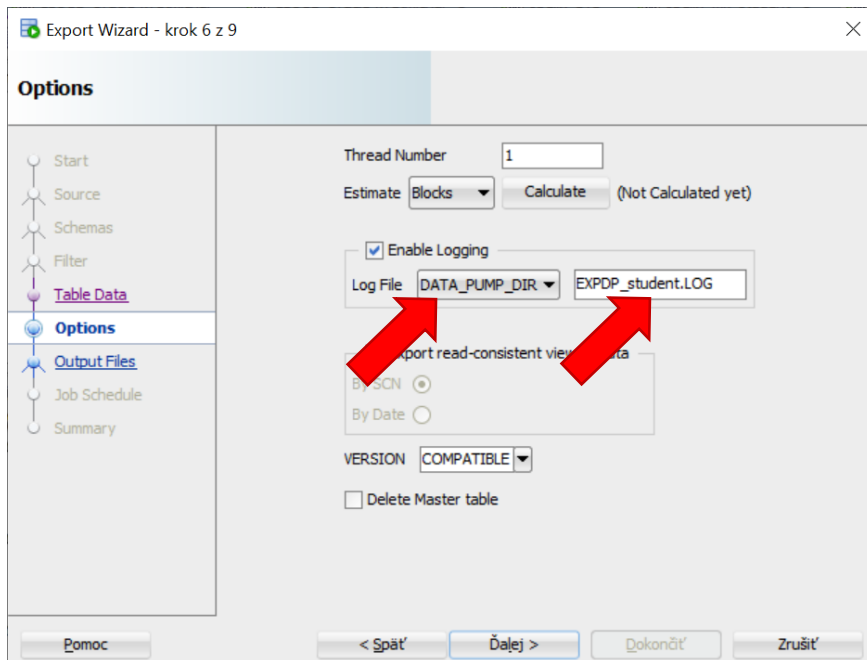


Fig. 6.63: Data Pump Export – step 6

In the *Output Files* step, result file names and other options influencing the compression, copy processes, etc., are specified. Output files will be part of the *DATA_PUMP_DIR* Oracle directory. Various parameters and flags can enhance the names for simplicity. For example, let's name the file – *EXPDP_LIBRARY.DMP*. Remove the check on the *Append Timestamp to Dump* (if checked, the file names will be enhanced by the timepoint of the execution).

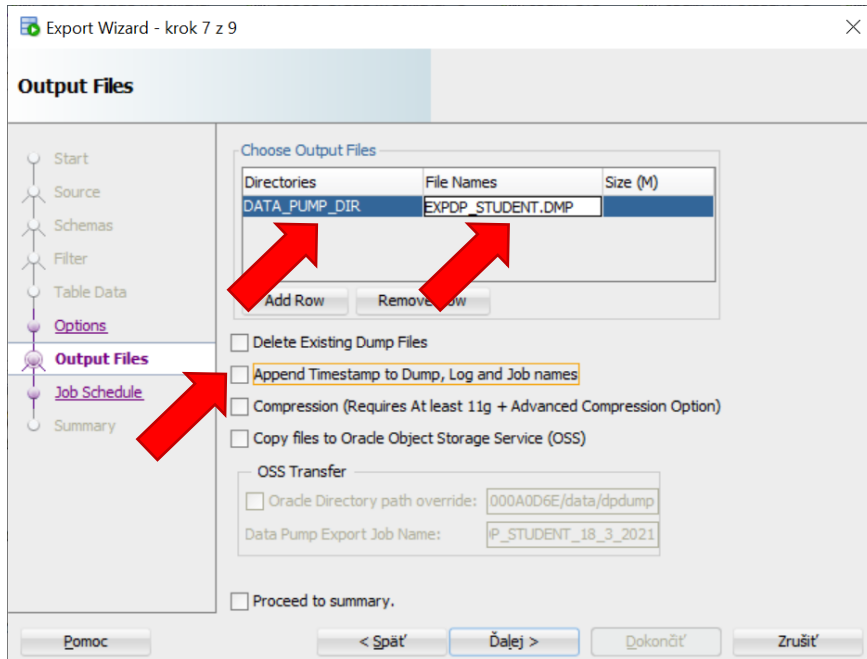


Fig. 6.64: Data Pump Export – step 7

Finally, *name the job* and optionally specify the description as well. *Job time planning* can be done in this phase, or the export process can be launched immediately.

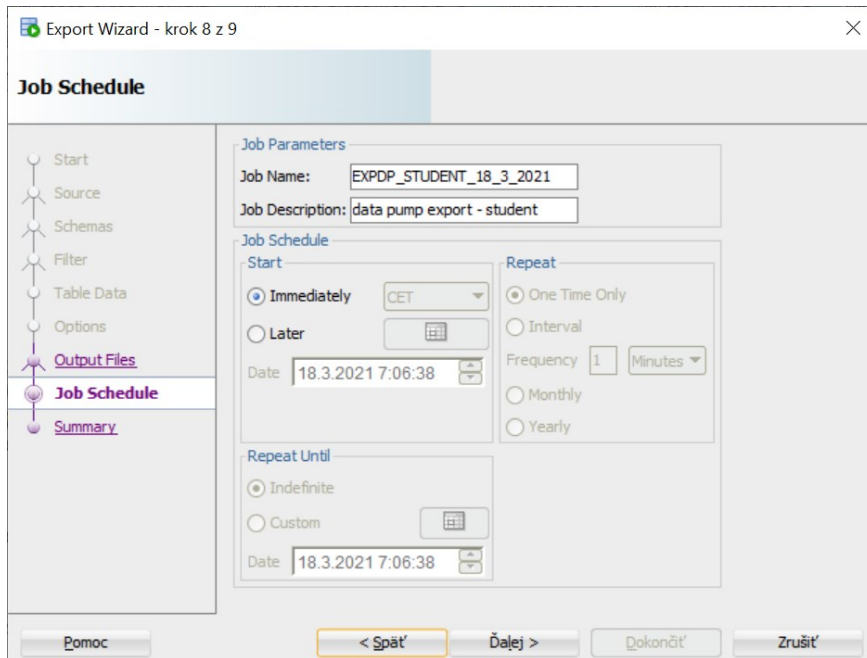


Fig. 6.65: Data Pump Export – step 8

Proceed with the summary, check the correctness and finish the definition by launching (or planning) the *expdp* process.

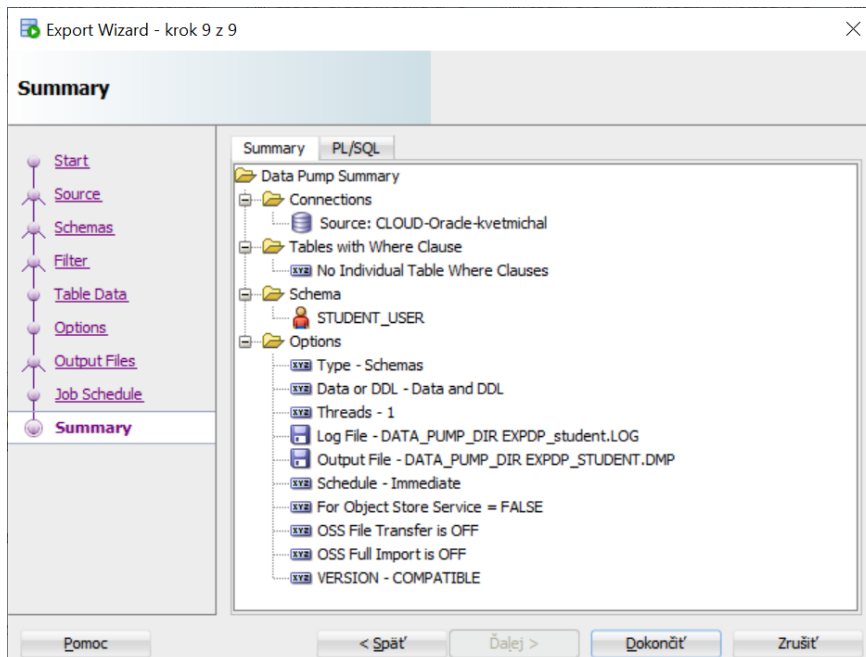


Fig. 6.66: Data Pump Export – step 9

Note that the generated script to be executed in the *PL/SQL* tab is visible.

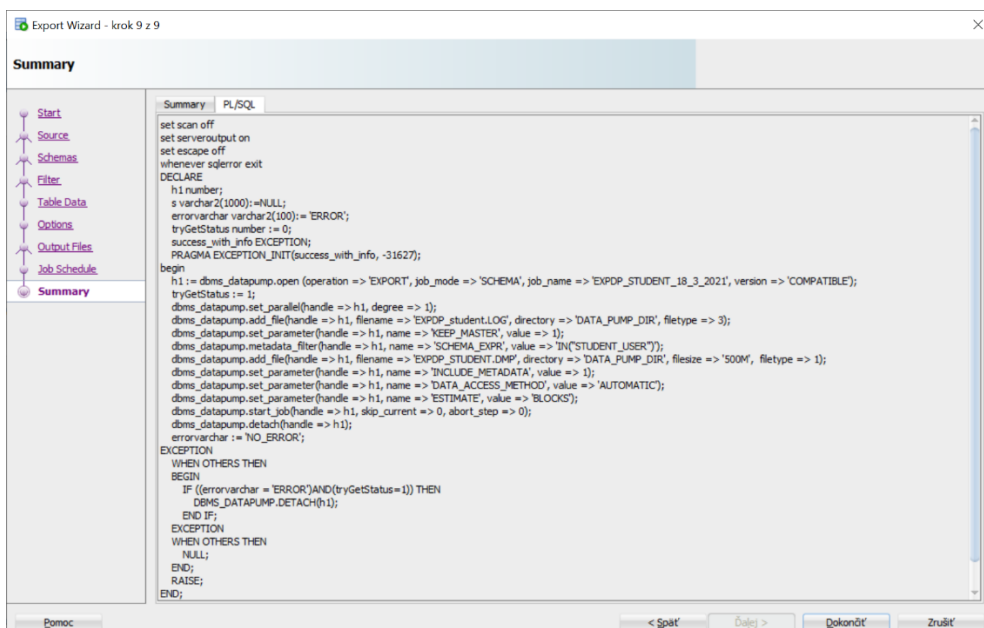


Fig. 6.67: PL/SQL script

The whole process can be monitored via *SQL Developer*. In the following figure, you can see that the specified log file has been created.

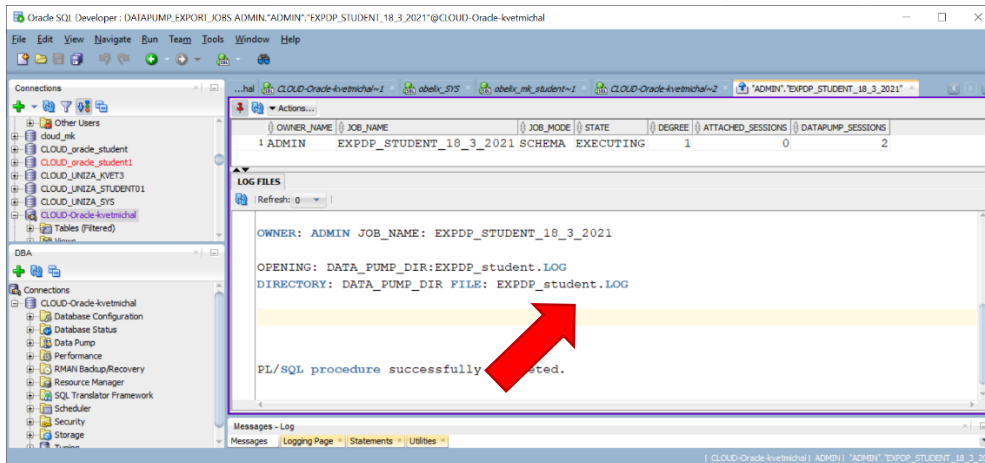


Fig. 6.68: Log file specification (ExpDp)

After the execution, let's make the output files visible and accessible via the *Oracle Object Storage* bucket. The principle is analogous, as already described in the *ImpDp* section. Execute the *Put_object* procedure of the *DBMS_CLOUD* package:

- *Credential_name* is a valid credential definition that has already been created. *Object_uri* is a created *URL* link to the *object* or the whole *bucket* extended by the destination file name.
- *Directory_name* represents an *Oracle cloud directory*, by which the original file is accessible. In our case, it is *DATA_PUMP_DIR*.
- The last parameter delimits the *file name* to be loaded into the *Object Storage* bucket. Its name has been specified during the data pump definition (*EXPDP_STUDENT.DMP*) – step 7.

```
BEGIN
  DBMS_CLOUD.PUT_OBJECT (
    credential_name => 'ATP_CREDENTIAL_MK',
    object_uri => 'https://objectstorage.eu-frankfurt-1.oraclecloud.com
                  /p/O8mTAKHDvT3qypjLlazUDoylyt-
                  KOemYLCqd19DhcpMiRS6BzM68vSd5EAi3OCd7
                  /n/frrt85axbzme
                  /b/bucket_library
                  /o/DP_STUDENT.DMP',
    directory_name => 'DATA_PUMP_DIR',
    file_name => 'EXPDP_STUDENT.DMP');
END;
/
```


Similarly, the log file can be put to the bucket:

```
BEGIN
DBMS_CLOUD.PUT_OBJECT (
  credential_name => 'ATP_CREDENTIAL_MK',
  object_uri => 'https://objectstorage.eu-frankfurt-
    1.oraclecloud.com
    /p/O8mTAKHDvT3gypjLlazUDoylyt-
    KOemYLcQdl9DhcpMiRS6BzM68vSd5EAi30Cd7
    /n/frrt85axbzme
    /b/bucket_library
    /o/EXPDP_student.LOG',
  directory_name => 'DATA_PUMP_DIR',
  file_name => 'EXPDP_student.LOG');
END;
/
```

In the *Oracle Cloud* environment, a particular *dump file* and a *log* are accessible. You can download them locally.

The screenshot shows the Oracle Cloud console interface. At the top, there's a navigation bar with the Oracle Cloud logo and a search bar. Below that, a banner indicates storage usage. The main section is titled 'bucket_library' and shows bucket information like visibility, namespace, and storage tier. On the left, there's a sidebar with 'Resources' and 'Objects' sections. The 'Objects' section displays a table of files in the bucket. A red arrow points to the file 'EXPDP_student.LOG' in the table.

Name	Last Modified	Size	Storage Tier
DP_STUDENT.DMP	Thu, Mar 18, 2021, 06:48:13 UTC	736 KB	Standard
EXPDP_student.LOG	Thu, Mar 18, 2021, 06:48:10 UTC	2.62 KB	Standard

Fig. 6.69: File in the bucket

Content of the log:

```
Processing "ADMIN"."EXPDP_STUDENT_18_3_2021":
Processing object type SCHEMA_EXPORT/TABLE/TABLE_DATA
. estimated "STUDENT_USER"."OS_UDAJE" 4.683 KB
. estimated "STUDENT_USER"."PREDMET" 4.683 KB
. estimated "STUDENT_USER"."PREDMET_BOD" 4.683 KB
. estimated "STUDENT_USER"."STUDENT" 4.683 KB
. estimated "STUDENT_USER"."ST_ODBORY" 4.683 KB
. estimated "STUDENT_USER"."ST_PROGRAM" 4.683 KB
. estimated "STUDENT_USER"."UCITEL" 4.683 KB
. estimated "STUDENT_USER"."ZAP_PREDMETY" 4.683 KB
Processing object type SCHEMA_EXPORT/TABLE/INDEX/STATISTICS/INDEX_STATISTICS
Processing object type SCHEMA_EXPORT/TABLE/STATISTICS/TABLE_STATISTICS
Processing object type SCHEMA_EXPORT/STATISTICS/MARKER
Processing object type SCHEMA_EXPORT/USER
Processing object type SCHEMA_EXPORT/SYSTEM_GRANT
Processing object type SCHEMA_EXPORT/ROLE_GRANT
Processing object type SCHEMA_EXPORT/DEFAULT_ROLE
Processing object type SCHEMA_EXPORT/PASSWORD_HISTORY
Processing object type SCHEMA_EXPORT/PRE_SCHEMA/PROCACT_SCHEMA
Processing object type SCHEMA_EXPORT/TABLE/TABLE
Processing object type SCHEMA_EXPORT/TABLE/CONSTRAINT/CONSTRAINT
Processing object type SCHEMA_EXPORT/TABLE/CONSTRAINT/REF_CONSTRAINT
Processing object type SCHEMA_EXPORT/POST_SCHEMA/PROCACT_SCHEMA
. . exported: "STUDENT_USER"."OS_UDAJE" 10.07 KB 35
```

```

. . exported: "STUDENT_USER"."PREDMET"          12.10 KB      218
. . exported: "STUDENT_USER"."PREDMET_BOD"       16.69 KB      372
. . exported: "STUDENT_USER"."STUDENT"           10.35 KB       37
. . exported: "STUDENT_USER"."ST_ODBORY"         6.773 KB        9
. . exported: "STUDENT_USER"."ST_PROGRAM"        22.39 KB      637
. . exported: "STUDENT_USER"."UCITEL"            7.25 KB       32
. . exported: "STUDENT_USER"."ZAP_PREDMETY"       25.93 KB     484
ORA-39173: Encrypted data has been stored unencrypted in dump file set
Master table "ADMIN"."EXPDP_STUDENT_18_3_2021" successfully loaded/unloaded
*****
Dump file set for ADMIN.EXPDP_STUDENT_18_3_2021 is:
/u03/dbfs/BD8E12C78924D962E0534111000A0D6E/data/dpdump/EXPDP_STUDENT.DMP
Job "ADMIN"."EXPDP_STUDENT_18_3_2021" successfully completed at Wed Mar 23 06:26:45
2022 elapsed 0 00:01:26

```

6.4.3 Useful notes

You can export all the data of defined schema or multiple schemas without the necessity to define individual tables separately using the clause `schemas`:

```
schemas=KVET1,KVETTT
```

It is also possible to define objects which will be part of *import* or export using clause *include* or *exclude*. Be aware when using these clauses, you refer to data dictionary views. Therefore, the names must be uppercase:

```
exclude=TABLE:"IN(\ 'FEES\ ',\ FEES_HISTORY\ ')"
include=TABLE:"LIKE \'K%\ '"
```

```
expdp db_kniznica@orcl
INCLUDE=TABLE:"LIKE \'K%\ ' "
directory=mk_exp_kniznica_dir
dumpfile=expdp_kniznica.dmp
logfile=expdp_kniznica.log

```

Lab 7 – Managing privileges

*This lab focuses on the Data Control Language (DCL) – **Grant** and **Revoke**. Privileges can be made to individual users, or roles can be created and associated with multiple users. Thus, instead of granting privileges to individual users sequentially, a defined role can be more useful for the maintenance.*

*When dealing with the **Grant** command, it is inevitable to distinguish between system and object privileges in case of allowing the grantee to inherit privileges.*

7.1 Introduction

Access control mechanisms to the database provide a significant security layer between users and objects, ensuring defense mechanisms. DBS systems highlight the strict control mechanisms provided by at least the privilege principle. Thus, any operation, which is not inevitable for a particular user, should be forbidden. DBS Oracle goes even further - the user does not have any privilege by default. Any user cannot even log on to the database without the grant.

Each database object is delimited by its creator (owner), who may assign appropriate privileges to individual users for object usage and manipulation. Privileges are managed using **Grant** and **Revoke** commands.

7.2 Grant command

This command allows you to assign privileges to the user or group, according to which a particular user will be able to manipulate with database objects. Generally, we can distinguish two **Grant** command types based on the handled object type.

7.2.1 System privilege management

System (database) privilege management allows users to access system resources. The syntax is like following:

```
grant database_privilege to { public | list_of_users }  
[with admin option];
```

There are three categories of this command reflecting the usage and access opportunities. As has been already mentioned, no privileges are automatically delegated. Thus, the first type covers the **Connect** privilege, which allows the user to connect to the database, but he can define no object. **Resource** privilege covers a group of database object definition opportunities – *create table, create trigger, create sequence, create procedure*, and some other ones, which are, however, out of the topic of this lab. To get the whole list of privileges associated with **Resource** privilege can be obtained using the following query. It uses data dictionary view **dba_sys_privs** (principles of data dictionary views are described in [Lab 14 – Data dictionary views](#)). The condition reflects the name of the privilege group. Notice that the value must be uppercase.

```
select privilege
from dba_sys_privs
where grantee = 'RESOURCE':
```

The last category of database privilege management is just *Dba* allowing the user to administer the database.

```
grant connect to kvet_eng;
grant resource to kvet_eng;
grant dba to kvet_eng;
```

The optional clause of the defined command is “*with admin option*”. It allows the user to grant received privileges to other users.

Let's have the following example. Create two *users* (*U1*, *U2*). One of them will have *Connect* privilege *granted*. Then, connect as a created user with *connect* privilege (*U1*) and try to add the same privilege (*Connect*) to user *U2*. As you can see, it is not possible because *Connect* privilege has not been granted with the “*with admin option*” clause. The following code shows the sequence of commands to demonstrate the principles. The last *Grant* command will raise the exception.

```
-- login as system user
create user U1 identified by password1;
grant connect to U1;
create user U2 identified by password2;
-- login as created user U1
grant connect to U2;
```

```
Error report -
ORA-01932: ADMIN option not granted for role 'CONNECT'
01932. 00000 - "ADMIN option not granted for role '%s'"
*Cause:      The operation requires the admin option on the role.
*Action:     Obtain the grant option and re-try.
```

However, if user *U1* has system privilege *Connect* – *with admin option* definition – it will be allowed to add a privilege to another user (*U2*) successfully. User *U1* can also *grant* privilege to user *U2* by using *with admin option* clause.

```
-- login as system user
grant connect to U1 WITH ADMIN OPTION;
-- login as created user U1
grant connect to U2;
```

```
Grant succeeded.
```

Naturally, several privileges to multiple users can be granted using one statement. Values are delimited by the comma (.). In this case, all privileges in the whole defined set of users are either *with the admin option* clause or not for any of them.

```
grant connect, resource to U1, U2;
```

```
grant connect, resource to U1, U2 with admin option;
```

Suppose it is necessary to divide the users based on particular clauses. In that case, separate commands must be used (user *U1* will get the privilege to grant it to other users, but user *U2* does not have such privilege).

```
grant resource to U1 with admin option;
```

```
grant resource to U2;
```

Notice that there is also a significant amount of other system privileges, which can be used. Principles and descriptions can be found in database system documentation (for DBS Oracle, use <https://docs.oracle.com/en/database/>).

7.2.2 Object privilege management

Object privilege command definition is always associated with the particular object by its name. The syntax is following:

```
grant object_privilege to { public | list_of users }  
[with grant option];
```

Notice the significant difference between object and system privilege in the syntax definition layer. In this case, for object privilege, the “*with grant option*” clause can be used. Generally, we distinguish four table privileges for accessing and managing table data – *Insert*, *Update*, *Delete* and *Select*. *Insert* and *Update* privilege can be extended by attribute list, to which the privilege applies (privilege to the *Select* statement can also be defined only for some attributes; however, it is done using views (see [Lab 12 – Views](#))). Grants represented by the attribute granularity are shown colored.

```
grant insert on personal_data to matiasko;
```

```
grant insert(personal_id, name, surname) on personal_data to krsak;
```

```
grant update on personal_data to matiasko;
```

```
grant update(date_to) on personal_data to krsak;
```

```
grant delete on personal_data to matiasko;
```

```
grant select on personal_data to matiasko;
```

However, if all mentioned privileges should be *granted* to the particular user, a special placeholder – “*all*” – can also be used. Thus, the last two commands are equivalent.

```
grant insert, update, delete, select on personal_data to matiasko;
```

```
grant all on personal_data to matiasko;
```

In the previous example, user “*krsak*” can insert into a *personal_data* table. However, only attributes *personal_id*, *name*, and *surname* (or their combinations) can be handled. Any attempt to insert another attribute value will end unsuccessfully by raising the following exception:

```
Error report -  
ORA-01031: insufficient privileges  
01031. 00000 - "insufficient privileges"  
*Cause:      An attempt was made to perform a database operation without  
              the necessary privileges.  
*Action:     Ask your database administrator or designated security  
              administrator to grant you the necessary privileges
```

A similar problem will occur if the user attempts to manage non-privileged table objects.

Management of *method* privilege is provided using **Execute** privilege. It ensures that the granted users (or groups) can launch such a *method*. DBS defines three method types – *procedure*, *function*, and *package* (reference [Lab 9 – Procedures, functions and packages](#)). Adding *execute* privilege to user “*matiasko*” is shown in the following commands (the first command reflects *procedure*, the second deals with *function*, and the last manages *package*). Highlighted value expresses the name of the object. Notice that the *execute* privilege cannot be set to the individual method of the package.

```
grant execute on procedure1 to matiasko;
```

```
grant execute on function1 to matiasko;
```

```
grant execute on package1 to matiasko with grant option;
```

By using the “*with grant option*” keyword, defined object privilege can be further delegated to other users. However, there is a significant difference in the functionality in comparison with the “*with admin option*” keyword for system (database) privilege, which will be described during the **Revoke** command definition, whereas it significantly influences mentioned operation and results.

7.3 Accessing another schema object

DML (*Insert*, *Update*, *Delete*, *Select*) statements can access not only owned tables but also structures, which were created by another user, who *granted* privileges for access and manipulation. The manipulation principles described sooner are the same. However, when dealing with objects, the owner schema must be declared explicitly. Therefore, by accessing another user object, the particular object name is preceded by the schema name.

Let’s execute the following statements by user “*kvet*”. In the first case, he accesses his table, whereas the second statement deals with a table in the “*kmat*” schema. Sure, user “*kvet*” must have granted privileges to the “*kmat*” table. Otherwise, an exception will be raised.

```
select name, surname
  from personal_data;      -- own table
```

```
select name, surname
  from kmat.personal_data;  -- table created (owned) by kmat
```

```
execute proc_man           -- own procedure
```

```
execute kmat.proc_man      -- procedure owned by kmat
```

7.4 Revoke command

This command allows the user to remove privilege from the particular user or group. Let’s consider the following syntax. Privilege in the syntax can be either *database* (system) privilege or *object* privilege (which also requires object naming used after the “*on*” keyword).

```
revoke privilege from { public | list_of users };
```

Also note the examples:

- Removing **database** privilege example:

```
revoke connect from matiasko;
```

```
revoke connect, resource from krsak;
```

```
revoke dba from kvet;
```

- Removing **object** privilege example:

```
revoke insert on personal_data from matiasko;
```

```
revoke select on personal_data from public;
```

The particular category forms the “**public**” role. If you grant any privilege to a *public* role, it will be automatically given to each user in the database space. Thus, such activity will be able to be performed by anyone. Therefore remember, that **public** role is implicitly *granted* to anyone and cannot be removed. However, it is strictly recommended to pay significant attention to granting privileges and monitoring accessible sources, data, and objects of the particular user.

Principles and limitations are described in the following example. Let’s have two users – *kvet* and *matiasko*. User *kvet* owns the table *personal_data* and *grants* the *Select* privilege to *matiasko*. Afterward, he also *grants* such *privilege* to all *users* (using **public** role). What will happen if the user *kvet* consecutively removes the *Select* privilege from the user *matiasko*? Will he even be allowed to query the *personal_data* table owned by the *kvet* user successfully?

Sure, he will be able. The reason is that user *kvet* is always part of the **public** role, so the privilege is still active. *Grant* commands are performed by the user *kvet*. *Select* statements are executed by the *matiasko* user.

```
-- KVET
grant select on personal_data to matiasko;
grant select on personal_data to public;
```

```
-- MATIASKO
select * from kvet.personal_data;
```

	PERSONAL_ID	NAME	SURNAME	STREET	TOWN	ZIP	NATIONALITY
1	841106/3456	Michael	Pearce	Kamenna 27	Banska Bystrica	97401	SK
2	840312/7845	Jack	Smith	Zelena 9	Nove Mesto nad Vahom	91501	SK
3	860907/1259	John	Young	Slnece namestie	Komarno	94501	SK
4	850130/3695	Carol	Pearce	Stred 49/7	Povazska Bystrica	01701	SK

```
-- KVET
revoke select on personal_data from matiasko;
```

```
-- MATIASKO
select * from kvet.personal_data;
```

	PERSONAL_ID	NAME	SURNAME	STREET	TOWN	ZIP	NATIONALITY
1	841106/3456	Michael	Pearce	Kamenna 27	Banska Bystrica	97401	SK
2	840312/7845	Jack	Smith	Zelena 9	Nove Mesto nad Vahom	91501	SK
3	860907/1259	John	Young	Slnece namestie	Komarno	94501	SK
4	850130/3695	Carol	Pearce	Stred 49/7	Povazska Bystrica	01701	SK

As already mentioned, there is a significant difference in management between *objects* and *system* privileges, and it is currently the suitable place to explain the characteristics deeper. In principle, **revoking object privilege cascade, system privilege remains**. Thus, if some user has the *privilege* to *grant* it to another user, at the moment such *privilege* is removed from him, it is automatically removed from all users who had such *privilege granted* by the particular user. It applies to object privileges.

Let's have the following example. User **Kvet**, who owns the *personal_data* table and has administrator privileges, will create two users (**Peter**, **Jacob**). He will grant the user *Peter* privilege to *connect* and the privilege to query his *personal_data* table (*Select* privilege). Moreover, such privileges can be granted later. Notice that these privileges must be granted using two commands, whereas one of them is *database* privilege associated with the “*with admin option*” clause, the second one is *object* privilege and is associated with the “*with grant option*” clause. Then, **Peter** connects to the database and grants his privileges to **Jacob**. Thus, **Jacob** can connect to the database and query the *personal_data* table residing in the **Kvet** schema. (Auxiliary question – can users **Peter** or **Jacob** create their objects? Why not?). Now, remove these privileges by user **ket** from **Peter** user. What will happen? Naturally, **Peter** will not be able to *connect* to the database and consequently will not be able to query any table. However, what about the user **Jacob**? Ascertain the solution using the following example – object privilege is removed (he will not be able to query *personal_data* table later), but database privilege will remain valid (he will be able to *connect* to the database).

The following code must be evaluated sequentially – the first column characterizes user **Kvet**, the second column deals with created user **Peter** and the last one is user **Jacob**. Be aware, if **Jacob** user queries *personal_data* of the **Kvet** user, raised *exception* does not explicitly describe the reason – **Jacob** will get information, that particular *table* does not exist. However, that is not true. He only does not have sufficient *privileges* to access it.

Tab. 7.1: Session management

Kvet	Peter	Jacob
<code>create user peter identified by pass_peter;</code>		
<code>create user jacob identified by pass_jacob;</code>		
<code>grant connect to Peter with admin option;</code>		
<code>grant select on personal_data to Peter with grant option;</code>		
	<code>-- Peter successfully connects to the database</code>	
	<code>select * from ket.personal_data;</code>	
	<code>grant connect to Jacob;</code>	

<i>Kvet</i>	<i>Peter</i>	<i>Jacob</i>
	<code>grant select on kvet.personal_data to Jacob;</code>	
		<code>-- Jacob successfully connects to the database</code>
		<code>select * from kvet.personal_data;</code>
<code>revoke connect from Peter;</code>		
<code>revoke select on personal_data from Peter;</code>		
	<code>-- Peter cannot connect later at all. An error was encountered performing the requested operation: ORA-01045: user Peter lacks CREATE SESSION privilege; logon denied</code>	
		<code>-- Jacob successfully connects to the database</code>
		<code>select * from kvet.personal_data;</code> <code>ORA-00942: table or view does not exist 00942. 00000 - "table or view does not exist"</code>

7.5 Grouping privileges to roles

Individual database systems allow the user to define *roles* (in DBS Postgres, its name is a *group*, however, principles are the same) to cover multiple *privileges*. These *roles* are consequently *granted* to *users*, so multiple *privileges* are not necessary to be granted individually. One of the examples of the role that has been already mentioned before – role *resource* – is system role, so it cannot be edited. The particular category covers *public role*, which is implicitly granted to each user (such role cannot be *revoked*).

The following code structure shows the syntax of the *role* definition. Then, the *role* is associated with the *privileges* using *Grant* and *Revoke* commands. In logical meaning in command, the group would behave the same as the user. Afterward, defined *roles* are *granted* to the users.

Role definition syntax:

```
create role role_name;
```

Setting privileges to role syntax:

```
grant privilege_name to role_name;
```

Associating user with the defined role syntax:

```
grant role_name to user_name;
```

This is an example of group definition, privilege management, and user group association.

Role definition syntax:

```
create role manager_role;
```

Setting privileges to role syntax:

```
grant select, insert, update(name, surname, street, town, zip)
on personal_data
to manager_role;
```

```
grant execute on function get_user_results to manager_role;
```

Setting privileges to role syntax:

```
grant manager_role to novakova, sicova;
```

7.6 Practice

Create *user_test* and *user_test2* accounts and grant them *connect* privileges.

1. Grant privilege to query your table *personal_data* to defined user *user_test*.
2. Can user *user_test* access your defined table? Can he change any value?
3. Can user *user_test2* access your defined table? Can he change any value?
4. Extend the previous privilege definition so that the *user_test* can grant the defined privilege to other users.
5. Grant the *Select* privilege on the *personal_data* table to all users by *user_test* account.
6. Can user *user_test2* access your defined table?
7. Remove the *Select* privilege from the *user_test2* account (by your personal account).
8. Can user *user_test2* access your defined table?
9. Remove the *Select* privilege on *personal_data* from the *user_test* user (by your personal account).
10. Can user *user_test* access your defined table?
11. Can user *user_test2* access your defined table?
12. Remove defined user accounts *user_test2*, *user_test2* if created explicitly by you.

Lab 8 – Advanced techniques of data retrieval

The focus of the Lab 2 is made by the *Select*, *From*, *Where* and *Order by* clauses of the *Select* statement. The discussion was done on the *Inner Join* limiting the output to the rows, which can be interconnected using multiple tables.

In this lab, we emphasize the aggregate functions by creating groups for which the calculation is done. Conditions related to the aggregations cannot be placed in the *Where* clause, whereas the groups have not been created yet. Instead, the Having clause is used, whereas it is evaluated at the end of the processing.

The second part is related to the extended versions of the *Join* operation and relational algebra operations Union, Difference, and set Intersection.

Finally, there is a practical discussion related to the self-relationship. In this chapter, it is evaluated from the data retrieval point of view.

8.1 Introduction

The *Select* statement is used to get data from one or more database tables, which can also be encapsulated by views. In most systems, the *Select* statement is the most often used *data manipulation language (DML)* command returning the result set consisting of zero or more rows. The main advantage of SQL is the non-procedurality of the statements. Thus, the query specifies only the result set format but does not specify how to get desired data, how to join them, or how to calculate results. This lab extends the capabilities of the previously defined lab – *Select* statements (Lab 2 – Basics of data retrieval) and highlights the possibilities of **aggregate functions**, extended methods for table joining with regards to an **outer join**, **recursive relationships**, and **joining the same table several times** in one *Select* statement.

8.2 Aggregate functions

Aggregate functions return a single result row based on groups of rows rather than on single rows. For this course, we will deal only with five main aggregate functions (**min**, **max**, **sum**, **avg**, **count**), using which you can understand principles and functionality. An aggregate function can be part of the *Select*, *Having*, and *Order by* clause. **You can never locate aggregate function to the Where clause** because the order of the execution processes conditions in the *Where* clause sooner than creating groups using *Group by* clause. Therefore, the system would not be able to evaluate it. If the condition is based on aggregate function, it must be processed after creating groups. Thus, it must be in the *Having* clause (place only conditions based on aggregate functions in the *Having* clause, standard conditions should be found in the *Where* clause. Putting common conditions on the *Having* clause is also possible, but ineffective).

Aggregate functions are commonly associated with the *Group by* clause, which defines the groups for which the aggregate function should be evaluated. Thus, **everything in the Select statement clause except the aggregate function MUST be noticed in the Group by clause, but there can also be something more**. As we can see in the following examples, it is often necessary to add other attributes (usually primary key) to prevent incorrect data groups processing (e.g., two namesakes cannot be processed as one person).

If only one value for the whole group created by the *Select* statement is returned, no **Group by** clause is necessary. This is because the result set consists of only one numerical value.

The following query gets the minimal value of the *student_id* attribute. One value is returned: 500422.

```
select min(student_id) from student;
```

To get the maximal value of the *student_id*, the principle is similar, but the **max** aggregate function will be used.

```
select max(student_id) from student;
```

The following query gets the total number of credits (*ects*) of one student (*student_id*=501103), which he can obtain if all registered subjects are passed successfully. Aggregate function **sum** is used.

```
select sum(ects) from study_subjects
where student_id=501103;
```

To get an average value for the processed attributes or function results, an *aggregate function avg* can be used. This case returns the average number of credits (*ects*) for the subject in the *school year* 2007 (the output value would be 4.8).

```
select avg(ects) from subject_year
where school_year=2007;
```

Count aggregate function in its pure way expresses the *cardinality* of the table – number of rows stored.

```
select count(*) from student;
```

The result expresses the total number of rows in the student table.

The expression inside the *aggregate function* can be *attribute value*, *function result* or can be substituted by the “*”, which reflects the whole row. Thus, the same results will be obtained if you replace the symbol “*” with the primary key or any *NOT NULL* attribute.

```
select count(student_id) from student;
```

```
select count(personal_id) from student;
```

However, if you put an attribute with *NULL* values inside the *aggregate function*, you will get a lower output value, whereas **aggregate functions ignore NULL values**. Let's consider the two following *Select* statements. The first one will return a value of 484. The second one will return 295.

```
select count(*) from study_subjects;
```

```
select count(result) from study_subjects;
```

So, 189 (484-295) rows in the table have a *NULL* value assigned for the *result* attribute.

```
select count(*) - count(result) from study_subjects;
```

As you can see, we can combine multiple *aggregate functions* in one statement, but they **MUST** be based on the same grouping set (they bind themselves to the same **Group by** clause, if defined).

Getting a number of the unique attribute values can be done by adding **Distinct** keyword:

```
select count(distinct class) from student;
```

The query's result is 4. This is because the attribute **class** consists of values 0, 1, 2, and 3. However, be aware **NULL** values are not processed at all. So, if you replace the value for **class 2** with **NULL** values, the same query will return only value 3.

```
update student  
set class = null  
where class = 2;
```

```
select count(distinct class) from student;
```

If you want to evaluate also **NULL** values, they must be replaced before processing, like this:

```
select count(distinct nvl(class, -1)) from student;
```

Notice also the difference between the following notations:

- **Count(distinct A)** – it removes duplicates for values of attribute *A*.
- **Distinct count(A)** – the result set will contain unique values of the function *count(A)* results.

8.3 Fundamentals for Group By clause management

If the output of the *Select* statement is only one value, **Group by** clause will not be present in the statement. Vice versa, if the aggregate function should be calculated for specific groups, they must be defined in the **Group by** clause.

If you state some attributes in the *Select clause* (not important whether direct or by using the function), they **MUST** be part of the **Group by** clause. Often, it is necessary to add other attributes to prevent incorrect data groups processing (e.g., two namesakes cannot be processed as one person).

Let's have the table "*flight*" identified by *id_flight* and *plane* table attribute – *capacity*. Let's also have the table consisting of sold flight numbers (*flight_ticket*). If you want to get actual free capacity for each *id_flight*, the *Select* clause will be following:

```
select id_flight, capacity - count(id_flight_ticket) ...
```

However, what about the **Group by** clause? Remember, at least each attribute in the *Select* clause except *aggregate function* must be present in the *Group by* set, thus remember, that also *capacity* **MUST** be there:

```
group by id_flight, capacity
```

8.4 Working with aggregate function Count and Group By clause

Let's have the following structure (table *study_subjects*) with these data (*assume that the particular table consists of only these values*). Consider the *Select* statements as well as provided results.

Tab. 8.1: Data table and result set

<i>Student_id</i>	<i>Subject_id</i>	<i>School_year</i>	<i>Result</i>	<i>Teacher_id</i>
501319	BL14	2005	C	EX001
501319	BE13	2005	D	KMT01
501319	BL11	2005		EX001
501319	BL11	2006	C	EX001
501201	IM16	2002	C	KMM01
501201	II08	2003		KDS01
501345	BA12	2002	C	KI003
501345	IS04	2003	D	KI001
550123	II07	2001		KI001
550123	IA07	2001	C	KMM02
550123	II17	2002	D	KI002

1. A total number of rows in the table:

```
select count(*) from study_subjects;
```

<i>COUNT(*)</i>
11

2. A total number of registered subjects for each student (the group is created for each student, thus, student identifier (*student_id*) must be placed in **Group by** section). Moreover, it is also placed in the *Select* clause:

```
select student_id, count(*)
from study_subjects
group by student_id;
```

<i>STUDENT_ID</i>	<i>COUNT(*)</i>
501319	4
501201	2
502345	2
550123	3

Tab. 8.2: Data table and result set

<i>Student_id</i>	<i>Subject_id</i>	<i>School_year</i>	<i>Result</i>	<i>Teacher_id</i>
501319	BL14	2005	C	EX001
501319	BE13	2005	D	KMT01
501319	BL11	2005		EX001
501319	BL11	2006	C	EX001
501201	IM16	2002	C	KMM01
501201	II08	2003		KDS01
501345	BA12	2002	C	KI003
501345	IS04	2003	D	KI001
550123	II07	2001		KI001
550123	IA07	2001	C	KMM02
550123	II17	2002	D	KI002

3. A total number of registered subjects for each student and school year (the group is created based on *student_id* attribute as well as *school_year*):

```
select student_id, school_year, count(*)
  from study_subjects
 group by student_id, school_year;
```

<i>STUDENT_ID</i>	<i>SCHOOL_YEAR</i>	<i>COUNT(*)</i>
501319	2005	3
501319	2006	1
501201	2002	1
501201	2003	1
501345	2002	1
201345	2003	1
550123	2001	2
550123	2002	1

Tab. 8.3: Data table and result set

<i>Student_id</i>	<i>Subject_id</i>	<i>School_year</i>	<i>Result</i>	<i>Teacher_id</i>
501319	BL14	2005	C	EX001
501319	BE13	2005	D	KMT01
501319	BL11	2005		EX001
501319	BL11	2006	C	EX001
501201	IM16	2002	C	KMM01
501201	II08	2003		KDS01
501345	BA12	2002	C	KI003
501345	IS04	2003	D	KI001
550123	II07	2001		KI001
550123	IA07	2001	C	KMM02
550123	II17	2002	D	KI002

A number of students assigned to the teacher:

```
select teacher_id, count(*)
  from study_subjects
  group by teacher_id;
```

TEACHER_ID	COUNT(*)
EX001	3
KDS01	1
KI001	2
KI002	1
KI003	1
KMT01	1
KMM01	1
KMM02	1

Tab. 8.4: Data table and result set

Student_id	Subject_id	School_year	Result	Teacher_id
501319	BL14	2005	C	EX001
501319	BL11	2005		EX001
501319	BL11	2006	C	EX001
501201	II08	2003	D	KDS01
501345	IS04	2003	C	KI001
550123	II07	2001		KI001
550123	II17	2002	D	KI002
501345	BA12	2002	C	KI003
501319	BE13	2005	D	KMT01
501201	IM16	2002	C	KMM01
550123	IA07	2001	C	KMM02

If you omit the attribute *teacher_id* in the *Select* clause, the results will be the same, but the output set will not contain *teacher_id* information. Instead, only the numbers will be listed.

4. Dealing with *NULL* values – aggregate functions ignore *NULL* values. Therefore, be careful that the provided result is correct, just as you expected
 - a) To eliminate *NULL* values – as you can see from the following table, rows which do not have assigned real value for *result* attribute are ignored.

```
select student_id, count(result)
  from study_results
  group by student_id;
```

STUDENT_ID	COUNT(RESULT)
501319	3
501201	1
502345	2
550123	2

Tab. 8.5: Data table and result set

<i>Student_id</i>	<i>Subject_id</i>	<i>School_year</i>	<i>Result</i>	<i>Teacher_id</i>
501319	BL14	2005	C	EX001
501319	BE13	2005	D	KMT01
501319	BL11	2005		EX001
501319	BL11	2006	C	EX001
501201	IM16	2002	C	KMM01
501201	II08	2003		KDS01
501345	BA12	2002	C	KI003
501345	IS04	2003	D	KI001
550123	II07	2001		KI001
550123	IA07	2001	C	KMM02
550123	II17	2002	D	KI002

- b) Eliminating NULL values and get a number of unique results (rows with the NULL value assigned to result attribute are ignored. Moreover, only unique values are evaluated. Thus, the result set will consist of a number of unique NOT NULL exam results for each student):

```
select student_id, count(DISTINCT result)
from study_results
group by student_id;
```

<i>STUDENT_ID</i>	<i>COUNT(DISTINCT RESULT)</i>
501319	2
501201	1
502345	2
550123	2

Tab. 8.6: Data table and result set

<i>Student_id</i>	<i>Subject_id</i>	<i>School_year</i>	<i>Result</i>	<i>Teacher_id</i>
501319	BL14	2005	C	EX001
501319	BE13	2005	D	KMT01
501319	BL11	2005		EX001
501319	BL11	2006	C	EX001
501201	IM16	2002	C	KMM01
501201	II08	2003		KDS01
501345	BA12	2002	C	KI003
501345	IS04	2003	D	KI001
550123	II07	2001		KI001
550123	IA07	2001	C	KMM02
550123	II17	2002	D	KI002

Be aware incorrect *Group by* definition can lead to incorrect results:

```
select name, surname, count(*)
from personal_data JOIN student using(personal_id)
group by name, surname;
```

NAME	SURNAME	COUNT(*)
Mark	Bailey	2
Milan	Clarke	2
Jack	Clever	1

If two people have the same *first name* and surname, they will be considered *one person*, which is **absolutely incorrect**. Therefore, an additional attribute for distinguishing and separating these people should be added. In our case, we add there primary key – *personal_id*.

```
select name, surname, count(*)
  from personal_data JOIN student using(personal_id)
  group by name, surname, personal_id;
```

NAME	SURNAME	COUNT(*)
Mark	Bailey	2
Milan	Clarke	1
Milan	Clarke	1
Jack	Clever	1

Let's have two following *Select* statements (based on the course student model). Is there any difference? How does it influence the results?

```
select name, surname, count(*)
  from personal_data JOIN student using(personal_id)
                        JOIN study_subjects using(student_id)
  group by name, surname, personal_id;
```

NAME	SURNAME	COUNT(*)
William	Whittel	9
Mark	Bailey	6
Jack	Robinson	8

```
select name, surname, count(*)
  from personal_data JOIN student using(personal_id)
                        JOIN study_subjects using(student_id)
  group by name, surname, student_id;
```

NAME	SURNAME	COUNT(*)
Mark	Bailey	4
Mark	Bailey	2
Jack	Robinson	3
Jack	Robinson	1
Jack	Robinson	4
William	Whittel	5
William	Whittel	4

The answer is clear. The first statement *gets the number of registered subjects* for a particular *person*. In contrast, the second statement *gets the number of registered subjects* for a specific *student*. In principle, results are different because each person can be present as a student multiple times (e.g., bachelor, master study).

8.5 Having clause

If you want to process data based on *aggregate function* condition, such expression must be placed in the **Having** clause, never put in the **Where** clause, whereas it is impossible. The evaluation starts with the *Where* clause and joining operations. Afterward, the groups are created. Thus, the *aggregate function* cannot be evaluated in the *Where* clause, whereas groups have not been created yet.

Let's have the *study_subjects* table. We want to list the *number of students* registered for the particular subject during the *year 2010*. However, the result set should contain those subjects which have at least 5 registered students. *Aggregate function* result is therefore limited to minimal value 5:

```
select subject_id, count(*)
  from study_subjects
 where school_year = 2010
 group by subject_id
 having count(*) >= 5;
```

As already mentioned, the *aggregate function* cannot be placed in the *Where* clause. It would lead to the *ORA-00934* exception.

```
select subject_id, count(*)
  from study_subjects
 where school_year = 2010
       and count(*) >= 5
 group by subject_id;
```

```
ORA-00934: group function is not allowed here
```

The following example shows how to list students based on their study results. The aim is to get the students with the worst results. So, first of all, calculate the study results for each student and sort the result set based on results.

```
select student_id,
       avg(decode(result, 'A', 1, 'B', 1.5, 'C', 2, 'D', 2.5, 'E', 3, 4))
  from study_subjects
 group by student_id
 order by 2 desc, 1;
```

The first proposed solution is based on using the *Rownum* function. For each row returned by the query, pseudo column *Rownum* indicates the order. The first selected row has a value of 1. The second has 2, and so on. Sorting the result set based on study results and limiting the *Rownum* value to 1 will list only one student, although several students have the same results.

```
select * from(
  select student_id,
         avg(decode(result, 'A', 1, 'B', 1.5, 'C', 2, 'D', 2.5, 'E', 3, 4))
    from study_subjects
   group by student_id
   order by 2 desc, 1
 )
 where rownum = 1;
```

The correct solution is based on using *Set* operations and subquery. The calculated study result grade average is compared to the highest (worst) values obtained by the subquery. The solution can look like this.

```
select student_id,
       avg(decode(result, 'A', 1, 'B', 1.5, 'C', 2, 'D', 2.5, 'E', 3, 4))
from study_subjects
group by student_id
having avg(decode(result, 'A', 1, 'B', 1.5, 'C', 2, 'D', 2.5, 'E', 3, 4)) in
       (select max(avg(decode(result, 'A', 1, 'B', 1.5, 'C', 2, 'D', 2.5, 'E', 3, 4)))
        from study_subjects
         group by student_id);
```

Notice that DBS Oracle allows you to define an aggregate function from an aggregate function, but it is the specialty of that DBS. The inner aggregate function reflects the *Group by* clause. The outer aggregate function processes obtained value and cannot have *Group by* section at all. So, only one value can be returned. First of all, the nested *Select* statement gets the average value for each student:

```
(select avg(decode(result, 'A', 1, 'B', 1.5, 'C', 2, 'D', 2.5, 'E', 3, 4))
      as st_avg
from study_subjects
group by student_id
)
```

Afterward, the maximal value from that is obtained. Notice that there is a necessity to define column alias.

```
(select max(st_avg)
from
  (select avg(decode(result, 'A', 1, 'B', 1.5, 'C', 2, 'D', 2.5, 'E', 3, 4))
      as st_avg
   from study_subjects
   group by student_id
  )
)
```

Complete solution can look like this:

```
select student_id,
       avg(decode(result, 'A', 1, 'B', 1.5, 'C', 2, 'D', 2.5, 'E', 3, 4))
from study_subjects
group by student_id
having avg(decode(result, 'A', 1, 'B', 1.5, 'C', 2, 'D', 2.5, 'E', 3, 4)) in
       (select max(st_avg)
        from
          (select avg(decode(result, 'A', 1, 'B', 1.5, 'C', 2, 'D', 2.5, 'E', 3, 4))
              as st_avg
           from study_subjects
           group by student_id
          )
       );
```

So, to get a universal solution, a nested (inner) *Select* statement must be used.

Another example is based on listing the *oldest actual student* based on his registration date (*first_date*). For these purposes, please *update* table student using the following command:

```
update student
  set first_date = to_date('01.09.2007', 'DD.MM.YYYY')
  where first_date is null and status = 'S';
```

Then, the oldest students can be listed as follows. Notice that the *first_date* attribute value is compared with the minimal value.

```
select name, surname, personal_id, first_date
  from personal_data join student using(personal_id)
  where status = 'S' and first_date IN (select min(first_date)
                                         from student
                                         where first_date is null);
```

Now, you can *rollback* the transaction so that table will hold original values.

8.6 Extended versions of table joining

Join is a query functionality that combines rows from two or more tables or views. It is performed whenever multiples tables appear in the *From* clause of the query (except *Cartesian product*). To avoid a *Cartesian product*, *join conditions* must be defined based on the whole attribute set for the connection.

The syntax of the *JOIN*:

```
select ...
  from table_name1 [ {INNER | {LEFT | RIGHT | FULL} [OUTER]} ] JOIN
    table_name2 { ON (joining_conditions) | USING (column_list) }
    [ {INNER | {LEFT | RIGHT | FULL} [OUTER]} ] JOIN
    table_namen { ON (joining_conditions) | USING (column_list) }
```

There are several *JOIN* types:

- *[INNER] JOIN* – the result set to be subsequently processed consists only of attributes with corresponding (same) values of *foreign key* and particular referenced *primary key*.
- *OUTER JOIN* – extends the *INNER JOIN* principle by adding data to the result set, which do not have the corresponding reference in the second table. Three types of *OUTER JOIN* are distinguished:
 - *LEFT OUTER JOIN* – all table data from the left table are processed, and they are supplemented by rows of the right table, which can be joined (there is a reference to PK),
 - *RIGHT OUTER JOIN* – all table data from the right table are processed, and they are supplemented by rows of the left table, which can be joined (there is a reference to PK),
 - *FULL OUTER JOIN* – consists of all data from both tables. Relevant data are joined. In this case, there is no data loss.
- *SEMI JOIN* gets rows from one table, which can be joined with the second one. It is based on these condition clauses – *IN*, *EXISTS* keywords.
- *ANTI JOIN* is vice versa based on the negations. It provides data from the first table, which cannot be joined with the second one. Thus keywords *NOT IN* and *NOT EXISTS* are used.

Graphical representation focusing on the provided data are shown in fig. 8.1.

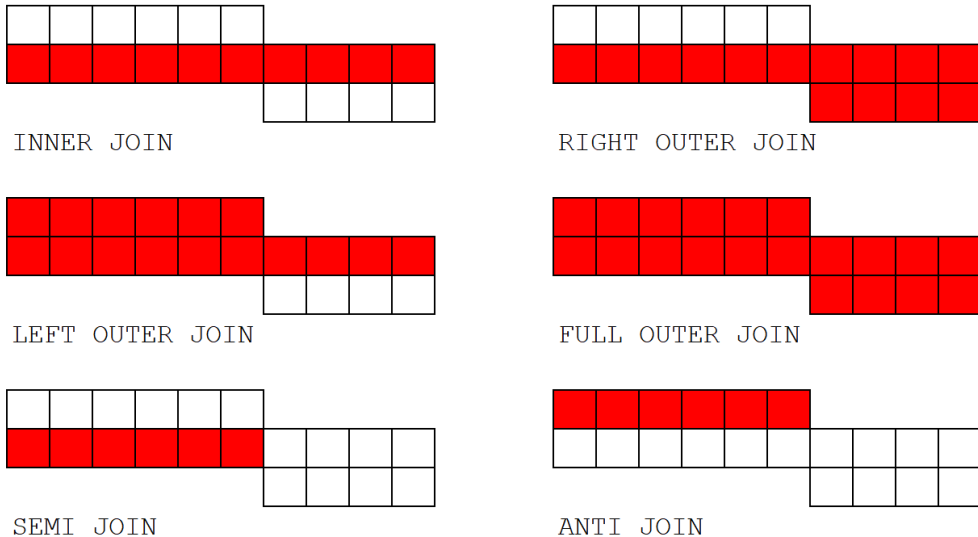


Fig. 8.1: Join types

Be aware. If you do not JOIN tables correctly, a **Cartesian product** is generated. Therefore, special attention must be used to reference composite keys. In both following examples, a **Cartesian product** is produced. The first example does not use the join condition at all. The second one uses only one element of the composite key.

```
select name, surname, s.student_id
  from personal_data, student s;
```

```
select field_name, spec_name, student_id
  from student JOIN st_field USING (field_id);
```

NATURAL JOIN is a particular category. In this case, it is not possible to use the *ON* (condition) or *USING* clause because the *join* will be done under the equivalent column names in the tables. Referential integrity is not checked. Such an approach is not used very often, whereas it can provide undesirable results, e.g., if the attribute name is renamed.

8.6.1 INNER JOIN type

Inner join has been described in Lab 2 – Basics of data retrieval. It uses only rows, which can be connected directly.

```
select name, surname, student_id, subject_id, result
  from personal_data JOIN student USING (personal_id)
                        JOIN study_subjects USING (student_id);
```

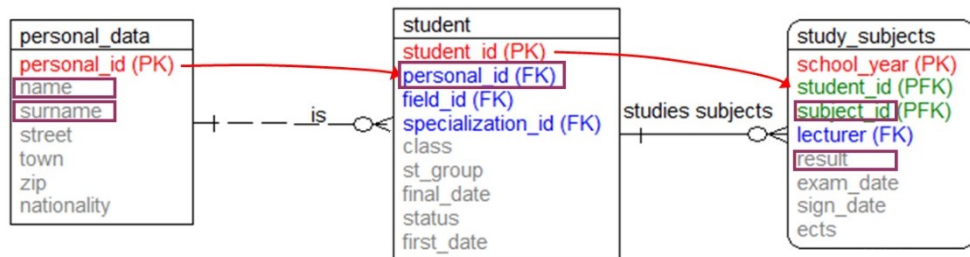


Fig. 8.2: Joining

8.6.2 ON / USING CLAUSE

USING keyword can be used only if the names of the attributes to be joined have the same names.

```
select name, surname, student_id, subject_id, result
  from personal_data JOIN student USING (personal_id)
        JOIN study_subjects USING (student_id);
```

USING clause is impossible to be used in the following example. Namely, in the *teacher* table, the primary key is *teacher_id*, but the reference in the *study_subjects* table is named as a *lecturer*.

```
select name, surname, teacher_id, subject_id
  from teacher t
    JOIN study_subjects ss ON (t.teacher_id = ss.lecturer);
```

ON – always possible to be done. Interconnected attributes are listed in both tables.

```
select name, surname, s.student_id, subject_id, result
  from personal_data pd
    JOIN student s ON (pd.personal_id = s.personal_id)
    JOIN study_subjects ss ON (s.student_id = ss.student_id);
```

8.6.3 LEFT OUTER JOIN type

Left Join selects all data from the left table of the relationship definition and connects them to the table on the right side, if possible. If the row cannot be joined to the second table, particular attributes reflecting the right table of the relationship will be listed as *NULL*.

```
select name, surname, p.personal_id,
       s.personal_id, s.student_id
  from personal_data p LEFT JOIN student s
                        ON (p.personal_id = s.personal_id);
```

Notice the *NULL* values for the *student_id* attribute as well as *personal_id_1* (created by renaming the *personal_id* attribute of the table student, whereas the result set must have unique column names).

SURNAME	PERSONAL_ID	PERSONAL_ID_1	STUDENT_ID
Pearce	855122/8569	855122/8569	550698
Hoom	890608/4543	890608/4543	550807
Murgas	900913/3326	900913/3326	550945
Pearce	841201/1248	(null)	(null)
Austin	871124/3578	(null)	(null)

```
select *
  from teacher LEFT JOIN study_subjects
           ON (teacher.teacher_id = study_subjects.lecturer);
```

Notice *NULL* values for attributes belonging to the *study_subjects* table.

NAME	SURNAME	DEPARTMENT	SCHOOL_YEAR	STUDENT_ID	SUBJECT_ID	LECTURER	RESULT	EXAM_DATE	SIGN_DATE	ECTS
Arnas	Beaudoin	DTK	2004	550945	BN10	KTK01	D	30.01.2003	09.07.2005	4
Wiliam	Santos	DI	2003	550945	BI06	KI001	C	31.05.2004	13.06.2004	6
Wiliam	Santos	DI	2008	550545	BI02	KI001	A	12.06.2009	30.05.2009	6
Mark	Madrigal	Gar	(null)	(null)	(null)	(null)	(null)	(null)	(null)	(null)
Owen	Boudreau	DTK	(null)	(null)	(null)	(null)	(null)	(null)	(null)	(null)

From the historical perspective, also the following syntax could be used. However, now, for clarity, it has been replaced with the *Left Join* keyword. Thus, everything is taken from the *teacher* table supplemented by the *study_subjects* table data, if possible (based on *join*).

```
select *
  from teacher, study_subjects
     where teacher.teacher_id = study_subjects.lecturer(+);
```

8.6.4 RIGHT OUTER JOIN type

Right Join selects all data from the right table of the relationship definition and connects them to the table on the left side, if possible. If the row cannot be joined to the first (left table of the relationship) table, particular attributes reflecting the left table of the relationship will be listed as *NULL*.

```
select name, surname, p.personal_id
  from personal_data p RIGHT JOIN student s
           ON (p.personal_id = s.personal_id);
```

SURNAME	PERSONAL_ID
Smith	840312/7845
Young	860907/1259
Pearce	850130/3695
Whittel	830514/5341

What about the data result differences between the previous *Select* statement modeled using *Right Join* and standard *Inner Join* in this case? Try to explain. The solution is based on relationship type with an emphasis on membership.

```
select name, surname, p.personal_id
  from personal_data p JOIN student s ON (p.personal_id = s.personal_id);
```

8.6.5 FULL OUTER JOIN type

Full Outer Join gets all the data from both tables. If defined table rows can be joined, it is done. If not, particular values will be noted as *NULL* values. It is done using the *Full Join* keyword for the relationship definition type.


```
select * from personal_data FULL JOIN contact using (personal_id);
```

Left table data can hold *NULL* values. The right table data can have *NULL* values as well. But naturally, it cannot happen that the whole row would hold *NULL* values due to primary key definition, which cannot hold *NULL* values.

NAME	SURNAME	STREET	TOWN	ZIP	NATIONALITY	TYPE	VALUE
Milan	Clarke	Ligetska 10	Handlova	97251	SK	M	8404097900
Thomas	Hall	SNP 41	Slovenska Lupca	97613	SK	M	908123456
(null)	(null)	(null)	(null)	(null)	(null)	M	1234567890
Sim	Eas	Kolarovce 12	Kolarovce	1401	SK	(null)	(null)
Daniel	Gomes	Razusa 40/10	Prievidza 4	97101	SK	(null)	(null)
John	Young	Bratislavská cesta 2	Zilina	1001	SK	(null)	(null)

8.6.6 SEMI JOIN type

Semi Join type selects all data from the left table of the relationship, which can be *inner* joined to the right table.

```
select *
  from personal_data
  where personal_id IN (select personal_id from student);
```

Let's also consider the second example providing similar data. Is there any difference between these two mentioned *Select* statements? If so, why? The answer is based on duplicates.

```
select p.*
  from personal_data p JOIN student s ON (p.personal_id = s.personal_id);
```

8.6.7 ANTI JOIN type

Anti Join type selects all data from the left table of the relationship, which cannot be *inner* joined to the right table. It is modeled by the *Set* operators.

```
select *
  from personal_data
  where personal_id NOT IN (select personal_id from student);
```

```
select *
  from personal_data p
  where NOT EXISTS(select 'x '
                    from student s
                    where s.personal_id = p.personal_id);
```

The following solution is similar to *ANTI JOIN* but does not remove duplicate values from the result set.

```
select p.*
  from personal_data p LEFT JOIN student ON (p.personal_id=s.personal_id)
  where s.personal_id IS NULL;
```

8.6.8 NATURAL JOIN type

Natural Join type reflects *Equi Join* type and is constructed so that relationship is created according to attributes with the same name. So, no *USING* nor *ON* keyword is used. Associated tables must have one or more identically named columns. Moreover, columns must have the same data type. *Using this type can be a bit dangerous because of the naming. Therefore, avoid using this approach if you are not sure that the attribute names cannot be changed (even later).*

```
select name, surname, personal_id
from personal_data NATURAL join student;
```

8.7 Relational algebra operations

An essential and inseparable part of any relational data model cover languages that allow you to specify operations that can make changes to the database or get required data from it. The query itself can be considered as a functionality of the database, which results in providing data in relations. The languages for query definitions are based on *relational algebra* and *relational calculus*.

Relational algebra represents the procedural language describing features and functionalities by which desired data results can be obtained. It is formed based on the algebraic concept. *Relational calculus* represents a declarative language, which describes the properties of the result set. Using *relational algebra* expressions, database commands can be constructed using several operations through which queries can be defined, expressed, and consequently evaluated by the optimizer.

These eight operations create the basic operation set of the *relational algebra*:

- Selection
- Projection
- Cartesian product
- Union
- Difference
- Intersection
- Division
- Join
- Split

These operations can be classified using various aspects:

- Number of source relations
 - *unary* – selection, projection
 - *binary* – Cartesian product, union, difference, intersection, division, join and split.
- Relation types
 - *Set operations* – union, difference, intersection, Cartesian product.
 - *Relational operations* – join, split, division, selection, projection.

Selection, projection, Cartesian product, and Inner Join, have been described in [Lab 2 – Basics of data retrieval](#) with the emphasis of *Select* statement syntax, therefore in the following section, we will define the rest part, again with the focus on real usage in the *Select* statement environment. The Operations *Union, Difference, and Intersection* require the processed sets to **be union compatible**, meaning they share the same amount of attributes

with the same data types and order. This is because of the attribute context considerations. For example, consider the following relations, they are *union compatible*.

- Student (name, surname)
- Employee (name, surname)

Following examples, however, show the relations, which are not compatible and therefore cannot be processed using mentioned operations:

- Person (name, surname)
- Country (name, population)

Another example of *union non-compatibility* is based on two student group definitions (students in the *Zilina* and students in the detached office *Prievidza*). Again, the relations are not compatible because of the attribute order:

- Student_ZA (name, surname, personal_id, class, status)
- Student_PD (personal_id, name, surname, class, status)

In the literature, other relational algebra operations can be defined, which form the extension of basic *relational algebra* operations:

- Natural join
- Theta join
- Inequi join
- External join
- Semi join
- Complement

8.7.1 Union operation

Union operation creates from two relations $R_1(A_1, A_2, \dots, A_n)$, $R_2(A_1, A_2, \dots, A_n)$ and the third relation R_3 with the same attribute definition $R_3(A_1, A_2, \dots, A_n)$, for which applies, that data tuple is part of the result relation R_3 , if it belongs to either input relation R_1 or relation R_2 :

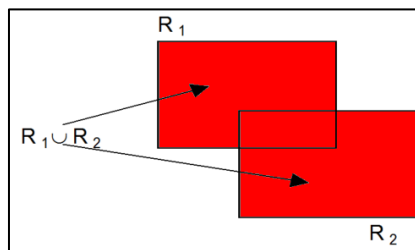


Fig. 8.3: Relational algebra operation Union

Graphical representation of the defined operation is following:

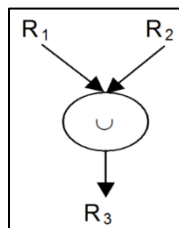


Fig. 8.4: Relational algebra operation Union

In the *Select* statement, it is represented by the *UNION* operation.

Let's have the example formed by two tables consisting of the musical instrument player information – *saxophone_player* and *guitar_player* table with the following data (export can be found in your CD, respectively server – *exp_music.exp*).

Saxophone_player table data:

SURNAME	PERSONAL_ID	CLASS
Pearce	841106/3456	3
Smith	840312/7845	2
Young	860907/1259	2
Pearce	850130/3695	1
Roger	781015/4431	3
Whittel	830514/5341	7

Fig. 8.5: Saxophone_player table data

Guitar_player table data:

SURNAME	PERSONAL_ID	CLASS
Pearce	850130/3695	1
Whittel	830514/5341	2
Bailey	800407/3522	1

Fig. 8.6: Guitar_player table data

Each table consists of the *name*, *surname*, *personal_id* of the *student*, as well as *class*.

To get data from both tables regardless of the studied musical instrument, *UNION* operation can be used.

```
select name, surname, personal_id
  from saxophone_player
 UNION
select name, surname, personal_id
  from guitar_player;
```

Notice that *UNION* operation automatically removes duplicate tuples. The cardinality of the table *saxophone_player* is 6, and the cardinality of the table *guitar_player* is 3. However, the result set consists of only 7 rows (*Carol Pearce* and *William Whittel* study both musical instruments).

	NAME	SURNAME	PERSONAL_ID
1	Carol	Pearce	850130/3695
2	Jack	Smith	840312/7845
3	John	Young	860907/1259
4	Mark	Bailey	800407/3522
5	Michael	Pearce	841106/3456
6	Peter	Roger	781015/4431
7	William	Whittel	830514/5341

To ensure that no duplicates will be removed, use the operator *UNION ALL* instead of *UNION* in its pure form.

```
select name, surname, personal_id
  from saxophone_player
UNION ALL
select name, surname, personal_id
  from guitar_player;
```

Now, the result set contains 9 rows:

	NAME	SURNAME	PERSONAL_ID
1	Michael	Pearce	841106/3456
2	Jack	Smith	840312/7845
3	John	Young	860907/1259
4	Carol	Pearce	850130/3695
5	Peter	Roger	781015/4431
6	William	Whittel	830514/5341
7	Carol	Pearce	850130/3695
8	William	Whittel	830514/5341
9	Mark	Bailey	800407/3522

Column names are formed based on the attribute name of the first *Select* statement. Thus, if the *guitar_player* attribute was renamed to *first_name*, the result set would contain “name” for the attribute name, whereas it is formed by the tuples from the *saxophone_player* table first.

```
alter table guitar_player rename column name to first_name;
```

```
select name, surname, personal_id
  from saxophone_player
UNION ALL
select first_name, surname, personal_id
  from guitar_player;
```

	NAME	SURNAME	PERSONAL_ID
1	Michael	Pearce	841106/3456
2	Jack	Smith	840312/7845
3	John	Young	860907/1259
4	Carol	Pearce	850130/3695
5	Peter	Roger	781015/4431
6	William	Whittel	830514/5341
7	Carol	Pearce	850130/3695
8	William	Whittel	830514/5341
9	Mark	Bailey	800407/3522

Therefore, if a column alias is used, it must be placed in the first *Select* statement:

```
select name, surname, personal_id as PID
  from saxophone_player
UNION ALL
select first_name, surname, personal_id
  from guitar_player;
```

	NAME	SURNAME	PID
1	Michael	Pearce	841106/3456
2	Jack	Smith	840312/7845

	NAME	SURNAME	PID
3	John	Young	860907/1259
4	Carol	Pearce	850130/3695
5	Peter	Roger	781015/4431
6	Wiliam	Whittel	830514/5341
7	Carol	Pearce	850130/3695
8	Wiliam	Whittel	830514/5341
9	Mark	Bailey	800407/3522

Vice versa, the *Order* method can be set at the end of the last *Select* statement. Naturally, it sorts the whole result set (do not place the *Order by* clause at the end of each statement, it is not possible):

```
select name, surname, personal_id as PID
  from saxophone_player
 UNION ALL
select first_name, surname, personal_id
  from guitar_player
  order by PID;
```

Notice that defined alias must be used in the *Order by* clause. Original name (*personal_id*) cannot be used.

8.7.2 Difference operation

Difference operation creates from two relations $R_1(A_1, A_2, \dots, A_n)$ and $R_2(A_1, A_2, \dots, A_n)$ the third relation R_3 with the same attribute definition $R_3(A_1, A_2, \dots, A_n)$, for which applies, that data tuple is part of the result relation R_3 , if it belongs to input relation R_1 , but is not part of the relation R_2 :

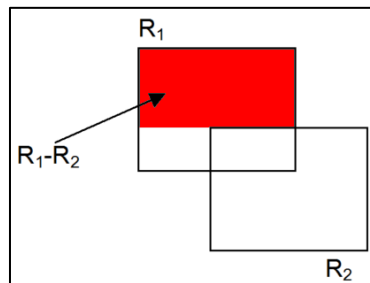


Fig. 8.7: Relational algebra operation Difference

Graphical representation of the defined operation is following:

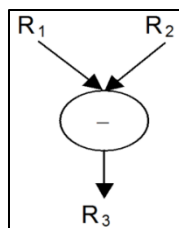


Fig. 8.8: Relational algebra operation Difference

In *Select* statements, the solution can be provided using *SET* operators or by using *MINUS* operation. Let's get the list of the students, who study “*Information systems*” as a field, but their specialization is not “*Applied informatics*”. For illustration purposes, create two tables. Table *ST_AI* will contain students of “*Applied informatics*”. *ST_IS* will contain all students of the “*Information systems*” as a field. Information systems field meets the value *field_id* = 200, *Applied informatics* is *specialization_id* = 2.

```
Create table ST_AI
as select name, surname, personal_id
from personal_data p
where EXISTS (select 'X'
              from student s
              where p.personal_id = s.personal_id
                 and field_id = 200
                 and specialization_id = 2);
```

```
Create table ST_IS
as select name, surname, personal_id
from personal_data p
where EXISTS (select 'X'
              from student s
              where p.personal_id = s.personal_id
                 and field_id = 200);
```

To get the results, the *MINUS* operator can be used.

```
select * from ST_IS
MINUS
select * from ST_AI;
```

The rest principles (aliases, sorting possibilities) are the same as described for the *UNION* operation.

8.7.3 Intersection operation

Intersection operation creates from two relations $R_1(A_1, A_2, \dots, A_n)$ and $R_2(A_1, A_2, \dots, A_n)$ the third relation R_3 with the same attribute definition $R_3(A_1, A_2, \dots, A_n)$, for which applies that data tuple is part of the result relation R_3 if it belongs to input relation R_1 as well as to relation R_2 :

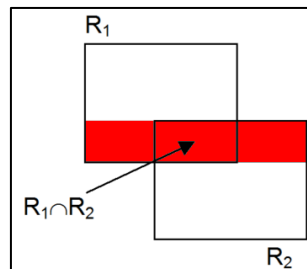


Fig. 8.9: Relational algebra operation Intersection

Graphical representation of the defined operation is following:

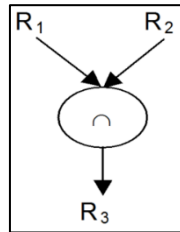


Fig. 8.10: Relational algebra operation Intersection

For illustration purposes, let's use previously created tables – *saxophone_player* and *guitar_player* table (export can be found in your CD, respectively server – *exp_music.exp*).

To get the solution – *students* playing saxophone as well as guitar, operation *INTERSECT* can be used:

```
select name, surname, personal_id
  from saxophone_player
 INTERSECT
select first_name, surname, personal_id
  from guitar_player;
```

NAME	SURNAME	PERSONAL_ID
Carol	Pearce	850130/3695
William	Whittel	830514/5341

The rest principles (aliases, sorting possibilities) are the same as described for the *UNION* operation.

8.8 Recursive relationships

A *recursive relationship* connects a *single table to itself* serving in another role (*person* has his *mother* and *father*, who are also *persons*; *the employee* is obviously supervised by one *manager*, whose data can also be found in the *employee* table). The *recursive relationship* defines a reference of the *foreign key* to the same table. Therefore, the *foreign key* attributes *must be renamed*.

Regarding the primary key, foreign key, and referential integrity definition. Can a *recursive relationship* be defined as *identifying*? Why not?

Modeling principles have been described in [Lab 4 – Data modeling](#), now will deal with using *recursive relationships* in the *Select* statements.

Principles will be described using *Smith's* family tree. To store it in the database, the *person_rec* table will be created with the explicit management of the parents (*mother_id* and *father_id*). Let's have the following family tree (fig. 8.11):

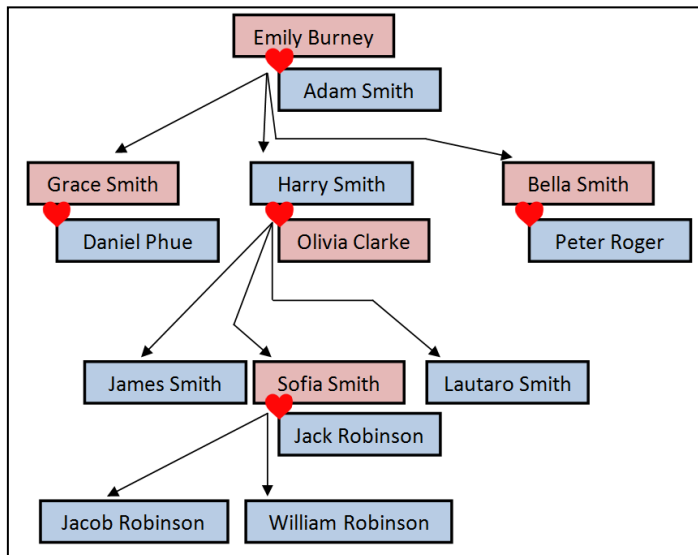


Fig. 8.11: Family tree

For illustration purposes, each person will be delimited by the unique identifier (*person_id*), which will also be the primary key of the table (fig. 8.12):

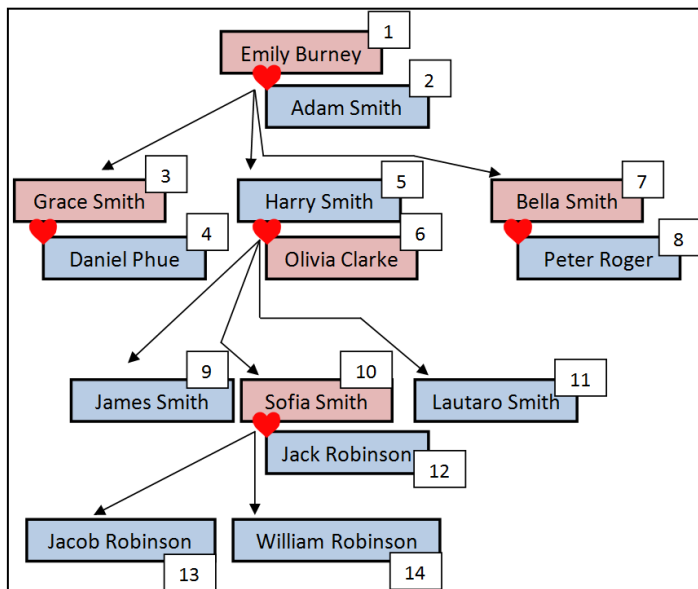


Fig. 8.12: Family tree

Next commands show the *person_rec* table definition:

```
create table person_rec(
    person_id integer primary key,
    name varchar2(20),
    surname varchar2(20),
    mother_id integer,
    father_id integer
);
```

```
alter table person_rec
add foreign key (mother_id)
references person_rec(person_id);
```

```
alter table person_rec
add foreign key (father_id)
references person_rec(person_id);
```

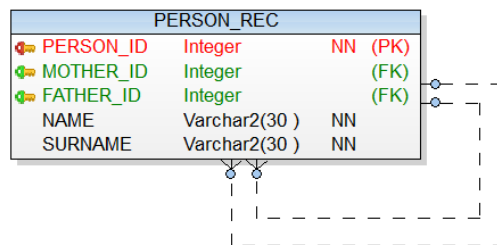


Fig. 8.13: *Person_rec* data model

Data shown in the previous figure are loaded into the database (script can be downloaded from the server – *family_tree_script.txt*):

PERSON_ID	NAME	SURNAME	MOTHER_ID	FATHER_ID
1	Emily	Burney		
2	Adam	Smith		
3	Grace	Smith	1	2
4	Daniel	Phue		
5	Harry	Smith	1	2
6	Olivia	Clarke		
7	Bella	Smith	1	2
8	Peter	Roger		
9	James	Smith	6	5
10	Sofia	Smith	6	5
11	Lautaro	Smith	6	5
12	Jack	Robinson		
13	Jacob	Robinson	10	12
14	William	Robinson	10	12

Fig. 8.14: Data in the *person_rec* table

Task 1: Get the name of the *mother* for *Sofia Smith*

To get the required data, the relationship must be used; thus, the defined *person_rec* table must be used twice and joined. For simplicity, imagine the table as two separate tables connected using a non-identifying relationship:

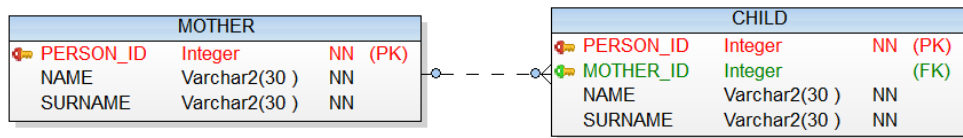


Fig. 8.15: Mother, Child table model

```
select m.name, m.surname
  from person_rec m join person_rec c on (m.person_id = c.mother_id)
 where c.surname = 'Smith' and c.name = 'Sofia';
```

The result should be “*Olivia Clarke*”:

Task 2: Get the name of the people, whose *mother* is *Sofia Smith*:

```
select c.name, c.surname
  from person_rec m join person_rec c on (m.person_id = c.mother_id)
 where m.surname = 'Smith' and m.name = 'Sofia';
```

NAME	SURNAME
Jacob	Robinson
William	Robinson

8.9 Using the same table multiple times in the Select statement

Tables can be linked together using various relationship types. Moreover, several relationships can be created between two tables (do you remember the table *flight* and *airport* from the data modeling lab, don’t you?). If the *departure airport*, as well as the arrival airport (names of the airports), should be listed for the defined *flight*, it is necessary to join table *airport* twice – one join expresses *departure*, the second one reflects the *arrival*.

Let’s have the following example for dealing with the *address*. *Town*, *region*, and *state* have been separated into separate tables as a result of *data normalization*. Thus, for any *person*, *airport*, or *company*, the current address (*street*) is stored with reference to town (e.g., *act_zip*).

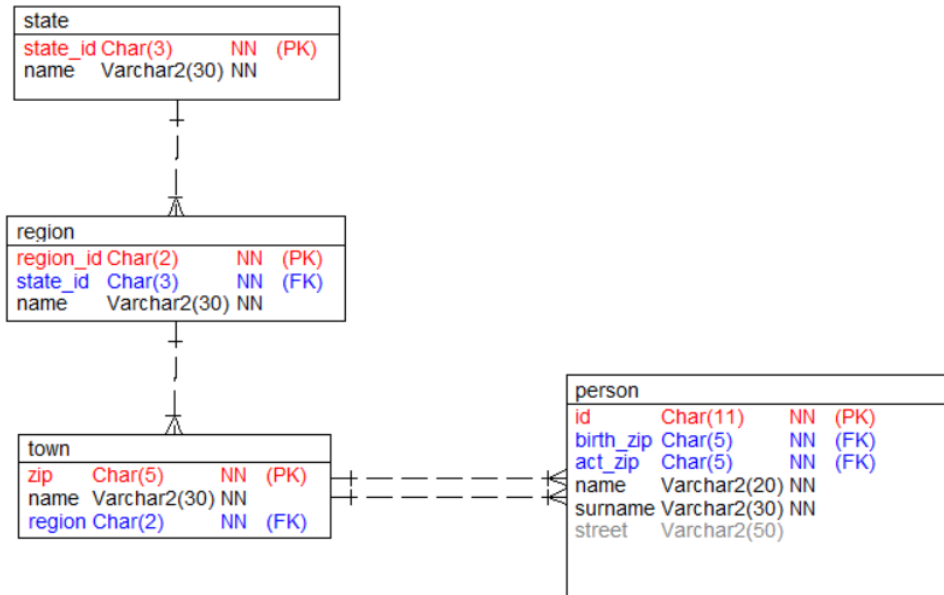


Fig. 8.16: Multiple relationships between Person and Town table

If you want to get the name list of the people *living in the same town as they were born*, attributes `birth_zip` and `act_zip` can be compared.

```
select name, surname
  from person
 where birth_zip = act_zip;
```

The previously defined statement will work; however, it will not provide desired data based on the task. Once again, the aim is to get a name list of the people living in the same town as they were born. Do not forget that multiple `zip` codes can delimit one town, so the name of the towns must be compared (one name of the towns can be used numerous times in the world. Therefore, the comparison is based on `region` as well). In the following example, tables must be aliased (whereas one table will be used multiple times), ***birth_town*** table alias expresses the town of person birth, ***act_town*** table alias defines the actual town, where the person is living.

```
select name, surname
  from person join town birth_town on (person.birth_zip = birth_town.zip)
        join town act_town on (person.act_zip = act_town.zip)
 where birth_town.name = act_town.name
        and birth_town.region = act_town.region;
```

Can `Join` clause type `USING` be used? If not, why? If so, under what conditions?

8.10 Practice

1. Get the *age* of the oldest student at the time of leaving school (use the attribute ***final_date*** in the student table).
2. List the name of the students who will celebrate a *birthday next month* (use actual *system time*).

3. List the name of the *students* who will celebrate the *anniversary* this year (e.g., 25, 30, 35, ... years old).
4. Get the following statistics for each *student* – the *best result*, *worst result*, and the *total number* of registered subjects in the *school year 2008*.
5. Get the *name list of the students* who have the *grade average* better than 3. At first, transfer the character value to the particular coefficient. If the subject is failed, it should be considered as the result 4.
 - A = 1
 - B = 1.5
 - C = 2
 - D = 2.5
 - E = 3
 - F, Fx = 4
6. List the names of the *subjects*, which *at least 4 students* have registered in the *school year 2006*.
7. List the name of the *students* who have *repeated some subjects*. Each student should be listed *only once* in the result set.
8. List the name of the *students* with the *total number of subjects* registered in the *school year 2008*.
9. List the name of the *subjects* which *have not been registered by any student* in the *school year 2006* but were available (particular subject can be found in the *st_program* table).
10. Get the *number of days* between the *sign_date* and *exam_date* for the subjects with accreditation and exam (get the required information from *ending_type* parameter) in the *subject_year* table.
 - subject_year.ending_type:
 - B = exam + accreditation to exam,
 - E = exam,
 - S = semester only (no exam).
11. List the name of the *students* and their *subjects*, which end with *accreditation to the exam* and an *exam*. List only those where the *difference* between *accreditation* and *exam* was *at least 1 month*.
12. List the name of the *students* who have *never repeated any subject*.
13. List the name of the *students* who have *at least one subject*, which the particular student has not *repeated*.
14. List the total number of *students* in each *class*.
15. List the total number of *students* for each *class* and each *study field* and *specialization*.
16. Get the list of the *optional* subjects for a student with *student_id* = 500439. The optional subject can be identified by the absence of *mandatory* or *mandatory-optional* subject sets with an emphasis on the *study field* and *specialization* of the *student*. However, be aware, if the *subject* is *valid* for the whole *study field* (regardless of the *specialization*), the particular *specialization_id* in the *st_program* table will have the value "0".
 - st_program.mandatory_type:
 - M = mandatory,
 - O = optional,
 - X = mandatory / optional.

17. List the name of the students who have never studied Information systems (**field_name**).
18. List the name of the *people* with textual information about their *student status*.
If the person is not a student, write three dashes (---).
 - student.status:
 - S = student (actual),
 - E = ended successfully,
 - A = aborted
 - X = fired due to disciplinary commission decision.
19. List the name of the mandatory subjects in the school year 2008 with the total number of registered students for them. If there is none registered, write there dash (-) symbol.

BONUS: Get the following statistics:

- Rows – individual study fields and specifications
- Columns – classes
- Cells – the total number of particular students

Advice: individual study fields and specifications will be in the group by section, numbers should be calculated conditionally.

	<i>FIELD</i>	<i>1.CLASS</i>	<i>2.CLASS</i>	<i>3.CLASS</i>	<i>4.CLASS</i>	<i>5.CLASS</i>
1	Information systems, Decision support systems	1	0	0	0	0
2	Computer engineering,	0	1	0	0	0
3	Computer engineering,	1	2	0	0	0
4	Management,	0	2	1	0	0
5	Information management,	1	0	1	0	0
6	Informatics,	2	4	1	0	0
7	Information systems, Applied informatics	1	4	0	0	0
8	Information systems, Information and communication systems	0	5	0	0	0

Fig. 8.17: Required data layout

Lab 9 – Procedures, functions and packages

In this lab, the reader will get a complex overview of the procedural extension of the SQL language (PL/SQL). It focuses on the named data blocks (procedures and functions), which can be optionally grouped into the packages and managed as one unit. Vice versa, anonymous blocks are executed only once with no consecutive evidence in the system nor the repeated reference opportunity.

This lab summarizes the code preliminaries – variable definitions, assignments, NULL handling, conditional processing and loops. Reader will get the complex overview. All principles are explained in the examples.

Both procedures and functions can be executed from the block or by using the EXEC command. If some prerequisites are passed, function can be executed using SQL, as well. The reader will learn three types of passing parameters – position way, named reference, and hybrid. Output can sent to the data structures, variables, etc. or can be printed to the console. It is commonly done using the methods of the DBMS_OUTPUT package.

In this chapter, the reader will learn how to access table data inside the block. The output of the Select statement must be stored into variables. Using Select ... Into variant, it must be ensured that the particular statement returns one row (no more, no less). Otherwise, the exception is raised.

The cursor provides the general solution, which associates the data with variables using the loop. The reader will be provided with various cursor types. For the purposes of this book, we will, however, just focus on the static cursor types.

9.1 Introduction

SQL itself is a non-procedural language – we define data we want to, but not how to get them. Therefore, complex usage in an application environment is ensured only in combination with procedural language or specific procedural database language, often referred to as fourth-generation language. In this lab, we will shortly introduce the syntax definition, specific clauses, and notations regarding standalone *procedures* and *functions*, but also *packages*, which can group multiple methods into the common class. *Procedures* and *functions* are created using PL/SQL (*Procedural Language of SQL*) and are called by their names. The difference between *procedure* and *function* is based on the possibilities of returning values. *The function* must have only one return value (which can also be composite), whereas the *procedure* cannot return any value by its name definition. The only way the procedure can return values are output (*IN OUT* or *OUT*) parameters. As we will describe later, if the *function* meets the essential prerequisites, it can be called from the *Select* statement (we have already dealt with the *to_char* conversion function, for example).

A significant advantage of the *package* is the possibility to group methods together. Privileges *granting* and *revoking* is done on *package* level instead of single methods (it is not possible to grant a privilege only to the particular method of the package). Moreover, the packaged method can be overloaded, which is not feasible for standalone methods.

A particular case of the block is an anonymous type. The principles are the same as a standalone procedure. However, such a block cannot pass parameters and is not stored

in the database. Thus, after its execution, it is removed from the system and cannot be called anymore.

9.2 Code preliminaries

In the following part, we will describe the essential code characteristics.

9.2.1 Variable definition

When managing *procedures* and *functions* (optionally enclosed by the *packages*), it is usually necessary to define *local variables* for the execution. Variable must have some *data type*, which can also be based on the table attribute data type, the row of the table, or the result of the *cursor*. It can be initialized to a specific value during definition or even be noted as constant (there is no possibility of changing its value later).

We strongly recommend not to use the same name of the variable as the table attribute name, mainly during the execution of the *Select* statement, whereas it can result in getting incorrect data (table attribute has a higher priority than variable). Therefore, we prefer the standard of naming – the first letter of the variable is “v”. In that case, there cannot be any misunderstanding.

Variable definition:

```
Variable data_type;
```

Definition and value initialization of the variable:

```
Variable data_type := init_value;
```

Defined variable has the same structure as the schema of the table (reflects the data types and attribute names):

```
Variable_record table%rowtype;
```

Defined variable data type will correspond table attribute data type:

```
Variable table.attribute%type;
```

Example of usage is following:

```
v_pid char(11); -- 11 character string
v_student_id integer := 1; -- integer variable, value 1 is assigned
                        -- during the definition
v_student_data student%rowtype; -- variable is record, has the same
                        -- elements as table attributes
v_surname personal_data.surname%type; -- data type of the variable is
                        -- the same as data type
                        -- for attribute surname of
                        -- the table personal_data
```

9.2.2 Assignment, NULL

Assignment inside the body of the expression value to the variable is done by the symbol of a colon followed by the equality sign (:=).

```
Variable := expression;
```


And also some examples. The result can be direct value, expression, or function result.

```
v_count := 10;
```

```
v_str := to_char(sysdate, 'DD. Month YYYY');
```

No command block can be empty. Therefore another command has been introduced, which, however, does not execute anything (*NULL*). It is primarily intended for testing purposes, to cover branches for which we do not want to run any command. It can also be placed in *LOOP*s and conditional processing (*IF*), with at least one command inside each processing line.

```
NULL;
```

Naturally, each created code can be rewritten so that this command will not be used at all.

9.2.3 Conditional processing

Branching of the execution code can be done using *IF – THEN – ELSE* command type or by using *CASE*.

IF condition

IF condition starts with the keyword “*IF*” followed by the condition, which should be evaluated as *Boolean* (*True*, *False*). After that, the keyword “*THEN*” is used and the commands of the positive branch. The negative branch is optional, characterized by the “*ELSE*” keyword. Do not forget to add the “*END IF*” keyword after the command itself to border it.

```
IF condition THEN
  Commands_to_be_executed;
[ELSE
  Commands_else_clause;]
END IF;
```

The syntax mentioned above uses only one, respectively, two branches. However, it can be extended by multiple branches forming more complex conditions. In this case, the negative branch is divided into several *IF* conditions using the keyword “*ELSIF*” (it is one word, character “e” is omitted). Thus, there can be several “*ELSIF*” conditional branches in one “*IF*” condition.

Another solution is to divide the “*ELSE*” branch of the condition into several conditions, which must also be enclosed by the “*END IF*” keyword.

```
IF condition THEN
  Commands_if;
ELSIF condition2 THEN
  Commands_elsif;
[ELSE
  Commands_else;]
END IF;
```

Therefore, notice a significant difference between “*ELSIF*” (part of the same *IF* condition) and “*ELSE IF*” (forming new *IF* condition). Each *IF* branching command must have pair – “*END IF*” keyword.

```

IF condition THEN
  Commands;
ELSE
  IF condition2 THEN
    Commands;
  [ELSE
    commands;]
  END IF;
END IF;

```

Condition-based on using *IF* conditions *cannot* be used in SQL statements like *Insert*, *Update*, *Delete*, *Select*. However, do not be scared, **CASE** commands can replace them.

In the previous lines, we explained that the result of the condition to be evaluated must be *Boolean*. But to be honest, it also reflects *NULL*, which can cause significant problems because of the 3 valued logic and evaluation.

It is important to remember that any condition with *NULL* expression is evaluated as *NULL* and led to the *ELSE* branch. Therefore, never compare values to *NULL* value using the equality sign (=). Problems are shown in the following example. Let's have the uninitialized variable and *IF* condition comparing the defined variable. First of all, ensure that variable with no explicit value assignment is treated as *NULL*. For these purposes, we declare simple integer variable and *IF* condition for comparison. Make sure that comparison is made using "*IS NULL*". *Put_line* method of the *dbms_output* package buffers data from the parameter and sends them to the console output. More about the *dbms_output* package can be found in chapter [9.5 Executing stored method](#).

```

declare
  v_int integer;
begin
  if (v_int is null) then
    dbms_output.put_line('v_int IS NULL');
  else
    dbms_output.put_line('v_int is NOT NULL');
  end if;
end;
/

```

```
v_int IS NULL
```

```
PL/SQL procedure successfully completed.
```

Notice that the output display must be enabled to see the results. The *SERVEROUTPUT* setting controls whether SQL*Plus prints the output generated by the *dbms_output* package from PL/SQL procedures. It must be enabled for the session (or for the whole server) before the first execution of the *dbms_output* package (see chapter [9.5 Executing stored method](#)). Otherwise, no output will be printed to the user.

```
set serveroutput on
```

Set serveroutput on executed in SQL Plus executes behind the scene following command (see chapter [9.5.2 Enable procedure](#)):

```
dbms_output.enable(buffer_size => NULL);
```

Vice versa, to disable output printed to the screen, *Set serveroutput off* can be used, which reflects the calling *disable* procedure of the *dbms_output* package (see chapter 9.5.1 Disable procedure).

```
set serveroutput off
```

```
dbms_output.disable;
```

It is not possible to evaluate *NULL* value using the equality sign. As we can see, regardless of the equality or even non-equality character, the condition is always routed to the *ELSE* branch (the result of the condition is *NULL*).

```
declare
  v_int integer;
begin
  if (v_int = NULL) then
    dbms_output.put_line('v_int = NULL');
  else
    dbms_output.put_line('v_int != NOT NULL');
  end if;
end;
/
```

```
v_int != NOT NULL
```

```
PL/SQL procedure successfully completed.
```

```
declare
  v_int integer;
begin
  if (v_int != NULL) then
    dbms_output.put_line('v_int = NULL');
  else
    dbms_output.put_line('v_int != NOT NULL');
  end if;
end;
/
```

```
v_int != NOT NULL
```

```
PL/SQL procedure successfully completed.
```

Notice that incorrect management of *NULL* values is a source of very severe problems, and it is really very hard to find the reasons and solve the issue. Therefore, be strictly attentive.

Moreover, also variables cannot be compared to nullity using the equality sign to get information, whether such variables are the same or not. Simply, two *NULL* values are never the same! Thus, let's have the anonymous block containing two variable definitions, which gets the *NULL* values. Once again, such defined condition is always evaluated as *NULL* and routed to the *ELSE* branch.

```
declare
  v_int1 integer;
  v_int2 integer;
begin
  if (v_int1 = v_int2) then
    dbms_output.put_line('are the same');
  else
    dbms_output.put_line('are NOT the same');
  end if;
end;
/
```

are NOT the same

PL/SQL procedure successfully completed.

```
declare
  v_int1 integer;
  v_int2 integer;
begin
  if (v_int1 != v_int2) then
    dbms_output.put_line('are NOT the same');
  else
    dbms_output.put_line('are the same');
  end if;
end;
/
```

are NOT the same

PL/SQL procedure successfully completed.

Also, a comparison based on mathematical operators *lower than* (<) or *higher than* (>) can cause problems with **NULL** values. Its evaluation always ends in the **ELSE** branch with strange results. The first evaluation result is that variable **v_int1** is lower, and the second one evaluates variable **v_int1** as higher than variable **v_int2**.

```
declare
  v_int1 integer:=1;
  v_int2 integer;
begin
  if (v_int1 > v_int2) then
    dbms_output.put_line('v_int1 is higher than v_int2');
  else
    dbms_output.put_line('v_int1 is lower than v_int2');
  end if;
end;
/
```

v_int1 is lower than v_int2

PL/SQL procedure successfully completed.

```

declare
  v_int1 integer:=1;
  v_int2 integer;
begin
  if (v_int1 < v_int2) then
    dbms_output.put_line('v_int1 is lower than v_int2');
  else
    dbms_output.put_line('v_int1 is higher than v_int2');
  end if;
end;
/

```

v_int1 is higher than v_int2

PL/SQL procedure successfully completed.

To have clear evidence, compare the evaluated condition based on “*IS NULL*”. As you can see, it has been evaluated as *NULL*.

```

declare
  v_int1 integer;
  v_int2 integer;
begin
  if ((v_int1 > v_int2) is NULL) then
    dbms_output.put_line('treated as NULL');
  else
    dbms_output.put_line('NOT treated as NULL');
  end if;
end;
/

```

treated as NULL

PL/SQL procedure successfully completed.

Fig. 9.1 shows the evaluation table of the conditions based on *NULL* values with an emphasizing *AND*, *OR*, and inversion (*NOT*) condition grouping.

<i>AND</i>	<i>TRUE</i>	<i>FALSE</i>	<i>NULL</i>
<i>TRUE</i>	TRUE	FALSE	NULL
<i>FALSE</i>	FALSE	FALSE	FALSE
<i>NULL</i>	NULL	FALSE	NULL

<i>OR</i>	<i>TRUE</i>	<i>FALSE</i>	<i>NULL</i>
<i>TRUE</i>	TRUE	TRUE	TRUE
<i>FALSE</i>	TRUE	FALSE	NULL
<i>NULL</i>	TRUE	NULL	NULL

<i>NOT</i>	
<i>TRUE</i>	FALSE
<i>FALSE</i>	TRUE
<i>NULL</i>	NULL

Fig. 9.1: Three-value logic

Condition CASE

There are two possibilities, how to deal with the *CASE*. The left part of the condition can be covered on the top level (in that case, it is then compared based on only the values themselves), or the conditions can be expressed on each level. It starts with the “*CASE*” keyword and ends with “*END CASE*” for a procedural language. When using in SQL, the keyword “*CASE*” is omitted from the end. Thus, there is “*END*”, not “*END CASE*”.

Moreover, in SQL value is evaluated and processed, there are no commands after the *Then* keyword.

```
CASE value_of_condition
  WHEN value1 THEN commands1;
  WHEN value2 THEN commands2;
  ...
  [ELSE commands_else];
END CASE;
```

```
CASE
  WHEN condition1 THEN commands1;
  WHEN condition2 THEN commands2;
  ...
  [ELSE commands_else];
END CASE;
```

The “*ELSE*” clause is optional. However, if you do not define it explicitly, the database system manager will automatically add it by raising an *exception* if no suitable processing branch is found:

```
ELSE RAISE CASE_NOT_FOUND;
```

Thus, create a simple anonymous block (all principles of the anonymous block will be described a bit later, we will now highlight only *CASE* usage), declare two variables, and see the principles. The first variable (*v_personal_id*) will deal with the *personal_id*, the second (*v_month*) will be the extraction of the birth month, and string text will be shown as output. In this case, we use the first *CASE* principle – the value of the *v_month* is compared with values covered in individual “*WHEN*” parts.

```
declare
  v_personal_id char(11);
  v_month integer;
begin
  v_personal_id := '690309/1234';
  v_month := substr(v_personal_id, 3, 2);
  case v_month
    when 1 then dbms_output.put_line('January');
    when 2 then dbms_output.put_line('February');
    when 3 then dbms_output.put_line('March');
    when 4 then dbms_output.put_line('April');
    when 5 then dbms_output.put_line('May');
    when 6 then dbms_output.put_line('June');
    when 7 then dbms_output.put_line('July');
    when 8 then dbms_output.put_line('August');
    when 9 then dbms_output.put_line('September');
    when 10 then dbms_output.put_line('October');
    when 11 then dbms_output.put_line('November');
    when 12 then dbms_output.put_line('December');
  end case;
end;
/
```

The second *CASE* principle will look like the following example. The whole condition is in each “*WHEN*” part.

```

declare
  v_personal_id char(11);
  v_month integer;
begin
  v_personal_id := '690309/1234';
  v_month := substr(v_personal_id, 3, 2);
  case
    when v_month=1 then dbms_output.put_line('January');
    when v_month=2 then dbms_output.put_line('February');
    when v_month=3 then dbms_output.put_line('March');
    when v_month=4 then dbms_output.put_line('April');
    when v_month=5 then dbms_output.put_line('May');
    when v_month=6 then dbms_output.put_line('June');
    when v_month=7 then dbms_output.put_line('July');
    when v_month=8 then dbms_output.put_line('August');
    when v_month=9 then dbms_output.put_line('September');
    when v_month=10 then dbms_output.put_line('October');
    when v_month=11 then dbms_output.put_line('November');
    when v_month=12 then dbms_output.put_line('December');
  end case;
end;
/

```

Do you consider the solutions to be correct? Why not? We do not care about women, do we? What will happen if we change the *personal_id* characterizing woman? Naturally, an exception will be raised:

```

ERROR at line 1:
ORA-06592: CASE not found while executing CASE statement
ORA-06512: at line 7

```

Thus, for women, get the month's value by subtracting the third and fourth value of the *personal_id* by 50 (we will use the *mod* function in the example to create a universal solution).

```

declare
  v_personal_id char(11);
  v_month integer;
begin
  v_personal_id := '695309/1234';
  v_month := mod(substr(v_personal_id, 3, 2), 50);
  case
    when v_month=1 then dbms_output.put_line('January');
    when v_month=2 then dbms_output.put_line('February');
    when v_month=3 then dbms_output.put_line('March');
    when v_month=4 then dbms_output.put_line('April');
    when v_month=5 then dbms_output.put_line('May');
    when v_month=6 then dbms_output.put_line('June');
    when v_month=7 then dbms_output.put_line('July');
    when v_month=8 then dbms_output.put_line('August');
    when v_month=9 then dbms_output.put_line('September');
    when v_month=10 then dbms_output.put_line('October');
    when v_month=11 then dbms_output.put_line('November');
    when v_month=12 then dbms_output.put_line('December');
  end case;
end;
/

```

Now, the solution is correct, however not so robust. What about typos in *personal_id* value? Naturally, an exception will be raised. To avoid it, add the “*ELSE*” branch of the case.

```

declare
  v_personal_id char(11);
  v_month integer;
begin
  v_personal_id := '699309/1234';
  v_month := mod(substr(v_personal_id, 3, 2), 50);
  case
    when v_month=1 then dbms_output.put_line('January');
    when v_month=2 then dbms_output.put_line('February');
    when v_month=3 then dbms_output.put_line('March');
    when v_month=4 then dbms_output.put_line('April');
    when v_month=5 then dbms_output.put_line('May');
    when v_month=6 then dbms_output.put_line('June');
    when v_month=7 then dbms_output.put_line('July');
    when v_month=8 then dbms_output.put_line('August');
    when v_month=9 then dbms_output.put_line('September');
    when v_month=10 then dbms_output.put_line('October');
    when v_month=11 then dbms_output.put_line('November');
    when v_month=12 then dbms_output.put_line('December');
    else dbms_output.put_line('Unknown month...');
  end case;
end;
/

```

Unknown month...

PL/SQL procedure successfully completed.

In the previous case, the solution provides correct results and is exception prove. Another situation will, however, occur if you want to process the age of the person. Extracting *value_of_condition* from the “*WHEN*” branch does not provide sufficient power. Why? It would be necessary to name all possible values separately (for simplicity, we declare variable *v_age* defining a transformation from the *personal_id* value):

```

declare
  v_age integer;
begin
  v_age := 19;
  case v_age
    when 1 then dbms_output.put_line('Child');
    when 2 then dbms_output.put_line('Child');
    when 3 then dbms_output.put_line('Child');
    -- ...
    when 18 then dbms_output.put_line('Adult');
    when 19 then dbms_output.put_line('Adult');
    -- ...
    else dbms_output.put_line('???');
  end case;
end;
/

```

It is too complicated and hard-coded, isn't it? If you create the second *CASE* type, comparison functions can be used, so the solution can be like following:


```

declare
  v_age integer;
begin
  v_age := 19;
  case
    when v_age between 0 and 17 then dbms_output.put_line('Child');
    when v_age > 18 then dbms_output.put_line('Adult');
    else dbms_output.put_line('???');
  end case;
end;
/

```

Significantly easier, isn't it? So, which *CASE* type is better? The answer is a bit tricky. Simply, it depends. One side of the issue is just the simplification of the code for the programmer. The second aspect is, however, just the performance. If you use the complex function, forming the whole condition requires evaluating it for each branch. It can be time and resource-demanding. It can be partially solved by variable definition for storing function result. Thus, each time, think of the consequences and try to optimize your code.

9.2.4 LOOPS

Database systems provide us multiple tools for repeated processing of the same code using LOOPS. In principle, we can distinguish these types:

Infinite loop, EXIT condition

```

LOOP
  Commands;
END LOOP;

```

In this case, there must be some condition, which will force the system to terminate cycle processing and move to execute consecutive code parts. To do so, the command *EXIT* is used. So, if the condition (*condition*) in the *IF* command is evaluated as “*True*”, the *EXIT* command is executed, and *Loop* processing is terminated.

```

LOOP
  Commands1;
  IF condition THEN
    EXIT;
  END IF;
  Commands2;
END LOOP;

```

Also, another syntax is possible to be defined. It can be considered as a particular type of the *IF* command – *EXIT WHEN condition*. Functionality is the same.

```

LOOP
  EXIT WHEN condition;
END LOOP;

```

The example can look like the following. The body of the *Loop* is executed three times.

```
declare
  i integer;
begin
  i := 1;
  loop
    dbms_output.put_line(i);
    exit when i = 3;
    i := i + 1;
  end loop;
end;
/
```

WHILE loop type

The execution of the cycle is delimited by the *WHILE* condition, which is evaluated at the beginning of each round of the cycle.

```
WHILE condition LOOP
  Commands;
END LOOP;
```

The previous example rewritten using *While* is the following:

```
declare
  i integer;
begin
  i := 1;
  while i <= 3 loop
    dbms_output.put_line(i);
    i := i + 1;
  end loop;
end;
/
```

FOR loop type

The number of rounds in the cycle is directly delimited by the two numbers – starting value (*min*) and maximal value (*max*). Each round of the cycle automatically increases the *control loop variable* (*control_variable*). Such variable is defined *implicitly*, and its scope is terminated after the cycle execution.

```
FOR control_variable IN min..max LOOP
  Commands;
END LOOP;
```

It is strongly recommended not to name the *control loop variable* with the same name as the existing parameter or variable. This is because, inside the cycle, the highest priority has just *control variable*. However, always try to prevent possible problems by using correct name notations.

```
begin
  FOR i IN 1..3 LOOP
    dbms_output.put_line(i);
  END LOOP;
end;
/
```

Notice that the *control loop variable* cannot be changed in the loop body.

```
begin
  FOR i IN 1..3 LOOP
    i := 2; -- it is NOT possible
    dbms_output.put_line(i);
  END LOOP;
end;
/
```

There is also a particular case when the *control loop variable* starting position is delimited by the *maximal value*, and its value is sequentially *decreased* (“-1”) during the execution. This functionality can be provided by using the **REVERSE** keyword. However, the order limitations (*min*, *max*) is the same as in the standard approach. Thus, the minimum value is listed sooner:

```
FOR control_variable IN REVERSE min..max LOOP
  Commands;
END LOOP;
```

```
begin
  FOR i IN REVERSE 1..3 LOOP
    dbms_output.put_line(i);
  END LOOP;
end;
/
```

9.3 PL/SQL anonymous block

PL/SQL block is a sequence of the commands to be executed. In principle, the structure of such defined PL/SQL block can be divided into two groups based on the storage principles. The first group covers an anonymous block, which has no name. Thus, it cannot be referenced from any other blocks, as well as from functions or procedures. It is executed directly after its definition.

Here is a simple example of the anonymous block.

```
begin
  dbms_output.put_line('Welcome...');
end;
/
```

After the definition and execution of such a block, we have no evidence about the past existence and processing. Thus, if we want to execute it once again, it is necessary to code it again.

Next block shows the syntactical structure of the anonymous PL/SQL block:

```
[DECLARE
  Variable datatype [:= init_value];
]
BEGIN
  Commands;
  [EXCEPTION
    WHEN exception_type1 THEN commands;
    WHEN exception_type1 THEN commands;
    ...
  ]
END;
/
```

Each PL/SQL block consists of the body, which is mandatory. It can also contain the **DECLARATION** part (between keywords **DECLARE** and **BEGIN**) and **EXCEPTION** part at the end of the body (enclosed between keywords **EXCEPTION** and **END**).

If you want to refer to the local variable, it must be first defined in the first part – *declaration*. Usually, local variables are initialized to **NULL**. If there is any *exception* raised in the body of the block, it can be processed using the **EXCEPTION** part of the body. The order of the clauses in the **EXCEPTION** clause is critical – if it finds a suitable **EXCEPTION** branch covering such a problem, it will not check later **EXCEPTION** branches. Thus, it is processed by the first condition it meets. A typical example is **OTHERS** as a type of the **EXCEPTION**, which covers all raised exceptions. Thus, none later defined can be processed at all.

The second block type is stored definition – procedure/function.

9.4 Procedure, function

If you want to store PL/SQL block for later referencing and calling, it must be named and stored as a *procedure* or *function*. In the following text, we will use the term **method** covering both procedures and functions. The main difference between them is the one value, which is returned using the function by its name. Thanks to that, the function can be used in the SQL statements (if some conditions are fulfilled, which will be described later).

9.4.1 Procedure syntax

```

CREATE [OR REPLACE] PROCEDURE proc_name
[(parameter1 [mode1] datatype1,
  parameter2 [mode2] datatype2,
  ...)]
IS | AS
[variable1 datatype1 [:= init_value];
 variable2 datatype2 [:= init_value];
 ...]
BEGIN
  Commands;
  [EXCEPTION
    WHEN exception_type1 THEN commands;
    WHEN exception_type2 THEN commands;
    ...
  ]
END [proc_name];
/

```

The keyword “**OR REPLACE**” is optional. However, very convenient, if you want to edit the structure of the method. Thanks to that, it is not necessary to *drop* it and *create* a new one. However, if it is defined without this clause, it cannot be replaced at all. It must be simply dropped and recreated.

The method can have multiple parameters. Some of them can be optional (in that case, the appropriate value must be obtained inside the procedure or replaced using the **DEFAULT** value). Be aware. Parameter data types *DO NOT* contain the size of the string format. Therefore, there is no *CHAR(10)* or *VARCHAR(10)*, but there is only data type definition – **CHAR**, **VARCHAR**. Additionally, each parameter can be delimited by its mode.

There are three types to be recognized:

- **IN** (default mode) – input parameter. It passes parameter from the calling environment to method execution (*constant* or *variable* value). Inside the method, it is noted as *constant*. If attempting to change will cause raising an error.
- **OUT** – output parameter. It passes a value to the calling environment code. Therefore, the output parameter must be associated with a variable, not constant.
- **IN OUT** – input-output parameter. Similar to output parameter mode – it must be associated with the variable. It passes input value from the environment, which can be optionally (and even usually) changed during execution.

Tab. 9.1: Method parameters

IN	OUT	IN OUT
can be represented by a variable (commonly) or constant	must be specified by the variable	must be specified by the variable
formal parameter works like constant	works like a non-initialized variable	works like an initialized variable

You can choose which keyword you will use (**IS** or **AS**). After this keyword, there is a list of the local variables.

In the procedure body, the keyword **RETURN** can be used as well, but there is no value connected to the procedure name to be processed as output. Using it means that no other code

after it will be processed inside the procedure at all. Thanks to that, it provides simpler mechanisms to end processing.

9.4.2 Function syntax

```
CREATE [OR REPLACE] FUNCTION func_name
[(parameter1 [mode1] datatype1,
 parameter2 [mode2] datatype2,
 ...)]
RETURN datatype
IS | AS
[variable1 datatype1 [:= init_value];
 variable2 datatype2 [:= init_value];
 ...]
BEGIN
  Commands;
  RETURN expression;
[EXCEPTION
  WHEN exception_type1 THEN commands;
  WHEN exception_type2 THEN commands;
  ...
]
END [func_name];
/
```

The difference between *procedure* is just the *Return* clause characterizing the value to be returned by the function name based on the defined data type specified in the function header.

Be aware, after raising the **RETURN** keyword, no more code is processed, and the management is returned to the calling environment. In function processing, be sure that each branch is associated with the **RETURN** keyword.

In addition, the result of the function cannot be thrown away but must be assigned to a variable, respectively, as a parameter to another function or procedure. The result can also be used by the *Select* statement if some requirements are met.

To allow users to create (any) procedures and functions, the appropriate privilege must be granted (this privilege does not differentiate between procedures and functions, thus if it is granted, a particular user can create procedures and functions, too):

```
GRANT CREATE ANY PROCEDURE TO username;
```

You must also distinguish another privilege, which allows the user to *execute* defined procedure:

```
GRANT EXECUTE ON procedure_name TO username;
```

Notice that if you want to deal with the *Select* statement in the PL/SQL block, the results must be stored (using **SELECT INTO**) or processed using *cursors*. **If the *Select Into* type is used, exactly one row must be returned.** Otherwise, an exception will be raised.

Let's have the following example. It returns name and surname in one string as well as a group of the student delimited by the *p_st_id* parameter.

```

create or replace procedure query_student_proc
(p_st_id IN student.student_id%type,
 p_name OUT varchar2,
 p_group OUT student.st_group%type)
is
begin
  select name || ' ' || surname, st_group into p_name, p_group
    from personal_data join student using(personal_id)
   where student_id = p_st_id;
end;
/

```

9.5 Executing stored method

Let's have the previous procedure defined. To execute it, first of all, whereas two parameters are **OUT** mode, it is necessary to define variables in the SQL environment. Variable definition is provided using the *variable* keyword of the SQL. Notice that there is no information for you after variable creation in the system.

```

variable v_name varchar2(30)
variable v_group char(6)

```

Then, it is possible to execute that procedure. But be aware that SQL variable name must be prefixed with a colon (:) when calling using procedure.

```
EXECUTE query_student_proc(501567, :v_name, :v_group);
```

After successful execution, you will get the following information:

```
PL/SQL procedure successfully completed.
```

On the other hand, when you want to write the value of the SQL variable to the console, there is no colon prefix:

```
PRINT v_name v_group
```

And this is the output:

```

V_NAME
-----
William Whittel

V_GROUP
-----
5ZI000

```

As already mentioned, any PL/SQL block can call a stored data block (*procedure*, *function*). Let's have the following example – calling the procedure using anonymous block.

Calling defined procedure from the anonymous block looks like the following example:

```
declare
    v_name varchar2(30);
    v_group char(6);
begin
    -- execute procedure
    query_student_proc(501567, v_name, v_group);
    -- get the variable values and show them on the console screen
    dbms_output.put_line('Name: ' || v_name);
    dbms_output.put_line('Group: ' || v_group);
end;
/
```

Values shown on the display are following:

```
Name: William Whittel
Group: 5ZI000
```

Realize that there is no “*execute*” keyword when calling the procedure from the PL/SQL block.

At this point, other methods of the *dbms_output* package should be explained, which are mostly used. Package *dbms_output* allows you to send messages from the blocks (anonymous, procedure, function, trigger) to the console output. It is beneficial for displaying PL/SQL debugging information.

Package *dbms_output* contains these methods:

Tab. 9.2: Methods of the *dbms_output* package

<i>Subprogram</i>	<i>Description</i>
DISABLE procedure	Disables message output
ENABLE procedure	Enables message output
GET_LINE procedure	Retrieves one line from the buffer
GET_LINES procedure	Retrieves an array of lines from the buffer
NEW_LINE procedure	Terminates a line created with the PUT method
PUT procedure	Places a partial line in the buffer
PUT_LINE procedure	Places line in the buffer

Source: https://docs.oracle.com/database/121/ARPLS/d_output.htm#ARPLS67312

9.5.1 Disable procedure

This procedure disables calls to *Put*, *Put_line*, *New_line*, *Get_line*, and *Get_lines* and purges the buffer of any remaining information.

```
DBMS_OUTPUT.DISABLE;
```

9.5.2 Enable procedure

This procedure enables calls to *Put*, *Put_line*, *New_line*, *Get_line*, and *Get_lines*. Calls to these procedures are ignored if the *dbms_output* package is not activated.

```
DBMS_OUTPUT.ENABLE(buffer_size IN INTEGER DEFAULT 20000);
```

Buffer_size parameter is an upper limit (in bytes) expressing the amount of buffered information. A *NULL* value means no limit.

9.5.3 Get_line procedure

This procedure retrieves a single line of buffered information.

```
DBMS_OUTPUT.GET_LINE
(
  line      OUT VARCHAR2,
  status    OUT INTEGER
);
```

Line parameter returns a single line of buffered information, excluding a final newline character. The maximal length for the parameter is 32767 (*VARCHAR2* (32767)) limited by potentially raised exception *ORA-06502: PL/SQL: numeric or value error: character string buffer too small*.

Status parameter expresses the evaluation result of the procedure. If the call completes successfully, then the returned value is 0. Otherwise, the *status* will hold value 1.

9.5.4 Get_lines procedure

This procedure retrieves an array of lines from the buffer.

```
DBMS_OUTPUT.GET_LINES
(
  lines      OUT   CHARARR,
  numlines   IN OUT INTEGER
);
```

```
DBMS_OUTPUT.GET_LINES
(
  lines      OUT   DBMSOUTPUT_LINESARRAY,
  numlines   IN OUT INTEGER
);
```

9.5.5 New_line procedure

This procedure puts an end-of-line marker. As a result, the content of the buffer is produced to the console output.

```
DBMS_OUTPUT.NEW_LINE;
```

9.5.6 Put procedure

This procedure places a partial line in the buffer.

```
DBMS_OUTPUT.PUT(item IN VARCHAR2);
```

Item parameter holds the item to buffer.

The function **put** works a bit differently as we know from the other programming language, where data are shown, but there is no *line spacing*. In PL/SQL, the principle is different. After the calling function **put**, data are *only buffered but not written to display at all*. This is done just after calling **put_line** or **new_line** methods, which flush the buffer by displaying data.

Let's have the following example. Function **put** is used. In the first case, no output is printed. The second solution is extended by calling the **new_line** method causing data to be displayed. Thus, be aware of it. Otherwise, no data will be written to the output.

```
begin
  dbms_output.put('Hello');
end;
/
```

PL/SQL procedure successfully completed.

```
begin
  dbms_output.put('Hello');
  dbms_output.new_line();
end;
/
```

Hello

PL/SQL procedure successfully completed.

9.5.7 Put_line procedure

This procedure places a line in the buffer and sends it to the console output.

```
DBMS_OUTPUT.PUT_LINE(item IN VARCHAR2);
```

Internally, it calls the *new_line* procedure.

Notice that the maximum *line* size is 32767 bytes. The *default buffer size* is 20000 bytes. The minimum size is 2000 bytes, and the maximum is *unlimited*.

Following exceptions can be raised when dealing with buffers using the *dbms_output* package:

Tab. 9.3: Exceptions of the *dbms_output* package

Error	Description
ORU-10027	Buffer overflow
ORU-10028	Line length overflow

Let's move forward to the functions like having the following one. It reflects the easy solution for getting the total amount of gained *ects* of the defined *student*. Notice that for the real environment, data are not correct. Also, information about the *sign_date*, *result*, *exam_date*, and *ending_type* must be evaluated. Is it necessary to define exception handling? No, because aggregate function *SUM* will always return one row. In this case, if the student does not exist, a *NULL* value will be returned.

```
Create or replace function get_ects_count
(p_st_id student.student_id%type)
return number
is
  v_count study_subjects.ects%type;
begin
  select sum(nvl(ects, 0)) into v_count
  from study_subjects
  where student_id = p_st_id and result IN ('A', 'B', 'C', 'D', 'E');
  return v_count;
end get_ects_count;
/
```

There are three possibilities how to execute the function. Compared to the procedure, which offers two methods – calling from SQL or PL/SQL block, function definition

also allows calling from the queries (after passing some preliminary conditions). So, define two variables – `v_credit_count` and `v_st_id`.

```
variable  v_credit_count  number
variable  v_st_id         number
```

The variable assignment can be done by calling **Execute** command, like the following example (assignment of the value to the variable can be therefore considered as a special case of function calling):

```
EXECUTE :v_st_id := 501567;
```

The function itself is also called using the **Execute** command. Returned value is assigned into `v_credit_count` variable. The `v_st_id` variable provides an identifier of the student. However, whereas it is an input parameter, also constant can be used.

```
EXECUTE :v_credit_count := get_ects_count(:v_st_id);
```

Assigned values after function execution are obtained and written to the output using the **PRINT** command.

```
print v_credit_count
```

```
V_CREDIT_COUNT
-----
              19
```

Naturally, it is possible to call the function from any PL/SQL block (anonymous block, procedure, function).

```
declare
  v_count integer;
  v_student_id integer := 501567;
begin
  v_count := get_ects_count(v_student_id);
  dbms_output.put_line('Student ' || v_student_id || ' has ' ||
                      v_count || ' credits.');
```

```
end;
/
Student 501567 has 19 credits.
```

9.6 Calling function from the Select statement

The function can be called from the SQL statement if it meets some requirements. Notice that the procedure cannot be called at all, whereas it does not return any value from the definition (only *OUT* parameters can be used, however, such parameters cannot be used in SQL statements).

Limitation of the functions called from *DML* statements:

- function bodies cannot contain destructive *DML* statements (*Insert*, *Update*, *Delete*),
- functions called from *Update* or *Delete* statement cannot include any *DML* statement (*Insert*, *Update*, *Delete*, *Select*) referencing the same table,
- functions called from any SQL statement cannot contain (or generate) *TCL* statement (*Commit*, *Rollback*),

- also, calling any method from them is prohibited if any of the above rules are violated.

We have dealt with multiple functions called from SQL, like *to_char*, *to_date*, *round*, *trunc*, and many others previously. Now, there is an example of calling a user-defined function. Principles are the same in comparison with standard server functions.

```
select student_id, name, surname,
       get_ects_count(student_id) as credit_number
from personal_data join student using(personal_id)
where student_id = 501567;
```

STUDENT_ID	NAME	SURNAME	CREDIT_NUMBER
501567	Wiliam	Whittel	19

However, what will happen, if you do not specify the person for who the total number of credits should be summed?

Will the error be raised? If yes, why?

Select Into statement used in the PL/SQL block requires one significant property – result set of the statement must get exactly one row (no less, no more).

What will happen if the *Select* statement in the *procedure* or *function* does not return any data? Think of the following example.

```
declare
  v_count integer;
  v_student_id integer := null;
begin
  v_count := get_ects_count(v_student_id);
  dbms_output.put_line('Student ' || v_student_id || ' has ' ||
                       v_count || ' credits.');
```

A student cannot exist without an assigned *student_id* value. Although an exception will be raised, it will not be visible to the user.

```
Student has credits.
```

Another example shows a similar situation. There is no student with *student_id=1*.

```
declare
  v_count integer;
  v_student_id integer := 1;
begin
  v_count := get_ects_count(v_student_id);
  dbms_output.put_line('Student ' || v_student_id || ' has ' ||
                       v_count || ' credits.');
```

The **exception** will be raised. However, it will be “invisible”. Thus, please do not rely on it and ensure that no exception can be raised, respectively manage them.

The next chapter deals with **Exception** handling.

9.7 Exception handling

The optional *Exception* handler can extend each PL/SQL block.

```
[DECLARE
  Variable datatype[:=init_value];
]
BEGIN
  Commands;
[EXCEPTION
  WHEN exception_type1 THEN commands;
  WHEN exception_type2 THEN commands;
  ...
]
END;
/
```

As already highlighted, a subgroup of the exception is covered by the super-group. Therefore, the order of the *Exception* listed is critical.

System exceptions characterize the first category. The second category forms user-defined exceptions. The user-defined exception is raised explicitly if some specific situation (problem) occurs. However, do not replace standard conditions, code tracing, and standard management by raising *exceptions*. They should cover only special cases, not standard functionality.

User *exceptions* can be raised by two approaches. The first one is based on the **RAISE_APPLICATION_ERROR** method:

```
RAISE_APPLICATION_ERROR(error_code, error_description);
```

The *error_code* must belong to interval <-20999; -20000>. There is no need for special functionality to cover user-defined exceptions in the *Exception handler*.

Another approach is based on the *variable declaration, raising exception and error processing*:

- Declaration – exception type:

```
v_error_variable EXCEPTION;
```

- Exception raising:

```
RAISE v_error_variable;
```

- Error handling and processing:

```
EXCEPTION WHEN v_error_variable THEN commands;
```

So, let's have a simple example, on which we will demonstrate principles and opportunities.

```

declare
  v_name personal_data.name%type;
  v_surname personal_data.surname%type;
begin
  select name, surname into v_name, v_surname
    from personal_data
   where personal_id IS NULL;
  dbms_output.put_line(v_name || ' ' || v_surname);
end;
/

```

After launching such code, an exception will be raised – *Select Into* has returned no value:

```

ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line 5

```

Think of the previous example. If no row is returned in the *SELECT INTO* command in the PL/SQL block, an *exception* is raised. So, to solve such situation, add an *Exception* handler to manage that exception type:

```

declare
  v_name personal_data.name%type;
  v_surname personal_data.surname%type;
begin
  select name, surname into v_name, v_surname
    from personal_data
   where personal_id IS NULL;
  dbms_output.put_line(v_name || ' ' || v_surname);
EXCEPTION
  WHEN no_data_found THEN
    dbms_output.put_line('Row with such defined personal_id ' ||
                        'does not exist');
  WHEN others THEN
    dbms_output.put_line('Not covered error raised...');
end;
/

```

What about the results? *No_data_found* exception raised:

```

Row with such defined personal_id does not exist

PL/SQL procedure successfully completed.

```

However, what will happen, if you change the order in the *Exception* clause, “*others*” type will also cover the “*no_data_found*” exception type.

```

declare
  v_name personal_data.name%type;
  v_surname personal_data.surname%type;
begin
  select name, surname into v_name, v_surname
    from personal_data
    where personal_id IS NULL;
  dbms_output.put_line(v_name || ' ' || v_surname);
EXCEPTION
  WHEN others then
    dbms_output.put_line('Not covered error raised...');
  WHEN no_data_found then
    dbms_output.put_line('Row with such defined personal_id ' ||
                        'does not exist');
End;
/

```

The result is that it is even not possible to build it:

PLS-00370: OTHERS handler must be last among the exception handlers of a block

So, notice that “*Others*” cover all exceptions, thus should be placed last.

The following table shows the most commonly used and raised standard (system generated) errors. All of them with the description can be found in Oracle documentation.

Tab. 9.4: Exceptions

<i>Exception</i>	<i>Oracle Error</i>	<i>SQLCODE Value</i>	<i>Raised when</i>
ACCESS_INTO_NULL	ORA-06530	-6530	Your program attempts to assign values to the attributes of an uninitialized (atomically <i>NULL</i>) object.
CASE_NOT_FOUND	ORA-06592	-6592	None of the choices in the <i>WHEN</i> clauses of a <i>CASE</i> statement is selected, and there is no <i>ELSE</i> clause.
COLLECTION_IS_NULL	ORA-06531	-6531	Your program attempts to apply collection methods other than <i>EXISTS</i> to an uninitialized (atomically <i>NULL</i>) <i>nested table</i> or <i>varray</i> . The program attempts to assign values to the elements of an uninitialized <i>nested table</i> or <i>varray</i> .
CURSOR_ALREADY_OPEN	ORA-06511	-6511	Your program attempts to <i>open</i> an already <i>open cursor</i> . A <i>cursor</i> must be <i>closed</i> before it can be reopened. A <i>cursor FOR</i> loop automatically <i>opens the cursor</i> to which it refers. So, your program cannot open that <i>cursor</i> inside the loop.

<i>Exception</i>	<i>Oracle Error</i>	<i>SQLCODE Value</i>	<i>Raised when</i>
DUP_VAL_ON_INDEX	ORA-00001	-1	Your program attempts to store duplicate values in a database column that is constrained by a unique index.
INVALID_CURSOR	ORA-01001	-1001	Your program attempts an illegal <i>cursor</i> operation, such as closing an unopened <i>cursor</i> .
INVALID_NUMBER	ORA-01722	-1722	In a SQL statement, converting a character string into a number can generally fail because the string does not represent a valid number. (In procedural statements, <i>VALUE_ERROR</i> is raised.) This <i>exception</i> is also raised when the <i>LIMIT</i> -clause expression in a bulk <i>FETCH</i> statement does not evaluate a positive number.
LOGIN_DENIED	ORA-01017	-1017	Your program attempts to log on to Oracle with an invalid <i>username</i> and/or <i>password</i> .
NO_DATA_FOUND	ORA-01403	100	A <i>SELECT INTO</i> statement returns no rows, or your program references a deleted element in a <i>nested table</i> or an uninitialized element in an <i>index-by table</i> . SQL <i>aggregate functions</i> such as <i>AVG</i> and <i>SUM</i> always return a value or a <i>NULL</i> . So, a <i>SELECT INTO</i> statement that calls an <i>aggregate function</i> never raises <i>NO_DATA_FOUND</i> . The <i>FETCH</i> statement is expected to return no rows eventually, so when that happens, no exception is raised.
NOT_LOGGED_ON	ORA-01012	-1012	Your program issues a database call without being connected to Oracle.
PROGRAM_ERROR	ORA-06501	-6501	PL/SQL has an internal problem.
ROWTYPE_MISMATCH	ORA-06504	-6504	The host <i>cursor</i> variable and PL/SQL <i>cursor</i> variable involved in an assignment have incompatible return types. For example, when an open host <i>cursor</i> variable is passed to a stored subprogram, the return types of the actual and formal parameters must be compatible.

<i>Exception</i>	<i>Oracle Error</i>	<i>SQLCODE Value</i>	<i>Raised when</i>
SELF_IS_NULL	ORA-30625	-30625	Your program attempts to call a <i>MEMBER</i> method on a <i>NULL</i> instance. That is, the built-in parameter <i>SELF</i> (which is always the first parameter passed to a <i>MEMBER</i> method) is null.
STORAGE_ERROR	ORA-06500	-6500	PL/SQL runs out of memory or memory has been corrupted.
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533	Your program references a <i>nested table</i> or <i>varray</i> element using an index number larger than the number of elements in the collection.
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532	Your program references a <i>nested table</i> or <i>varray</i> element using an index number outside the legal range (e.g. -1).
SYS_INVALID_ROWID	ORA-01410	-1410	The conversion of a character string into a universal ROWID fails because the character string does not represent a valid <i>ROWID</i> .
TIMEOUT_ON_RESOURCE	ORA-00051	-51	A time-out occurs while Oracle is waiting for a resource.
TOO_MANY_ROWS	ORA-01422	-1422	A <i>SELECT INTO</i> statement returns more than one row.
VALUE_ERROR	ORA-06502	-6502	Arithmetic, conversion, truncation, or size-constraint error occurs. For example, when your program <i>selects</i> a column value into a character variable, if the value is longer than the declared length of the variable, PL/SQL aborts the assignment and raises <i>VALUE_ERROR</i> . In procedural statements, <i>VALUE_ERROR</i> is raised if the conversion of a character string into a number fails. (In SQL statements, <i>INVALID_NUMBER</i> is raised.)
ZERO_DIVIDE	ORA-01476	-1476	Your program attempts to divide a number by zero.

Source: docs.oracle.com

Raising user-defined exception using **RAISE_APPLICATION_ERROR** method:

```
CREATE OR REPLACE PROCEDURE
    Proc_register_subj(p_st_id student.student_id%type,
                      P_subj_id subject.subject_id%type,
                      P_year study_subjects.school_year%type)
IS
    V_count integer;
begin
    select count(*) into v_count
    from student
    where student_id = p_st_id;
    IF v_count = 0 THEN
        RAISE_APPLICATION_ERROR(-20000, 'Such student does not exist...');
    END IF;

    select count(*) into v_count
    from subject
    where subject_id = p_subj_id;
    IF v_count = 0 THEN
        RAISE_APPLICATION_ERROR(-20001, 'Such subject does not exist...');
    END IF;

    select count(*) into v_count
    from subject_year
    where subject_id = p_subj_id and school_year = p_year;
    IF v_count = 0 THEN
        RAISE_APPLICATION_ERROR(-20002, 'Such subject cannot be ' ||
        'registered during defined school year...');
    END IF;

    select count(*) into v_count
    from study_subjects
    where subject_id = p_subj_id and school_year = p_year
    and student_id = p_st_id;
    IF v_count = 1 THEN
        RAISE_APPLICATION_ERROR(-20003, 'Such subject is already ' ||
        'registered for particular student and school year');
    END IF;

    -- everything ok, new data row can be inserted
    insert into study_subjects(school_year,student_id,subject_id,lecturer)
    select school_year, p_st_id, subject_id, guarantee
    from subject_year
    where subject_id = p_subj_id and school_year = p_year;
end;
/
```

An exception will be raised after launching a defined procedure because the determined student does not exist.

```
execute Proc_register_subj(null, 'BI06', 2015);
```

```
ORA-20000: Such student does not exist.
```

Now, try another example. Execute procedure with other parameters:

```
execute Proc_register_subj(550545, 'BI06', 2015);
```

```
ORA-20002: Such subject cannot be registered during the defined school year.
```

If you define the correct data, the procedure will be completed successfully.

```
execute Proc_register_subj(550545, 'BI06', 2009);
```

```
PL/SQL procedure successfully completed.
```

The second example is based on the **RAISE** command:

```
CREATE OR REPLACE PROCEDURE
    Proc_register_subj(p_st_id student.student_id%type,
                      p_subj_id subject.subject_id%type,
                      p_year study_subjects.school_year%type)
IS
    v_count integer;
    Err1 EXCEPTION;
    Err2 EXCEPTION;
    Err3 EXCEPTION;
    Err4 EXCEPTION;
begin
    select count(*) into v_count
    from student
    where student_id = p_st_id;
    IF v_count = 0 THEN
        RAISE err1;
    END IF;

    select count(*) into v_count
    from subject
    where subject_id = p_subj_id;
    IF v_count = 0 THEN
        RAISE err2;
    END IF;

    select count(*) into v_count
    from subject_year
    where subject_id = p_subj_id and school_year = p_year;
    IF v_count = 0 THEN
        RAISE err3;
    END IF;

    select count(*) into v_count
    from study_subjects
    where subject_id = p_subj_id
    and school_year = p_year
    and student_id = p_st_id;
    IF v_count = 1 THEN
        RAISE err4;
    END IF;
```

```

-- everything ok, new data row can be inserted
insert into study_subjects(school_year, student_id, subject_id, lecturer)
select school_year, p_st_id, subject_id, guarantee
from subject_year
where subject_id = p_subj_id and school_year = p_year;

EXCEPTION
WHEN err1
THEN dbms_output.put_line('Such student does not exist.');
```

```

WHEN err2
THEN dbms_output.put_line('Such subject does not exist.');
```

```

WHEN err3
THEN dbms_output.put_line('Such subject cannot be registered during
                           the defined school year.');
```

```

WHEN err4
THEN dbms_output.put_line('Such subject is already registered for
                           particular student and school year');
```

```

WHEN others
THEN dbms_output.put_line('Another exception has been raised...');
```

```

end;
/
```

Thus, how to define a function to get a total number of credits for a particular student without raising any error? What about students with no registered (passed) subjects yet?

You can use **OUTER JOIN** to ensure that the total number will be processed. However, an exception will be raised if you specify *student_id*, which is not assigned to any student.

Be aware, the name of the parameter or variable cannot be the same as the attribute name of the table or view in the Select statement. It is a severe mistake, which the compiler cannot distinguish. However, it can produce incorrect data results. The name of the attribute has higher importance (weight).

9.8 Ways of passing parameters

Database system Oracle provides three ways to pass parameters to the stored PL/SQL block – by **position**, by **name** or the **combination** of both previously mentioned.

9.8.1 Position way of passing parameters.

If the position way of passing parameters is used, in that case, the order of values to be passed must reflect the order of definition of the stored PL/SQL block, so let's have the following example:

```

create or replace proc_register_subj
(p_st_id student.student_id%type,
 p_subj_id subject.subject_id%type,
 p_year study_subjects.school_year%type)
```

So, the first parameter should identify the *student* (*p_st_id*), then, you must write an identifier of the *subject* (*p_subj_id*), and the last one, there is information about the *school year* (*p_year*). By using position way, there is no possibility to change the order. If some attribute value should be omitted, it must also be noted there (e.g., by using **NULL** value):

```

execute proc_register_subj(12345, 'BI06', 2015);
```

```
execute proc_register_subj(12345, 'BI06', NULL);
```

Let's have another example of the procedure:

```
create or replace procedure proc_set_exam_result
    (p_st_id student.student_id%type,
     p_subj_id subject.subject_id%type,
     p_year study_subjects.school_year%type,
     p_result study_subjects.result%type,
     p_date study_subjects.exam_date%type DEFAULT sysdate)
...

```

In the case of the position way approach, the execution command looks like following:

```
exec proc_set_exam_result(1, 'BI', 2014, 'A',
                          to_date(15.7.2017, 'DD.MM.YYYY'));
```

Notice the default value in the procedure header. *DEFAULT* values can be processed only at the end of the definition. Thus, if we have the following header of the function, we have to code each value for the first four parameters explicitly. The last one is enhanced with a *default* value.

```
exec proc_set_exam_result(1, 'BI', 2014, 'A');
```

However, if you change the order in the definition, then the *DEFAULT* value cannot be used when using position way:

```
create or replace procedure proc_set_exam_result
    (p_date study_subjects.exam_date%type DEFAULT sysdate,
     p_st_id student.student_id%type,
     p_subj_id subject.subject_id%type,
     p_year study_subjects.school_year%type,
     p_result study_subjects.result%type)
...

```

9.8.2 Passing parameters using names

Named way of passing parameters is another technique, each parameter value is associated directly with the name. In that case, the order of parameter definition (and *DEFAULT* values) is not important:

```
proc_set_exam_result(123      => p_st_id,
                      sysdate-1 => p_date,
                      'BI06'    => p_subj_id,
                      2015      => p_year,
                      'C'       => p_result);
```

The *default* value can be assigned automatically by omitting a particular value during the calling – *p_date* will be set based on the *default* value.

```
proc_set_exam_result(123      => p_st_id,
                      'BI06'    => p_subj_id,
                      2015      => p_year,
                      'C'       => p_result);
```

9.8.3 Hybrid passing

In this case, the first parameters can be passed using position, followed by naming convention. However, be aware, all parameters after using any named convention must be called by name. Hybrid passing is a combination of position and name passing.

Let's have the following example again:

```
Create or replace procedure proc_set_exam_result
(p_st_id student.student_id%type,
 p_subj_id subject.subject_id%type,
 p_year study_subjects.school_year%type,
 p_result study_subjects.result%type,
 p_date study_subjects.exam_date%type DEFAULT sysdate)
```

So, the first three examples are correct, however, the fourth cannot be used:

```
proc_set_exam_result(123, 'BI06', 2015, 'A', sysdate);
```

```
proc_set_exam_result(123, 'BI06', 'A' => p_result, 2015 => p_year,
 sysdate => p_date);
```

```
proc_set_exam_result(123, 'BI06', 2015 => p_year, sysdate => p_date,
 'A' => p_result);
```

```
proc_set_exam_result(123, 'BI06', 2015 => p_year, 'A', sysdate);
```

In the following calling, default value for *p_date* will be used:

```
proc_set_exam_result(123, 'BI06', 2015 => p_year, 'A' => p_result);
```

9.9 Differences between anonymous and stored (named) PL/SQL block

The main difference between anonymous and stored (name) PL/SQL blocks is execution. Anonymous block is executed directly after compilation, and its code is not stored in the database. On the other hand, procedure and function are delimited by their unique name, and its processing has two phases – compilation and execution. Once the stored PL/SQL block is compiled, it can be used and executed multiple times without direct access to the implementation code. Keeping the named (stored) PL/SQL block can be processed more effectively based on optimization techniques (like using statistics). The code of the named PL/SQL block is stored in the data dictionary. Original file with the code of the method is not later necessary for the processing.

9.10 Removing procedures and functions

To remove stored PL/SQL block from the system, the following syntax should be used:

```
drop procedure procedure_name;
```

```
drop function function_name;
```

```
drop procedure raise_salary;
```

Result:

```
Procedure dropped;
Function dropped.
```

9.11 Select statement in PL/SQL

Results of the *Select* statement in the PL/SQL block must be stored for processing. If the statement result is only one row, the ***SELECT ... INTO type*** definition can be used. The *cursor* can provide generalization and can deal with any *Select* statement based on resulting cardinality.

9.11.1 SELECT INTO type

To use the ***SELECT INTO*** type, the particular *Select* statement must return directly one row. Otherwise, an *Exception* will be raised (*no_data_found* or *ORA-01422: exact fetch returns more than requested a number of rows*).

After keyword ***INTO***, there is a list of variables. The order must correspond with the definition in the *Select* statement clause. For variables definition, the data type can be copied from the table attribute (*table_name.attribute_name%type*). Moreover, the record can be defined to store all attributes together.

```
SELECT list_of_attributes INTO list_of_variables
FROM ...
```

Example:

```
declare
  v_name personal_data.name%type;
  v_surname personal_data.surname%type;
  v_count integer;
begin
  select name, surname, count(subject_id) into v_name, v_surname, v_count
    from personal_data JOIN student using(personal_id)
      LEFT JOIN study_subjects using(student_id)
    where student_id = 550545
    group by name, surname, student_id;
  dbms_output.put_line('Total number of student ('
                        || v_name || ' ' || v_surname
                        || ') subjects is ' || v_count || '.');
end;
/
```

```
Total number of student (Carol Pearce) subjects is 3.
```

Using record definition, the solution will look like the following. The record is defined as the type with individual elements listed after the “*is record*” keyword. Each element

consists of the *name* and *data type*. The defined variable itself uses such type as data type (*v_rec t_rec*):

```
declare
    type t_rec is record(
        name personal_data.name%type,
        surname personal_data.surname%type,
        count integer
    );
    v_rec t_rec;
begin
    select name, surname, count(subject_id) into v_rec
    from personal_data JOIN student using(personal_id)
    LEFT JOIN study_subjects using(student_id)
    where student_id = 550545
    group by name, surname, student_id;
    dbms_output.put_line('Total number of student (' || v_rec.name || ' '
        || v_rec.surname || ') subjects is '
        || v_rec.count || '.');
end;
/
```

9.11.2 CURSOR

In this chapter, we will deal only with *static cursor* type. *Dynamic cursor* definition is out of the scope of this subject. Using cursor definition, the total number of data rows in the result set is not essential. However, they have to be processed sequentially. DBS Oracle does not provide *scroll cursor* functionality for row identifier management and definition. It can manage only *sequential cursors*.

There are many cursor definitions. *Cannan* [6] defines it as a mechanism allowing access to table rows. *Pokorný* [68] extends the definition – a *cursor* is an object of SQL language, which numbers the data in the record set obtained by the *Select* statement and allows to *update* or *delete* currently addressed record tuple. Finally, *Matiaško* [49] [50] provides the following definition – *cursor* is the object of SQL language, which makes entries available, obtained from the *Select* statement.

We will distinguish three cursor types. The coding techniques only cause their difference. However, they provide the same result sets.

Let's have an anonymous block, by which we want to list the names and actual class of the actual students:

The *Select* statement to provide required data will look like this:

```
select name, surname, student_id, class
from personal_data JOIN student using(personal_id)
where final_date IS NULL;
```

To encapsulate it into PL/SQL block, it is necessary to define a *cursor*.

Fig. 9.2 shows the commands for dealing with cursors – *declare*, *open*, *fetch*, process and *close*.

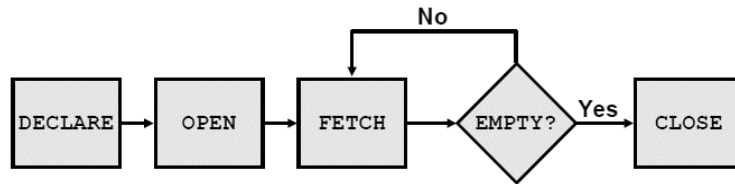


Fig. 9.2: Cursor processing step; source: Oracle PL/SQL and administration materials

Cursor processing and management can be associated with the box (container) in the memory. The *Declaration* clause creates the box with the data inside. *Opening* cursor causes opening the box making data available. Individual *fetch* operations retrieve data until the box is empty. Then, the defined box is *closed* and *freed* from the memory.

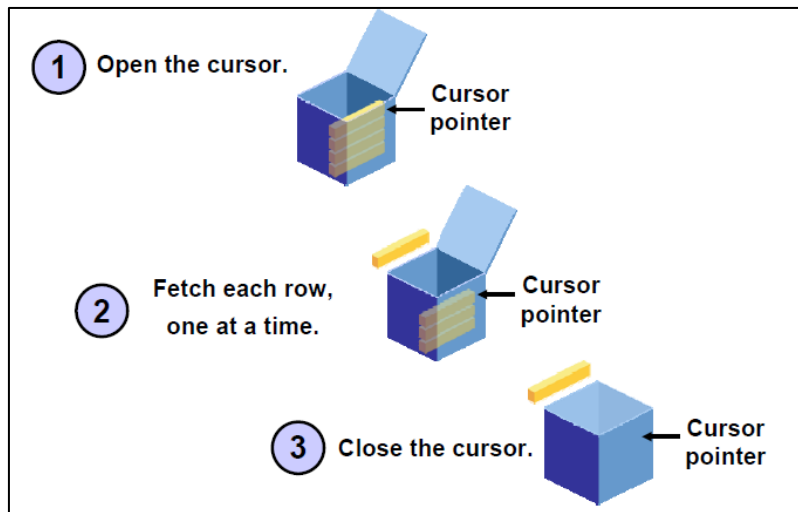


Fig. 9.3: Cursor processing step; source: Oracle PL/SQL and administration materials

Open command ensures these activities:

- the syntax of the *Select* statement checking,
- controlling access rights,
- checking the existence of the tables and columns inside,
- specifies the active set,
- allocates memory for the defined command,
- sets the pointer to the beginning of the memory space.

Close command:

- undefines the active set,
- releases and frees allocated memory.

The cursor itself can be used after the *declaration* and *opening* cursor itself. Otherwise, an *exception* will be raised.

Fetch command is a core part of the cursor definition – moves the pointer to the current (consecutive) row of the active set and enforces the data selection of the current row from the database by placing it into allocated memory.

Notice that DBS usually locks all rows, which are accessed using a *cursor*. Therefore, *commit* and *rollback* commands *close* all *opened* cursors and release locks.

The following example shows the cursor's explicit definition and manages it explicitly by **opening**, **fetching**, and **closing** the cursor.

```
declare
    cursor cur_st IS select name, surname, student_id, class
                      from personal_data JOIN student using(personal_id)
                      where final_date IS NULL;
    v_row cur_st%rowtype;
begin
    open cur_st;
    loop
        fetch cur_st into v_row;
        exit when cur_st%notfound;
        dbms_output.put_line(v_row.name || ' ' || v_row.surname || ', ' ||
                             v_row.student_id || ', class: ' || v_row.class);
    end loop;
    close cur_st;
end;
/
```

The results will look like this:

```
Milan Clarke, 500426, class: 2
Hugo Davis, 500425, class: 2
Michael Pearce, 501512, class: 3
Carol Pearce, 550545, class: 1
John Pearce, 550698, class: 2
...
```

Do not forget to **close** the cursor at the end of the execution, although it will be closed automatically after the execution. However, if you would like to process data multiple times using the same cursor in one PL/SQL block, it must be **opened** before processing (an **open** operation can be made only if the cursor is in the **closed** state).

For example purposes, we have used the *record* **v_row** defined automatically with the same structure as the cursor. Thus, the data type will be defined like this – **cur_st%rowtype**.

An infinite loop is used with the command **Exit when cur_st%notfound** to ensure the limitation of the processing. If no other data can be obtained, processing in the loop ends.

Be careful. When using this *cursor* processing type, the following keywords must be present (in the following order) when coding – **open, loop, fetch, exit, end loop, close**.

Another cursor type is easier for coding because **open**, **close**, and **fetch** operations are managed automatically using the code and should not be written explicitly. It is based on using **FOR** loop:

```
declare
    cursor cur_st IS select name, surname, student_id, class
                      from personal_data JOIN student using(personal_id)
                      where final_date IS NULL;
begin
    for v_row in cur_st loop
        dbms_output.put_line(v_row.name || ' ' || v_row.surname || ', ' ||
                             v_row.student_id || ', class: ' || v_row.class);
    end loop;
end;
/
```

Using this **FOR** loop approach, **FOR** keyword expresses and also executes automatic **OPENing** of the cursor. Each transition reflects **FETCH** operation. **END LOOP** command also executes the **CLOSE** operation. Therefore, they cannot be coded explicitly. If you try to *close* the cursor after processing explicitly, an exception will be raised:

```
declare
    cursor cur_st IS select name, surname, student_id, class
                      from personal_data JOIN student using(personal_id)
                      where final_date IS NULL;
begin
    for v_row in cur_st loop
        dbms_output.put_line(v_row.name || ' ' || v_row.surname || ', ' ||
                              v_row.student_id || ', class: ' || v_row.class);
    end loop;
    close cur_st;
end;
/
```

```
ORA-01001: invalid cursor
```

A particular case of the **FOR** loop processing with emphasis on the cursor is based on the direct association of the **FOR** loop with the cursor definition by the *Select* statement. Thus, there is no defined variable for the cursor. However, if you want to use such a cursor twice or more times, every time, it must be coded explicitly without any possibility to reference it once again.

```
begin
    for v_row in (select name, surname, student_id, class
                  from personal_data JOIN student using(personal_id)
                  where final_date IS NULL) loop
        dbms_output.put_line(v_row.name || ' ' || v_row.surname || ', ' ||
                              v_row.student_id || ', class: ' || v_row.class);
    end loop;
end;
/
```

Likewise, **FOR** loop implicitly defines the structure for result fetching. The structure is the same as the data inside the cursor.

Although it can also be defined in the declare section, they are separate variables with differing relevance, as demonstrated by the following example.

```
declare
    control integer := 1;
begin
    for control in (select student_id from student) loop
        dbms_output.put_line(control.student_id);
    end loop;
    dbms_output.put_line(control);
end;
/
```

```
550123
550127
550807
550945
...
```

There is no possibility to manage variable “*control*” inside the loop because, in that block, it is predefined using *FOR* loop.

Be sure implicitly defined variable “*control*” inside the *FOR* loop is always record, although only one attribute or function is referenced inside the particular *Select* statement. Name of the attribute or alias (if dealing with functions creating an attribute, the result must be aliased to be possible to reference them) delimits the structure of the record. Thus, in standard conditions, inside the *FOR* loop, the only reference to particular record can be done:

```
declare
    control integer := 1;
begin
    for control in (select student_id from student) loop
        control.student_id := 10;
        control := 1; --it is impossible to perform
        dbms_output.put_line(control.student_id);
    end loop;
    dbms_output.put_line(control);
end;
/
```

However, do not be confused. It is available to reference and overlap individual validities. Individual parts of the code can be named, and by using such names, the individual variable scope can be delimited to potential problems. How does it work? The principles can be effectively described in the example:

```
<<main>>
declare
    control integer:=0;
begin
    <<inner>>
    for control in (select student_id from student) loop
        main.control := main.control + 1;
        dbms_output.put_line(control.student_id);
        dbms_output.put_line(inner.control.student_id);
        dbms_output.put_line(main.control);
    end loop;
    dbms_output.put_line(control);
end;
/
```

Provided results:

```
500422
500422
2
500423
500423
3
500424
500424
4
...
```

In that case, in the inner block, a higher priority gets the record. Thus, it can be referenced directly (*control.student_id*) or by using name of the block (*inner.control.student_id*).

But referencing variable defined in the outer (<<*main*>>) block, it is necessary to use also its name – block name (*main. control*).

So, as you can see, you can have multiple structures with the same name, but it is not very convenient to use it. Developers and programmers managing code can be confused.

As procedures and functions, also cursors can depend on parameters, which are then obviously reflected as conditions. Parameters are defined after the name of the cursor:

```
declare
  cursor cur_st(p_class integer) is
    select name, surname, student_id
      from personal_data join student using(personal_id)
     where class = p_class;
begin
  for v_row in cur_st(1) loop
    dbms_output.put_line(v_row.name || ' ' || v_row.surname || ', ' ||
                        v_row.student_id);
  end loop;
end;
/
```

```
Jack Robinson, 501333
Mark Bailey, 501555
Carol Pearce, 550545
John Young, 550127
Suzanne Walker, 550123
Mark Vox, 501448
```

To conclude the processing data generated by the *Select* statement in the PL/SQL block, let's show you one more complex example of cursor processing. The aim is to create a header for each *student* followed by his registered *subjects*. The solution will be based on two cursors. One of them will make a *header*. The second one will list the *registered subjects*. As you can see, the second cursor (*cur_subj*) will be used multiple times but with different parameters. Therefore, proper managing of the *CLOSE* operation is significant.

For the explanation purposes, lines are numbered:

```
1 declare
2   cursor cur_st is select name, surname, student_id
3                     from personal_data join student
                                     using(personal_id);

4   cursor cur_subj(st_id integer) is (select subject_id, name
5                                       from study_subjects join subject
                                               using(subject_id)
6                                       where student_id=st_id);
7   v_student cur_st%rowtype;
8   v_subject cur_subj%rowtype;
9 begin
10  open cur_st;
11  loop
12    fetch cur_st into v_student;
13    exit when cur_st%notfound;
14    dbms_output.put_line('Name: ' || rpad(v_student.name,15) ||
15                        'Surname: ' || rpad(v_student.surname,15) ||
16                        'ID: ' || rpad(v_student.student_id,15));
17    open cur_subj(v_student.student_id);
```

```

18      loop
19          fetch cur_subj into v_subject;
20          exit when cur_subj%notfound;
21          dbms_output.put_line('...' || rpad(v_subject.subject_id,8) ||
                                v_subject.name);
22      end loop;
23      close cur_subj;
24  end loop;
25  close cur_st;
26 end;
```

Lines 1-8 are used for *variables* declaration. There are two *cursors* defined – the second one is parametrical. *Fetched records* of the *cursors* are assigned to defined *variables* (lines 7,8). Lines 10-26 form the body of the PL/SQL block. First, the *cursor* for managing students is *opened* (line 10), then individual student records are subsequently assigned to a defined variable (line 12) and written to the console (line 14-16). The limitation of the *LOOP* processing is provided by line 13. If the row has been read, a particular student identifier (*student_id* of the *v_student* variable) is passed as a parameter to the second *cursor* (line 17), which lists the subjects for a particular student (line 18-23). If you omit to *CLOSE* the second cursor (line 23), during the processing of the second student, an *exception* will be raised:

```
ORA-06511: PL/SQL: cursor already open.
```

The output will look like this:

```

Name: Jacob          Surname: Murgas          ID: 550945
...BN10  Communication technologies
...BI10  Java
...BI06  Database systems - the best subject :)
Name: Jack          Surname: Clever          ID: 501003
...BI06  Database systems - the best subject :)
...BI06  Database systems - the best subject :)
...BS03  Software engineering
Name: Mark          Surname: Vox          ID: 501448
...BI03  Programming language C
...BI23  Object programming
...BA10  Theory of managing schedules
Name: Sim           Surname: Eas          ID: 501559
...BI06  Database systems - the best subject :)
```

9.12 Increasing control – access rights

The standard method management approach is based on accessing objects owned by the owner of such a method. Thus, if there is no fully qualified object name (*owner_name.object_name*), the owner schema object is accessed by default. Let's have a simple example, which gets the number of rows of the student table. Let's assume that such a procedure is created by user *Kvet* (*Kvet* is the owner of the procedure).

```
create or replace procedure get_student_count
is
    v_count integer;
begin
    select count(*) into v_count from student;
    dbms_output.put_line('Number of rows in the student table is: ' ||
                          v_count);
end;
/
```

Regardless of who executes the procedure, the result will be 37.

```
-- KVET
execute get_student_count;
```

```
Number of rows in the student table is: 37
```

Then, *grant execute privilege to Kmat*:

```
-- KVET
grant execute on get_student_count to Kmat;
```

And execute defined procedure by user *Kmat*:

```
-- KMAT
execute kvet.get_student_count;
```

```
Number of rows in the student table is: 37
```

To demonstrate that it reflects the table of the *Kvet* schema, let's remove all data from the *student* table of the user *Kvet* (be careful with referential integrity):

```
-- KVET
delete from study_subjects;
delete from student;
commit;
```

The result of the calling procedure by both users is value 0:

```
-- KVET
execute get_student_count;
```

```
Number of rows in the student table is: 0
```

```
-- KMAT
execute kvet.get_student_count;
```

```
Number of rows in the student table is: 0
```

Some cases require increasing control access rights mechanism to the used objects. One way to deal with it is to transfer *control rights to the user who wants to execute a particular method*.

If we want to *control user access rights*, who wants to execute the method, it is necessary to define it in the particular method using the ***AUTHID CURRENT_USER*** clause. It means that the default approach controlling the owner of the method is redefined and moved to the user, who executes the method (default schema for object access is redefined). Thus, control mechanisms (rights on objects used in the method) are performed before the execution itself. If no ***AUTHID CURRENT_USER*** clause is defined, only rights to execute a particular method are controlled (***EXECUTE*** privilege). Notice that

in the previous example, user *Kmat* does not need to have privileges to the *student* table of *Kvet* schema.

Let's have the following example.

First of all, user *Kvet* creates a *table* converting the exam percentage to the string format.

```
-- Kvet
Create table result_tab
  (perc_from integer,
   perc_to integer,
   result char(1),
   description varchar2(50));
```

Then, the table is filled.

```
-- Kvet
insert into result_tab values(93, 100, 'A', 'excellent results');
insert into result_tab values(85, 92, 'B', 'results above average');
insert into result_tab values(77, 84, 'C', 'results on average');
insert into result_tab values(69, 76, 'D', 'acceptable result');
insert into result_tab values(61, 68, 'E',
                             ' results fulfilling the minimum requirements');
insert into result_tab values(0, 60, 'F',
                             ' failed - further work required');
commit;
```

Afterward, user *Kvet* creates *function* *get_result* and authorizes the user who wants to execute such function.

```
-- Kvet
Create or replace function get_result (p_points integer)
  return varchar2
  AUTHID CURRENT_USER
is
  v_Result varchar2(35);
begin
  select result_tab.result || ' - ' || description into v_result
  from   kvet.result_tab
        where p_points between perc_from and perc_to;
  return v_result;
EXCEPTION
  when others then return 'unknown';
end;
/
```

Add privileges to *Kmat* a let him execute the defined function.

```
-- Kvet
grant execute on get_result to KMAT;
```

```
-- KMAT
select Kvet.get_result(95) from dual;
```

What will be the result? Will it be associated with result “A”? No, at all. The result will be “unknown”.

```
Kvet.GET_RESULT(95)
-----
unknown
```


Try to explain the reason why that happened.

The reason is that exception has been raised because user *Kmat* does not have the privilege to access the *result_table* table of the user *Kvet*. Whereas the ***OTHERS*** type in the ***EXCEPTION*** covers all exception types, it has been processed by it. That is the consequence of using ***AUTHID CURRENT_USER***, so access rights to the table must be granted to the caller. Notice that the owner of the table inside the function is defined explicitly. A side effect of using the ***AUTHID CURRENT_USER*** clause is checking access rights for the objects inside the method.

Therefore, *if the table privilege is granted*, results are corrected (notice that the table in the query inside the function contains the fully qualified name of the table (*kvet.result_tab*)):

```
-- KJET
grant select on result_tab to KMAT;
```

Then, the results of the method are correct – privileges are successfully checked, table of *kvet* user is accessed:

```
GET_RESULT(95)
-----
A - excellent results
```

By the definition of *access rights controlling*, one stored PL/SQL block can deal with multiple tables based on the schema of the caller. So, if the table name in the *Select* statement in the PL/SQL block is not *prefixed by the user schema*, the caller schema will be used during the execution.

Let's create the *function* in the ***KJET*** schema and *grant* the ***EXECUTE*** privilege to ***KMAT***:

```
-- KJET
create or replace function get_result (p_points integer)
return varchar2
AUTHID CURRENT_USER
is
v_Result varchar2(35);
begin
select result_tab.result || ' - ' || description into v_result
from result_tab
where p_points between perc_from and perc_to;
return v_result;
EXCEPTION
when others then return 'unknown';
end;
/
```

```
-- KJET
grant execute on get_result to KMAT;
```

Now, let's have the table ***result_tab*** in ***KJET*** and also ***KMAT*** schema. What will happen if the data in the tables *are not the same* (realize that the table inside the function does

not denote schema explicitly)? They will get different results, although both call the same function. Let's have the example:

```
-- KMAT
Create table result_tab(perc_from integer,
                        perc_to integer,
                        result char(1),
                        description varchar2(30));
```

```
-- KMAT
insert into result_tab values(90, 100, 'A', 'excellent results');
insert into result_tab values(80, 89, 'B', 'results above average');
insert into result_tab values(70, 79, 'C', 'results on average');
insert into result_tab values(60, 69, 'D', 'acceptable result');
insert into result_tab values(50, 59, 'E',
                              ' results fulfilling the minimum requirements');
insert into result_tab values(0, 49, 'F',
                              ' failed - further work required');
```

What about the results? Emphasize the results:

```
-- KVET
select KVET.get_result(90) from dual;
```

```
GET_RESULT(90)
-----
B - results above average
```

```
-- KMAT
Select KVET.get_result(90) from dual;
```

```
GET_RESULT(90)
-----
A - excellent results
```

Thus, be aware *each user processes his own table representation*. Therefore, they can get different results.

9.13 Packages

A package is a schema object which can group multiple types, items, and subprograms (procedure and functions). Compared with standalone function or procedure, the package supports the overloading of the methods.

The package usually has two parts – *specification* and *body*. However, the *body* is *optional* but traditionally defined, too (if the *specification* does not have a method definition, there is no necessity to define the *body*. Otherwise, it is required – all methods must be implemented in the *body*). The *specification* defines the interface between the implemented code of the subprograms and the user interface (application interface). It contains all *variables*, *constants*, *cursors*, *exceptions*, and *header of the methods*, which can be called from the outside environment, so it is a public part of the *package*. On the other hand, there is also a private part, called a *body*. It contains the *implementation of all methods*, regardless of whether they are public or private, and also it deals with local (private) *variables*, *cursors*, and *exceptions*. Private methods and items can be managed only *inside* the package by the implemented methods. There is no possibility to deal with them externally. There is also possible to authorize the caller (*AUTHID CURRENT_USER*), but that clause

is associated with the whole package, not with individual methods, and should be listed in the *specification*. The following figure shows the principles of the package definition and association with the applications, and then, the syntax is defined.

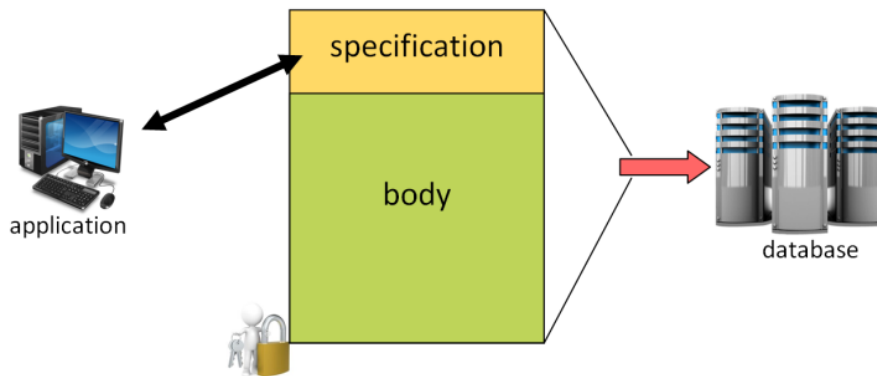


Fig. 9.4: Package

9.13.1 Package specification syntax

```
CREATE [OR REPLACE] PACKAGE package_name
[AUTHID {CURRENT_USER | DEFINER}]
{IS | AS}
[PRAGMA SERIALLY_REUSABLE;]
[collection_type_definition ...]
[record_type_definition ...]
[subtype_definition ...]
[collection_declaration ...]
[constant_declaration ...]
[exception_declaration ...]
[object_declaration ...]
[record_declaration ...]
[variable_declaration ...]
[cursor_spec ...]
[function_spec ...]
[procedure_spec ...]
[call_spec ...]
[PRAGMA RESTRICT_REFERENCES(assertions) ...]
END [package_name];
/
```

9.13.2 Package body syntax

```
[CREATE [OR REPLACE] PACKAGE BODY package_name {IS | AS}
  [PRAGMA SERIALLY_REUSABLE;]
  [collection_type_definition ...]
  [record_type_definition ...]
  [subtype_definition ...]
  [collection_declaration ...]
  [constant_declaration ...]
  [exception_declaration ...]
  [object_declaration ...]
  [record_declaration ...]
  [variable_declaration ...]
  [cursor_body ...]
  [function_spec ...]
  [procedure_spec ...]
  [call_spec ...]
[BEGIN
  sequence_of_statements]
END [package_name];]
/
```

When trying to compile the *package*, **at first, compile specifications. It is impossible to compile the body successfully without the compilation of specifications without errors.** Moreover, the specification can be used without a body, but no body can exist without specification.

Be aware that each package *specification* and package *body*, as well as standalone procedure or function, **MUST** end with the command “**END**”; followed by the slash (/) in the separate line. If it is missing, during the compilation, the system will wait to complete the definition. We want to highlight it because often, students are confused and do not know why the defined method is not compiled. Slash is a default delimiter of the code block.

Example of the package specification:

```
Create or replace package pack_student
is
  Procedure Register_subject(p_st_id integer,
                           p_subj_id char,
                           p_year study_subjects.school_year%type);
  Procedure Set_result(p_st_id integer,
                     p_subj_id char,
                     p_year study_subjects.school_year%type,
                     p_result study_subjects.result%type);
  Procedure Set_result(p_st_id integer,
                     p_subj_id char,
                     p_year study_subjects.school_year%type,
                     p_result study_subjects.result%type,
                     p_exam_date date);

end;
/
```

Example of the package body:

```
Create or replace package body pack_student
is
  procedure register_subject(p_st_id integer,
                             p_subj_id char,
                             p_year study_subjects.school_year%type)
  is
    v_count integer;
  begin
    select count(*) into v_count
      from student
      where student_id = p_st_id;
    IF v_count = 0 then
      RAISE_APPLICATION_ERROR(-20000, 'Such student does not exist...');
    END IF;

    select count(*) into v_count
      from subject
      where subject_id = p_subj_id;
    IF v_count = 0 then
      RAISE_APPLICATION_ERROR(-20001, 'Such subject does not exist...');
    END IF;

    select count(*) into v_count
      from subject_year
      where subject_id = p_subj_id and school_year = p_year;
    IF v_count = 0 then
      RAISE_APPLICATION_ERROR(-20002, 'Such subject cannot be registered
during defined school year...');
    END IF;

    select count(*) into v_count
      from study_subjects
      where subject_id = p_subj_id and school_year = p_year and
            student_id = p_st_id;
    IF v_count = 1 then
      RAISE_APPLICATION_ERROR(-20003, 'Such subject is already registered
for particular student and school year');
    END IF;

    -- everything ok, new data row can be inserted
    insert into study_subjects(school_year, student_id, subject_id,
                              lecturer)
      select school_year, p_st_id, subject_id, guarantee
        from subject_year
        where subject_id = p_subj_id and
              school_year = p_year;
  end register_subject;

  Procedure Set_result(p_st_id integer, p_subj_id char,
                       p_year study_subjects.school_year%type,
                       p_result study_subjects.result%type)
  is
    v_count integer;
```

```

begin
    select count(*) into v_count
    from study_subjects
    where school_year = p_year
    and student_id = p_st_id
    and subject_id = p_subj_id;

    IF v_count=0 then
        RAISE_APPLICATION_ERROR(-20004, 'Such subject has not been
registered for particular student and school year');
    END IF;

    select count(*) into v_count
    from study_subjects
    where school_year = p_year
    and student_id = p_st_id
    and subject_id = p_subj_id
    and RESULT IS NULL;

    IF v_count=0 then
        RAISE_APPLICATION_ERROR(-20005, 'Such subject has been already
evaluated by the result. You cannot change it.');
```

```

    END IF;

    -- everything ok, row can be updated
    update study_subjects
    set result = p_result, exam_date = sysdate
    where school_year = p_year
    and student_id = p_st_id
    and subject_id = p_subj_id;
end Set_result;

Procedure Set_result(p_st_id integer, p_subj_id char,
                    p_year study_subjects.school_year%type,
                    p_result study_subjects.result%type,
                    p_exam_date date)
is
    v_count integer;
Begin
    select count(*) into v_count
    from study_subjects
    where school_year = p_year
    and student_id = p_st_id
    and subject_id = p_subj_id;

    IF v_count=0 then
        RAISE_APPLICATION_ERROR(-20004, 'Such subject has not been
registered for particular student and school year');
```

```

    END IF;

    select count(*) into v_count
    from study_subjects
    where school_year = p_year
    and student_id = p_st_id
    and subject_id = p_subj_id
    and RESULT IS NULL;

```

```
IF v_count=0 then
    RAISE_APPLICATION_ERROR(-20005, 'Such subject has been already
    evaluated by the result. You cannot change it.');
```

```
END IF;

-- everything ok, row can be updated
update study_subjects
    set result = p_result, exam_date = p_exam_date
    where school_year = p_year
        and student_id = p_st_id
        and subject_id = p_subj_id;

end Set_result;

end pack_student;
/
```

What about if you want to change the implementation of any procedure? No problem, existing applications are associated with package *specification*. Thus, only the package *body* is needed to be recompiled. If you want to add a private method, you can make the change and then recompile only the package *body*.

However, the problem can occur if you want to change the package *specification*. If the *specification* is modified, it has to be compiled. Moreover, also package *body* should be recompiled.

Let's have the following example. We want to *add a public function*, which will express whether such student has already passed successfully defined subject (return value will be *TRUE*), otherwise, return value will be *FALSE*. *Alter package* is used.

```
Alter package pack_student
    add function student_pass(p_st_id integer, p_subj_id char)
        return boolean rebuild;
```

Now, new functionality is added to package *pack_student specification*. However, it cannot be used because the package *body* does not reflect the change, so it is necessary to add its implementation to the package *body*. Notice that such added *function* cannot be used in SQL statements because it returns non-SQL data type result (*boolean*).

Packages have a lot of advantages in comparison with standalone methods. It is possible to group related actions and types together with regards to *overloading*, which is not possible to be done directly (without packaging). Moreover, we can define private methods, which will not be available outside the package. Also, private items (*constants, variables, ...*) can be defined.

9.13.3 Overloading

As you can see in the previous example, one of the main advantages of the *package* is *overloading* technology (*only methods in the package can be overloaded*). In that case, *methods* can have the *same name*, but they *differ in parameters* – multiple solutions can have the same name). However, take care of *implicit conversions* when using *overloading*. If various methods can be used, the system cannot evaluate which one it should use so that the *exception* will be raised.

Let's see the following example. There is no problem distinguishing between methods to be used because each one has another number of parameters.

```

Create or replace package pack_student
is
  procedure Register_subject(p_st_id integer,
                           p_subj_id char,
                           p_year
                           study_subjects.school_year%type);
  procedure Set_result(p_st_id integer,
                      p_subj_id char,
                      p_year study_subjects.school_year%type,
                      p_result study_subjects.result%type);
  procedure Set_result(p_st_id integer,
                      p_subj_id char,
                      p_year study_subjects.school_year%type,
                      p_result study_subjects.result%type,
                      p_exam_date date);
end;
/

```

However, let's have another example. Can you assume, which procedure will be used, when calling? No, because both procedures have the same name and compatible parameter data type, which can be converted to each other implicitly. In that case, the system cannot decide which one should be used.

```

Create or replace package pack_overloading
is
  Procedure proc(str char);
  Procedure proc(str varchar);
end;
/

```

However, if the names of the parameters differ, the named notation can be used to distinguish the method to be called.

```

create or replace package pack_overloading
is
  Procedure proc(str1 char);
  Procedure proc(str2 varchar);
end;
/

```

```

exec pack_overloading.proc('some string' => str1);

```

9.13.4 Initialization block

An optional part of the package is an *initialization block*, which is executed only once when there is the *first reference* on the package – *when it is loaded to the memory*. *Initialization block* is located *at the end of the package body*, started with the **BEGIN**

command, until the end of the body. Notice there is no extra **END** command of the initialization, only the global end of the package *body*:

```
[CREATE [OR REPLACE] PACKAGE BODY package_name {IS | AS}
  [PRAGMA SERIALLY_REUSABLE;]
  [collection_type_definition ...]
  [record_type_definition ...]
  [subtype_definition ...]
  [collection_declaration ...]
  [constant_declaration ...]
  [exception_declaration ...]
  [object_declaration ...]
  [record_declaration ...]
  [variable_declaration ...]
  [cursor_body ...]
  [function_spec ...]
  [procedure_spec ...]
  [call_spec ...]
[BEGIN
  sequence_of_statements]
END [package_name];]
/
```

So, let's have the following example – create a package with one public procedure, which can set the private variable and one public function to get the actual value of it. Let's create an initialization block and see the principles and results:

```
create or replace package pack_init
is
  procedure Set_value_proc(p_id integer);
  function Get_value_func return integer;
end;
/
```

```
create or replace package body pack_init
is
  value integer;

  procedure Set_value_proc(p_id integer)
  is
  begin
    value := p_id;
  end;

  function Get_value_func return integer
  is
  begin
    return value;
  end;

  begin
    value := 1;
  end pack_init;
/
```

So, execute the following statement sequence. What about the results? Think and check your assumption (results are bold).

```
select pack_init.Get_value_func() from dual;
```

```
PACK_INIT.GET_VALUE_FUNC()
-----
1
```

```
exec pack_init.Set_value_proc(2);
```

```
PL/SQL procedure successfully completed.
```

```
select pack_init.Get_value_func() from dual;
```

```
PACK_INIT.GET_VALUE_FUNC()
-----
2
```

Be aware. The defined package is loaded into memory during the first reference. However, it is loaded to **PGA** (*Process Global Area*), not **SGA** (*System Global Area*), causing that every process (each session) has its own values and variables of the package. Thus, one session does not affect the other:

Add user “**Kmat**” privilege to execute such defined package:

```
grant execute on pack_init to Kmat;
```

Now, execute the following methods by user “**Kvet**” and “**Kmat**” and compare results. See that the results do not influence another user.

```
-- Kvet
select pack_init.Get_value_func() from dual;
```

```
PACK_INIT.GET_VALUE_FUNC()
-----
1
```

```
exec pack_init.Set_value_proc(2);
```

```
PL/SQL procedure successfully completed.
```

```
select pack_init.Get_value_func() from dual;
```

```
PACK_INIT.GET_VALUE_FUNC()
-----
2
```

```
-- Kmat (immediately after)
select kvet.pack_init.Get_value_func() from dual;
```

```
PACK_INIT.GET_VALUE_FUNC()
-----
1
```

```
exec pack_init.Set_value_proc(3);
```

```
PL/SQL procedure successfully completed.
```

```
select pack_init.Get_value_func() from dual;
```

```
PACK_INIT.GET_VALUE_FUNC()
-----
3
```

```
-- Kvet (immediately after)
select pack_init.Get_value_func() from dual;
```

```
PACK_INIT.GET_VALUE_FUNC()
-----
2
```

The same results will be reached if two sessions of the same user are used.

9.14 Practice

- Create procedure **Get_group_proc**, which will consist of these six parameters:
 - Workplace** (abbreviation of the town – first letter):
 - Z – Zilina
 - P – Prievidza
 - Field** (its numerical value) – a reference to *st_field* table
 - Specialization** (its numerical value) – a reference to *st_field* table
 - Class**
 - Sequence number of the group** (1, 2, 3, ... A, B, C ...)
 - St_group** as an OUTPUT parameter

The aim is to create the value for the study group and return it using the output parameter *st_group*.

Example: input: Z, 100, 0, 1, 2 output: 5ZI012
 input: Z, 101, 0, 3, A output: 5ZP03A

Abbreviations for fields and specializations can be found in the table **ABBREVIATION_TAB** in the **KVET_ENG** schema.

```
desc kvet_eng.abbreviation_tab
```

Name	Null	Type
FIELD_ID	NOT NULL	NUMBER (3)
SPECIALIZATION_ID	NOT NULL	NUMBER (1)
FIELD_ABBR		CHAR (1)
SPEC_ABBR		CHAR (1)

- Rewrite the previous procedure and create a similar function (**Get_group_func**). The value to be returned is a study group.
- Try to use such function (**Get_group_func**) in the *Select* statement. Is it possible?
- Create procedure **Add_subject_proc**, which will execute *Insert* statement of the new subject into the *subject* table.
 Try to insert the following data. Is it possible? If not, why?
 - Subject_id: BI14, name: Advanced database indexing
 - Subject_id: BI12, name: Introduction to studying
 - Subject_id: BI12, name: Introduction to studying
- Extend the previous procedure (**Add_subject_proc**) by adding particular *exception handling*. Test it.
- Try to use the previous procedure (**Add_subject_proc**) to add the following data to the table. Is it possible? If not, why?
 - Subject_id: null, name: Unknown subject name

7. Extend the previous procedure (*Add_subject_proc*) by adding particular *exception handling*. Test it.
8. Create function *Get_student_count_func*, which will have two parameters (subject identifier and school year). The result of the processing should be the total number of students who registered for a particular subject at a defined school year.
9. List the name of the subjects with the total number of students by using the previous function (*Get_student_count_func*).
10. Create the function *Register_func*, which will register the student for a particular *subject*. The *school year* should be based on the *actual date*:
 - if the actual month belongs to the following interval <1,8>, the school year should be decreased by one,
 - if the actual month belongs to the following interval <9,12>, use the actual school year.

Example: actual date: 23.3.2016 --> 2015
 1.11.2016 --> 2016

Check the conditions before attempting to *Insert* new data. Return information (*boolean*) expresses whether it is possible to add it or not. Can we use that function in the *Select* statement? If not, adjust it and recompile once again.

Homework practice:

1. Create function *Get_birth_func*, which parameter will be *personal_id*, and the return value will be the *birth date*.
2. Create function *Change_subj_func*, which will perform the *Update* operation of the subject name. Input parameter will be the *identifier of the subject* and *new name*. The *return value* should be:
 - The old name of the subject (if the defined subject exists),
 - Constant string “nothing”, if no subject with defined identifier was found.
3. Create package *pack_management* and define methods ensuring the following requirements:
 - *Register students for the particular subject*. It can be done only if he has not already passed it successfully before (also during another study of a particular person).
 - Get the actual value of the credits associated with the student (only if the subject is passed successfully (based on **ending_type**)).
 Attribute *ending_type* of the *subject_year* table:
 - B exam + accreditation to exam,
 - E exam,
 - S semester only (no exam).
 - Get the length of the study of the defined student.
 - *Register (insert) new student into the system*. Ensure that he has no parallel study during a defined time.
4. Create procedure *List_stud_proc*, which will list all students (using *dbms_output.put_line* method) who achieved *at least 40 credits (based on attribute cts)* (take into account only subjects, which are passed successfully).
5. Create procedure *Delete_stud_proc*, which will accept one parameter – *identifier of the student*. It will *delete* data from the *student* table (**emphasizing referential integrity**). If he has no other references in the *student* table, remove his information from the *personal_data* table.

Lab 10 – Triggers

This lab provides the reader a complex integrity management overview using the trigger associated with Insert, Update or Delete statement forming the DML trigger. It is a specific functionality that is fired automatically if a particular event occurs. The trigger can be statement type (fired only once regardless of the number of changed rows) or row (fired once for each applied row). In this lab, the reader will be navigated through the syntax and definition restrictions. He will learn how to influence the values manipulated through the operations (records) and limit the trigger firing to specific conditions.

As discussed in the section, triggers provide a robust solution for maintaining integrity. However, some constraints can be ensured using easier solutions, like CHECK constraints, default values, etc.

In section 10.11, the reader will learn the sequences, their definitions, and parameters, which can provide sufficient solutions for the primary key definition. The trigger commonly does association. There are two methods for obtaining value – NEXTVAL by applying the specified INCREMENT and CURVAL getting the current value.

Finally, the reader will get an overview related to the DDL and event triggers.

10.1 Introduction

The trigger is a stored procedure associated with the object or object type. Oracle manager automatically executes a trigger (independent of user or application that ran particular command) if the defined conditions are met. *DML* trigger can be fired only with destructive *DML* statements (*Insert*, *Update*, *Delete*). The important fact is that one trigger can be associated only with one object (table or view). It cannot be associated with the *Select* statement at all. Moreover, it cannot accept arguments.

Triggers provide a wide range of possibilities; the main tasks for them are following:

- ensure complex data security,
- restrict undesirable activities,
- allow creating strategic application rules,
- monitor user activity and data processing (audits),
- ensure synchronization,
- create statistics about the table management and activities,
- ensure consistency and referential integrity for all nodes in a distributed environment,
- ...

A particular category is formed using *DDL* triggers. They deal with creating persistent database objects and reflect the security politics.

The central part of this lab focuses on *DML* triggers. We will just briefly introduce principles of *DDL* trigger as well in the second part of the lab.

10.2 Syntax

```
CREATE [OR REPLACE] TRIGGER [schema.]trigger_name
{ {BEFORE | AFTER}
  {INSERT | DELETE | UPDATE [OF column_name1 [column2 [, ... ]]]}
  OR {DELETE | INSERT | UPDATE [OF column_name1 [column2 [, ... ]]]}
  [...]
}
|
INSTEAD OF {INSERT | UPDATE | DELETE}}
ON [schema.]table_name
[REFERENCING {OLD [AS] old_name | NEW [AS] new_name}]
[FOR EACH ROW]
[WHEN (condition)]
trigger_body
```

Each trigger must have its name. The firing position can be either *before* or *after* the associated operations. We can define one trigger for multiple operations, but only for one object (table or view). Triggers can be fired either at all times or when the conditions are met. By using the **UPDATE [OF column_name1 [column2 [, ...]]** clause, the number of times the trigger is fired is reduced – trigger is not associated only with the update statement itself, but also particular attributes must be updated. List of them is in **UPDATE [OF column_name1 [column2 [, ...]]** clause.

Referencing clause allows renaming referential records. However, it can be used only for triggers, which are launched for each changed row. Thus, it must contain **For Each Row** clause to express it. Two records can be recognized – the **new** record can be used only for the *Insert* and *Update* statement operation (*Delete* statement naturally has the only old image of the row). **The old** record can be used only for the *Update* and *Delete* statement. The structure of the **New** and **Old** record is the same as the associated table or view schema (e.g. when the trigger is associated with student table, the particular record has these elements – *student_id*, *personal_id*, *field_id*, *specialization_id*, *class*, *st_group*, *final_date*, *status*, and *first_date*). Using **New** and **Old** records can identify attribute changes with the *Update* operation. Some specific conditions and checks can be done using them.

Notice that **new** or **old** values must be colon prefixed. However, in the **When** clause, particular records **New** and **Old** are **not prefixed** by the colon (:).

```
:new.personal_id
:old.class
```

One of the essential parameters influencing how many times the trigger is launched for a particular statement is based on using or not using **For Each Row** clause.

Let's have the following example. Create *log_table* consisting of the username (*user_name*) and the actual time of the command execution (*exec_date*):

```
Create table log_table(user_name varchar2(20), exec_date date);
```

Create a trigger (associated with the *Update* operation, it can be executed before as well as after performing *Update* statements, whereas it deals with another table) with no **For Each**

Row clause (in that case, we can say that trigger type is *Statement Trigger* – executed only once regardless the number of processed rows):

```
Create or replace trigger trig_log_ss
after update on study_subjects
begin
    insert into log_table
        values(user, sysdate);
end;
/
```

Now, *update* multiple rows using only one *Update* statement:

```
update study_subjects
    set school_year = 2009
    where school_year = 2008;
```

As a result, five rows are updated.

5 rows updated.

However, how many rows have been inserted into *log_table*? *Only one, whereas trigger has been executed once for the whole statement.*

COUNT(*)
1

Also, notice that in that case, *New* or *Old* records cannot be used at all.

If you want to execute it for each changed row, an additional clause *For Each Row* has to be added. In that case, each row change is reflected by the *log_table*, and the trigger is considered as *Row Trigger*. Moreover, in this case, it is also possible to get and store historical value (using *Old* record) and actual record (using *New* record).

So, if the previous *Update* statement is executed once again (active transaction is rolled back) and five rows are updated, also five rows are inserted into *log_table*:

```
rollback;
```

```
Create or replace trigger trig_log_ss
after update on study_subjects
    FOR EACH ROW
begin
    insert into log_table
        values(user, sysdate);
end;
/
```

As evidence, *update* multiple rows using only one *Update* statement. What will happen?

```
update study_subjects
    set school_year = 2009
    where school_year = 2008;
```

5 rows updated.

In this case, the trigger is fired for each influenced row. Check the number of log records:

```
select count(*) from log_table;
```

COUNT(*)
5

So, now, let's create another logging table (*log_student*) containing information about the original and new attribute values:

```
Create table log_student
(student_id integer,
 new_field integer,
 new_specialization integer,
 old_field integer,
 old_specialization integer,
 old_st_group integer,
 new_st_group integer,
 ...);
```

The original attribute value is stored in the particular attribute prefixed by “*Old*”, new values are stored using “*New*” prefix of the attributes:

```
Create or replace trigger trig_st
before update on student
for each row
begin
insert into log_student
values(:new.student_id, :new.field_id, :new.specialization_id,
      :old.field_id, :old.specialization_id, ...);
end;
/
```

However, what data are stored in the individual records if the value does not change? *NULL* or not? Let's examine the following example. Create *log_table* to manage the status of the student, associate it with the trigger and execute the *Update* statement:

```
Create table log_table(old_status char(1),
 new_status char(1),
 user_name varchar2(20),
 exec_date date);
```

```
Create or replace trigger trig_student_status
before update on student
for each row
begin
insert into log_table
values(:old.status, :new.status, user, sysdate);
end;
/
```

```
update student
set final_date = sysdate
where student_id = 550945;
```

```
1 rows updated.
```

What about the data in the *log_table*? Realize that the *status* of the student has not been updated.

```
select old_status as "OLD", new_status as "NEW", user_name, exec_date
from log_table;
```


OLD	NEW	USER_NAME	EXEC_DATE
S	S	KVET_ENG	06.02.2017

Even though attribute value has not been changed, particular record elements store the real values (*new* – after the operation, *old* – original values).

10.3 Restrictions for trigger definition

There are several restrictions to be highlighted dealing with the trigger definition and management, namely:

- The trigger's body can contain data manipulation language (*DML*) statements, but it cannot handle the same table. *Select* statements themselves must be encapsulated by the cursors or should return only one row. In that case, the *Select Into* type can be used.
- No *TCL* statements are allowed (*Commit*, *Rollback*, and *Savepoint*), whereas processed data changes would become permanent, respectively, they would be immediately abolished.
- *DDL* statements are not allowed inside the trigger body at all. Why? Naturally, because of the transactions.
- Variables with data types *Long* and *Long Raw* cannot be used with *Old* and *New* records.
- Moreover, these requirements also apply to methods, which are called inside the body of the trigger.

10.4 Triggers turning on and off

The trigger can be associated with multiple operations and is fired automatically. Database systems allow you to manage triggers with emphasis on their enabling or disabling. In general, when the trigger definition is compiled, the particular trigger is turned on. However, if it is necessary to suspend the trigger, two possibilities are proposed – using *alter trigger* characteristics – it is used to disable the trigger temporarily.

```
ALTER TRIGGER [schema.]trigger {ENABLE | DISABLE};
```

To disable all triggers associated with the particular table, it is not necessary to do it sequentially for each trigger, but all of them can be managed using one command:

```
ALTER TABLE [schema.]table_name {ENABLE | DISABLE} ALL TRIGGERS;
```

Another approach is to *drop* the trigger. However, after that, there will be no information about the trigger's existence nor the body of the trigger.

To remove the trigger from the system, use the following command code:

```
DROP TRIGGER [schema.]trigger;
```

10.5 Changes monitoring

The trigger can provide a powerful solution for change monitoring over time. For these purposes, a row trigger should be defined for accessing individual changes. If you want to add only changed data to the *log_table*, the following solution can be introduced. However, remember that *NULL* value, in this case, delimits no change. Even if *NULL* values have

special meaning in a given system, no conflict can occur because if the value is changed in one record, it can be easily discovered (the original and new value would not be the same).

Let's have the table *personal_data* from our labs. As we can see, the majority of attributes can be *NULL*.

Which attributes will change their values? How often? Typically, the name is not changed frequently (rarely). Another case, however, occurs with the surname. If the woman gets married, in our region, she typically takes the surname of her husband. Thus, for men, changes are not performed. For women – it is usually a question of one update. Sure, a few exceptions are allowable. Vice versa, a person's address can be changed unlimited times. Thus, individual attribute changes have a different granularity of the changes. Therefore, *NULL* can mainly solve those granularity inconsistencies.

So, let's demonstrate the situation. Let's have the *log_table* and particular trigger for managing changes. It will be associated with the *Update* statements.

```
Create table log_person
(personal_id char(11),
 old_name varchar2(15),
 old_surname varchar2(15),
 old_street varchar2(20),
 old_town varchar2(50),
 old_zip char(5),
 old_nationality char(2),
 new_name varchar2(15),
 new_surname varchar2(15),
 new_street varchar2(20),
 new_town varchar2(50),
 new_zip char(5),
 new_nationality char(2));
```

The body of the trigger should evaluate individual attribute changes. Notice that you cannot change *New* or *Old* record values because if so, the change will be reflected in the database by the *Update* statement.

Thus, we will define a local variable for each attribute and evaluate the change between *Old* and *New* record in the body:

```
create or replace trigger trig_person_change
before update on personal_data
for each row
declare
v_personal_id personal_data.personal_id%type;
-- old
v_old_name personal_data.name%type;
v_old_surname personal_data.surname%type;
v_old_street personal_data.street%type;
v_old_town personal_data.town%type;
v_old_zip personal_data.zip%type;
v_old_nationality personal_data.nationality%type;
-- new
v_new_personal_id personal_data.personal_id%type;
v_new_name personal_data.name%type;
v_new_surname personal_data.surname%type;
v_new_street personal_data.street%type;
v_new_town personal_data.town%type;
v_new_zip personal_data.zip%type;
v_new_nationality personal_data.nationality%type;
```

```

begin
  if :new.name <> :old_name then
    v_new_name := :new.name;
    v_old_name := :old.name;
  end if;

  if :new.surname <> :old_surname then
    v_new_surname := :new.surname;
    v_old_surname := :old.surname;
  end if;

  if :new.street <> :old_street then
    v_new_street := :new.street;
    v_old_street := :old.street;
  end if;

  if :new.town <> :old_town then
    v_new_town := :new.town;
    v_old_town := :old.town;
  end if;

  if :new.zip <> :old_zip then
    v_new_zip := :new.zip;
    v_old_zip := :old.zip;
  end if;

  if :new.nationality <> :old_nationality then
    v_new_nationality := :new.nationality;
    v_old_nationality := :old.nationality;
  end if;

  insert into log_person values(:new.personal_id,
                                v_old_name, v_old_surname, v_old_street,
                                v_old_town, v_old_zip, v_old_nationality,
                                v_new_name, v_new_surname, v_new_street,
                                v_new_town, v_new_zip,
                                v_new_nationality);

end;
/

```

String variables are implicitly defined as *NULL*. If that is not so, we can initialize values in the variable definition part or add the *Else* clause of the processing.

The following example is based on adding the *Else* clause.

```

-- ...
if :new.name <> :old_name then
  v_new_name := :new.name;
  v_old_name := :old.name;
else
  v_new_name := NULL;
  v_old_name := NULL;
end if;
-- ...

```

Another solution is based on using the *Initialization clause*.

```

create or replace trigger trig_person_change
before update on personal_data
for each row
declare
  v_personal_id personal_data.personal_id%type;
  -- old
  v_old_name personal_data.name%type := NULL;
  v_old_surname personal_data.surname%type := NULL;
  -- ...

```

Be aware, an empty string is considered as the *NULL*:

```

declare
  v_str varchar2(10) := ''; -- there cannot be any character inside!
begin
  if v_str IS NULL then
    dbms_output.put_line('Variable is NULL');
  else
    dbms_output.put_line('Variable is empty, but NOT NULL');
  end if;
end;
/

```

When executing the defined block, the following result will be obtained:

```
Variable is NULL
```

Question for thinking about – what is the limitation of the *Row* and *Statement* trigger? Can they be directly replaced by each other?

10.6 Default values

Default values can be assigned to the table attribute in the definition. If no attribute value is set, the value to be stored to be used is replaced by the default value. ***However, be aware no value, in this case, does not equal NULL value***, as it already shows the following example:

```
create table TAB(id integer not null primary key, val integer DEFAULT 3);
```

```

insert into TAB values(1, 1);
insert into TAB values(2, null);
insert into TAB(id) values(3);

```

Data in the table are following:

```
select * from TAB;
```

ID	ID2
1	1
2	(null)
3	3

Thus, as you can see, if a *NULL* value is written explicitly, the *default* value will not be used at all.

To remove that limitation, the trigger for setting value can be defined. In that case, it can also be extended for *NULL* values – if the *NOT NULL* value is required, the predefined

value will be used. However, it cannot be said, and it is the default value because of the keyword of the database system. The following solution is prone to *NULLs*.

```
create table TAB2(id integer not null primary key, val integer);
```

```
Create or replace trigger trig_Tab2_default
before insert on TAB2
for each row
begin
    if :new.val IS NULL then
        :new.val := 3;
    end if;
end;
/
```

Perform three *Insert* statements and compare the results with the previous example.

```
insert into TAB2 values(1, 1);
insert into TAB2 values(2, null);
insert into TAB2(id) values(3);
```

```
select * from TAB2;
```

ID	ID2
1	1
2	3
3	3

In Oracle 12c version, a new clause – *default on null* – was introduced. Thus, if the value is undefined or not specified, it will be replaced by the default value. Consequently, a *NULL* value is replaced, as well.

```
create table TAB3(id integer not null primary key,
    val integer DEFAULT ON NULL 3);
```

```
insert into TAB3 values(1, 1);
insert into TAB3 values(2, null);
insert into TAB3(id) values(3);
```

Data in the table are following:

```
select * from TAB3;
```

ID	ID2
1	1
2	3
3	3

10.7 Conditions for trigger firing

Typically, some data portions can be made only by privileged people. **Grant** command cannot be directed for particular table rows, only for the whole object (table). Thus, if you **Grant** the user **KMAT** privilege to *Update* table **Employee**, in that case, **KMAT** can update any attribute of the particular table. To highlight the problem and propose a solution, create the table **Employee**, *insert* one row to it, end successfully transaction (**Commit**) and **grant**

privileges to **KMAT**. Consequently, his task will be to update the existing row of the **Employee** table owned by **KVET**.

```
-- KVET
Create table employee(emp_id integer primary key,
                     date_from date not null,
                     date_to date,
                     salary integer);

insert into employee values(1, sysdate, null, 1000);
commit;
grant update on employee to KMAT;
```

Notice that in the previous definition, there is no necessity to write the **Commit** command explicitly. The reason is based on the **Grant** command definition, which is automatically associated with **Commit** (when managing transactions, once again, never forget that all **TCL**, **DDL**, and **DCL** always commands end transaction successfully (implicit **Commit**)).

```
select * from employee;
```

EMP_ID	DATE_FROM	DATE_TO	SALARY
1	06.02.2017	(null)	1000

If the user **KMAT** performs an **Update** statement in the **Employee** table owned by **KVET**, it will not be automatically visible to other sessions because of the transaction isolation property. So, what must be done to do so?

```
-- KMAT
update kvet.employee set salary = 5000;

-- KVET
select * from employee;
```

EMP_ID	DATE_FROM	DATE_TO	SALARY
1	06.02.2017	(null)	1000

```
-- KMAT
select * from kvet.employee;
```

EMP_ID	DATE_FROM	DATE_TO	SALARY
1	06.02.2017	(null)	5000

Sure, to see the same results, began transaction of the **KMAT** user must be confirmed.

```
commit;
```

Now, also user **KVET** will see the same results as user **KMAT**.

```
-- KVET
select * from employee;
```

EMP_ID	DATE_FROM	DATE_TO	SALARY
1	06.02.2017	(null)	5000

To prevent users from changing sensitive data, the trigger can be added limiting a particular operation to a specific user or group based on defined conditions:

```
Create or replace trigger trig_emp
before update of salary on employee
for each row
when (user not in 'KVET')
begin
raise_application_error(-20000, 'Sorry, you cannot change salary.');
```

Any attempt to update the *Employee* table except owner (*KVET*) will end with raising an *exception*. Thus, no data will be updated.

```
-- KMAT
update kvet.employee set salary = 3000;
```

```
ORA-20000: Sorry, you cannot change salary.
ORA-06512: at "KVET.TRIG_EMP", line 2
ORA-04088: error during execution of trigger 'KVET.TRIG_EMP'
```

How does it work? When will the trigger be fired? What are the conditions? When will be the *When* clause executed?

The trigger is fired only if the condition in *When the* clause is evaluated as *True*. Thus, for user *KVET*, no trigger is fired.

```
-- KMAT
update kvet.employee set date_to = sysdate; -- trigger does not fire...
```

```
1 row updated.
```

However, an attempt to update also *salary* attribute will activate the *trigger*, and an *exception* will be raised:

```
-- KMAT
update kvet.employee
set date_to = sysdate+30, salary = 2000;
```

```
ERROR at line 1:
ORA-20000: Sorry, you cannot change salary.
ORA-06512: at "KVET_ENG.TRIG_EMP", line 2
```

Whereas *exception* has been raised, a particular statement (not the whole transaction) is rolled back. Thus, the *date_to* attribute value is not updated, too.

What about the difference and performance consequences if the *When* clause is omitted, respectively moved to the trigger's body? Will it be better, or not?

In the following example, the trigger will always be fired, and if the condition is met, an *exception* will be raised. Thus, also for *KMAT*, as well as *KVET*, the trigger is fired.

```

Create or replace trigger trig_emp
before update on employee
for each row
begin
  if (user not in ('KVET') and :new.salary <> :old.salary) then
    raise_application_error(-20000, 'Sorry, you cannot change salary.');
```

The solution to using *When* clause looks like following:

```

Create or replace trigger trig_emp
before update on employee
for each row
  when (user not in ('KVET') and new.salary <> old.salary)
begin
  raise_application_error(-20000, 'Sorry, you cannot change salary.');
```

Notice that using an update of the clause is more convenient and effective:

```

Create or replace trigger trig_emp
  before update of salary on employee
for each row
  when (user not in 'KVET')
begin
  raise_application_error(-20000, 'Sorry, you cannot change salary.');
```

10.8 One trigger – multiple operations

One trigger can be associated with multiple *DML* operations specified in the header of a particular trigger. However, it can be associated only with one table or view. If you create a trigger for logging, it will be necessary to distinguish also operation, which has been performed. For these purposes, the condition **IF INSERTING**, **IF UPDATING**, or **IF DELETING** can be used. Let's create a table for logging performed operations on the *student* table. *If such a table exists in your system, you can drop or rename it.*

```

Create table log_student
(old_student_id integer,
 new_student_id integer,
 operation char(1),
 username varchar2(30),
 exec_date date);
```

We will define only one trigger for all destructive *DML* operations. In the body of the trigger, conditions are used to distinguish between performed operations. Such a solution aims to group and manage common code together. *However, the same solution would be obtained if you divide the solution into three separate triggers that call the same stored procedure/function with emphasis on parameter values (differentiating the operation).*


```
Create or replace trigger trig_log_st
before insert or update or delete
on student
for each row
begin
  if inserting then
    insert into log_student
      values(null, :new.student_id, 'I', user, sysdate);
  end if;

  if updating then
    insert into log_student
      values(:old.student_id, :new.student_id, 'U', user, sysdate);
  end if;

  if deleting then
    insert into log_student
      values(:old.student_id, null, 'D', user, sysdate);
  end if;
end;
/
```

This trigger ensures that each performed destructive operation will be logged. This information of activity will be stored:

- name of the user, who performed the operation,
- when the operation has been performed,
- which operation type has been executed.

In that case, any update operation will be logged. If you want to reduce individual operations to be logged (e.g., only for *status* updating), a particular condition is extended by the name **IF UPDATING('attribute_name')** like in the following example:

```
Create or replace trigger trig_log_st
before insert or update or delete
on student
for each row
begin
  if inserting then
    insert into log_student
      values(null, :new.student_id, 'I', user, sysdate);
  end if;

  if updating('status') then
    insert into log_student
      values(:old.student_id, :new.student_id, 'U', user, sysdate);
  end if;

  if deleting then
    insert into log_student
      values(:old.student_id, null, 'D', user, sysdate);
  end if;
end;
/
```

Each update operation of the attribute *status* will be logged. However, what will happen, if you update such attribute value with its original one? Will it be logged?

```
update student set status=status;
```

```
37 rows updated.
```

All rows in the student table will be updated. However, the new value will be the same as the original. Thus, as a consequence, another 37 rows will be inserted into the *log_student* table by the trigger. To overcome this deficiency, implemented conditions of the *Update* statement are extended, as follows:

```
Create or replace trigger trig_log_st
before insert or update or delete
on student
for each row
begin
    if inserting then
        insert into log_student
            values(null, :new.student_id, 'I', user, sysdate);
    end if;

    if (updating('status') and :old.status<>:new.status) then
        insert into log_student
            values(:old.student_id, :new.student_id, 'U', user, sysdate);
    end if;

    if deleting then
        insert into log_student
            values(:old.student_id, null, 'D', user, sysdate);
    end if;
end;
/
```

10.9 Referential integrity management

The trigger can also be defined for referential integrity management. If you want to remove a student from the system, particular references must be solved sooner (information about studied subjects must be deleted sooner). The trigger can be defined to provide the desired functionality to remove the necessity of explicit management of such a situation.

Let's have the following trigger to provide cascade operations – to delete registered subject data before attempting to delete the student data themselves.

```
create or replace trigger trig_st_del_cascade
before delete on student
for each row
declare
    v_count integer;
begin
    select count(*) into v_count
    from study_subjects
    where student_id = :old.student_id;
    delete from study_subjects
    where student_id = :old.student_id;
    dbms_output.put_line(v_count ||
        ' rows have been deleted from the study_subjects table');
end;
/
```

To avoid querying table *study_subjects* twice (*select* and *delete*), a predefined function for getting the *RowCount* of the last processed operation can be used:

```
create or replace trigger trig_st_del_cascade
before delete on student
for each row
begin
delete from study_subjects
where student_id = :old.student_id;
dbms_output.put_line(SQL%ROWCOUNT ||
    ' rows have been deleted from the study_subjects table');
end;
/
```

A bit complicated situation can occur if you want to deal with the referential integrity in cascade type for the table *personal_data*. If you want to remove a person from the system, particular references must be solved sooner (student data must be deleted sooner). However, when dealing with a student removal, information about studied subjects must be deleted. The trigger can be defined to provide the desired functionality to remove the necessity of explicit management of such a situation. However, how to get the identifier of the student? Can the *SELECT ... INTO* statement type be used? Why not?

Let's have the following trigger to provide cascade operations for deleting data from the *personal_data* table. First, student identifier (*student_id*) for such a person is obtained and processed using a cursor. For each student found, particular *study subjects* are deleted, followed by the *student* delete himself. Then, the *personal_data* row is deleted automatically (after execution of the trigger).

```
create or replace trigger trig_person_del_cascade
before delete on personal_data
for each row
declare
cursor st_cur(p_person_id char) is select student_id
                                from student
                                where personal_id = p_person_id;

v_count_st integer:=0;
v_count_subj integer:=0;
begin
for rec in st_cur(:old.personal_id) loop
delete from study_subjects
where student_id = rec.student_id;
v_count_subj := v_count_subj+SQL%ROWCOUNT;
delete from student
where student_id = rec.student_id;
v_count_subj := v_count_subj+1;
end loop;
dbms_output.put_line(v_count_st ||
    ' rows has been deleted from the student table');
dbms_output.put_line(v_count_subj ||
    ' rows has been deleted from the study_subjects table');
end;
/
```

However, is it possible to simplify the previous code to avoid using cursors? If not, why?

The solution is based on using subqueries.

```
create or replace trigger trig_person_del_cascade
before delete on person
for each row
begin
delete from study_subjects where student_id IN (select student_id
                                                from student
                                                where
                                                personal_id = :old.personal_id);

dbms_output.put_line(SQL%ROWCOUNT ||
                      ' rows has been deleted from the student table');
delete from student where personal_id = :old.personal_id;
dbms_output.put_line(SQL%ROWCOUNT ||
                      ' rows has been deleted from the study_subjects table');
end;
/
```

10.10 Changing the value of the primary key

The problem can arise if there is a necessity to update the *primary key* value. If the primary key consists of the value obtained by the sequence (chapter [10.11 Sequences and triggers](#)), there is no reason to update it. This is because it does not have a specific meaning. However, what about our *student* model and table *personal_data*? The primary key of the table *personal_data* is *personal_id* and reflects the *birth_date* and *gender* of the person. If there is any mistake when adding a new person to the database, it is necessary to correct it later (when the error is discovered).

Direct *Update* statement of the primary key of the table *personal_data* is not possible due to referential integrity. Then, we will try to update the *personal_id* of the person **Jack Robinson** from the value “791229/5431” to “790229/5431”. *It would be possible for a person who is not a student to execute it like this.*

```
update personal_data
set personal_id = '790229/5431'
where personal_id = '791229/5431';
```

```
ORA-02292: integrity constraint (KVET_ENG.SYS_C00552853)
violated - child record found
```

To solve the problem without using triggers, we have to perform multiple DML operations (**INSERT INTO PERSONAL_DATA, UPDATE STUDENT, DELETE FROM PERSONAL_DATA**), whereas the *personal_id* value is referenced in the student table.

The aim is to update the *personal_id* of the person **Jack Robinson** from the value “791229/5431” to “790229/5431”. In the following example, the new value of the *personal_id* attribute is written as constant in the *Select* statement forming *Insert* operation.

```
insert into personal_data(personal_id, name, surname,
                          street, zip, town, nationality)
(select '790229/5431', name, surname, street, zip, town, nationality
 from personal_data
 where personal_id = '791229/5431');
```

```
update student
set personal_id = '790229/5431'
where personal_id = '791229/5431';
```

```
delete from personal_data
where personal_id = '791229/5431';
```

A bit complicated, isn't it? However, changing the primary key's value using a trigger, the solution is more accessible, whereas integrity constraints are checked after the trigger operation.

```
CREATE OR REPLACE TRIGGER trig_cascade_pid
AFTER UPDATE OF personal_id ON person
FOR EACH ROW
BEGIN
    UPDATE student SET personal_id = :new.personal_id
    WHERE personal_id = :old.personal_id;
END;
/
```

10.11 Sequences and triggers

A *sequence* is a database object mainly used for assigning a value to the attribute (like autoincrement, which is not directly defined for Oracle DBS). Each sequence can be identified by its unique name and can provide two methods:

- to get actual value – *seq_name.currval*
- to get the following value-based on definition – *seq_name.nextval*.

10.11.1 Sequence syntax

```
CREATE SEQUENCE [schema.]sequence_name
[ {INCREMENT BY | START WITH} integer]
[ {MAXVALUE integer | NOMAXVALUE}]
[ {MINVALUE integer | NOMINVALUE}]
[ {CYCLE | NOCYCLE}]
[ {CACHE integer | NOCACHE}]
[ {ORDER | NOORDER}];
```

All of the proposed clauses are self-explanatory. However, some principles will be described using examples. Create the following sequence.

```
create sequence seq1
start with 100
increment by 10
maxvalue 200
cycle;
```

Such a defined *sequence* has the following parameters:

- starting value (*start with*) is *100*,
- executing *next_val* function means adding the *increment* (value *10*) to the actual value,
- maximal value (*maxvalue*) for associating values is *200*,
- if the maximal value is reached, the *CYCLE* keyword forces the sequence to be restarted.

So, if the sequence is created, we can use it, e.g., in *Select* statements. However, to use such a defined sequence, it must be at *first initialized* by calling its function – *nextval*. If not, an *exception* will be raised:

```
select seq1.currval from dual;
```

```
ORA-08002: sequence SEQ1.CURRVAL is not yet defined in this session
```

Thus, the initialization is done by calling the *nextval* function.

```
select seq1.nextval from dual;
```

```
100
```

Recalling of the function *nextval* will provide value *110*.

```
select seq1.nextval from dual;
```

```
110
```

The last value in the first round is *200*. If the second round is started, what value will be used? One hundred? Or one?

It is necessary to differentiate between starting value and minimal value (which is not set using our defined sequence, so default value “1” will be used automatically). Thus, if there is no minimum value set, the next value will be one. So, let’s assume that the actual value of the sequence is *200*. Calling the function *nextval* will provide the value *1*.

```
select seq1.nextval from dual;
```

```
1
```

If you create another sequence with the minimal value, when reaching 200, the new associated value will be *100*.

```
create sequence seq2
  minvalue 100
  start with 100
  increment by 10
  maxvalue 200
  cycle;
```

See the demonstration of the solution, assume the actual value of the sequence – *200* (*nextval* function has been performed *11* times):

```
select seq2.nextval from dual;
```

```
200
```

```
select seq2.nextval from dual;
```

```
100
```

```
select seq2.nextval from dual;
```

```
110
```

If the **NOCYCLE** keyword is added and maximal value is reached, by calling the *nextval* function, an exception will be raised:

```
create sequence seq3
start with 100
increment by 10
maxvalue 200
nocycle;
```

Once again, assume that the actual value of the sequence is 200. The *exception* will be raised after trying to get the next value:

```
select seq3.nextval from dual;
```

```
200
```

```
select seq3.nextval from dual;
```

```
ORA-08004: sequence SEQ2.NEXTVAL exceeds MAXVALUE and cannot be
instantiated
```

Notice that all sequence properties can use default values (in a standard environment, the default value for sequence property is “1”).

Adding the **CACHE** keyword makes it possible to store a predefined number of consecutive values in memory.

Typically, sequences are associated with identifiers of the objects – the primary key.

A particular value from the sequence is then automatically assigned. Thanks to that, no problem with the uniqueness of the primary key can occur (if no **CYCLE** keyword is used). So, create a *sequence* and assign it to the primary key of the table *student*.

```
create sequence seq_st_id
start with 1
increment by 1;
```

However, is such start position (*I*) correct? Think that some data portions are already stored in that table.

Existing sequence properties can be changed using *Alter sequence* command. It is necessary to adjust starting, respectively actual sequence position (current value) in our case. To get the value by which the sequence should be altered, use the following command (whereas we will use the *nextval* method, the result should be lowered by *I*):

```
select max_value - seq_st_id.nextval -1
from (select max(student_id) as max_value
      from student);
```

```
550943
```

By using *Alter Sequence* command, we can change the actual position of the sequence by using three steps (whereas there is no actual position, which can be set directly). First of all, the current *increment* is changed to a previously obtained value. Then, the current position is shifted to a consecutive one. Finally, the increment is changed to value “1” and the sequence is ready to deal with new primary key values.

```
alter sequence seq_st_id increment by 550943;
```

Now, if you write a query to get the *nextval* of the sequence, the result will be correct: **550944**.

```
select seq_st_id.nextval from dual;
```

```
550944
```

In the end, it is necessary to change the increment step to the value “1”.

```
alter sequence seq_st_id increment by 1;
```

Particular values can be associated with the primary key. The following code shows the result of calling the *nextval* function of the *sequence*.

```
select seq_st_id.nextval from dual;
```

```
550945
```

```
select seq_st_id.nextval from dual;
```

```
550946
```

Altering *sequence* provides a powerful tool for influencing sequence characteristics. All of the clauses of the definition can be updated. There are some examples of the *Alter Sequence* operation:

```
alter sequence teacher_seq MAXVALUE 1500;
```

```
alter sequence teacher_seq NOCYCLE CACHE 5;
```

If the value of the *sequence* is set, we can create the *trigger* to set primary key values.

```
create or replace trigger trig_ins_st
before insert on student
for each row
begin
    :new.student_id := seq_st_id.nextval;
end;
/
```

Notice that since Oracle 12c, DBS allows you to create autoincrement column directly. However, internally, it is managed using *sequence* and *trigger*.

Another example can be based on the management *personal_id* of the *personal_data* table. If you want to add a new person to the system, the primary key must be set before inserting. Therefore, there are two possibilities – generate it or use the explicitly written, if possible. In that case, the condition inside the *trigger* body can look like this:

```
-- ...
if :new.personal_id IS NULL then
    :new.personal_id := GeneratePID
end if;
-- ...
```

As well as other objects, the *sequence* can also be dropped:

```
drop sequence seq_name;
```

10.11.2 Sequence and transaction correlation

As has been already mentioned, values of the *sequences* are often used for primary key definitions. A *transaction* is a base unit of the database system management. It influences

processing. If the transaction is rolled back, all changes associated with the particular transaction are removed. *However, sequences are not affected by the transactions abort – the assigned value of the sequence is not moved backward (lowered).* It can be incremented using the *nextval* function. The position can be exclusively changed using the *Alter sequence* command described in the previous part.

Principles are described in the following example. Create a simple table (*Table1*) containing only one attribute (*val*). Also create sequence (*seq_val*) and use it for inserting 3 rows into a defined table. Then, successfully end transaction using *Commit* command. Insert another 3 rows. Then, rollback the transaction. Get the actual value of the sequence. Is it 3 or 6?

```
create table Table1 (val integer);
```

```
create sequence seq_val;
```

```
insert into Table1 values(seq_val.nextval);  
insert into Table1 values(seq_val.nextval);  
insert into Table1 values(seq_val.nextval);
```

```
commit;
```

```
insert into Table1 values(seq_val.nextval);  
insert into Table1 values(seq_val.nextval);  
insert into Table1 values(seq_val.nextval);
```

```
rollback;
```

```
select seq_val.currval from dual;
```

```
6
```

10.12 DDL triggers

Database systems also provide a technology for *DDL* triggers management. They are associated with the *data definition language* (*DDL – Create, Alter, Drop*) statements. If any of them is executed, the particular trigger is fired automatically.

```
CREATE [OR REPLACE] TRIGGER trigger_name  
  BEFORE | AFTER  
    [ddl_event1 [OR ddl_event2 OR ... ]]  
  ON DATABASE | SCHEMA  
    trigger_body
```

Ddl_event can be *Create, Alter, Drop*. Moreover, the trigger is defined either for *schema* or *database*:

- *on database* – trigger will be fired for all objects in any schema,
- *on schema* – trigger will be fired only for *DDL* operations on particular user objects.

However, there is impossible to associate a specific operation with an exactly defined object:

```
BEFORE DROP study_subjects
```

BEFORE DROP TABLE

To do that, a **When** clause must be used, associate it with the defined table name (a *type of the object must be "table"*).

```
create or replace trigger ddl_trigger
before drop on schema
  when ((ora_dict_obj_name = 'STUDY_RESULTS')
        and (ora_dict_obj_type = 'TABLE'))
begin
  raise_application_error(-20000, 'Such data table cannot be dropped!');
end;
/
```

So, if you try to drop table *Study_Results*, it will not be possible:

```
drop table study_results;
```

```
ORA-00604: error occurred at recursive SQL level 1
ORA-20000: Such data table cannot be dropped!
ORA-06512: at line 2
```

However, if you try to drop table *st_program*, there will be no problem:

```
drop table st_program;
```

```
Table dropped.
```

So let's have the tricky example. Create a trigger, which will *not allow the user to drop any object*:

```
create trigger trig_drop
before drop on database
begin
  raise_application_error(-20001, 'No object cannot be dropped at all.');
```

```
end;
/
```

```
Trigger created.
```

Try to drop the defined sequence. Is it possible?

```
drop sequence seq3;
```

```
ERROR at line 1:
ORA-00604: error occurred at recursive SQL level 1
ORA-20001: No object cannot be dropped at all.
ORA-06512: at line 2
```

No, nor the table, even trigger cannot be dropped.

```
drop table TAB1;
```

```
ORA-00604: error occurred at recursive SQL level 1
ORA-20001: No object cannot be dropped at all.
ORA-06512: at line 2
```

So, as you can see, no object can be dropped at all. So, what to do now? How to solve that problem? Is it even possible to *drop any object*? Sure, the question is positive.

Do you think that the database administrator (*DBA*) can do that? Sure, he can. But it is coded safe, so the object owner can also *drop* the object (only that one!), although it

should be prohibited from the definition. For the object owner, in that case, the trigger will not be fired.

```
Drop trigger trig_ins_st;
```

```
Trigger dropped.
```

10.13 Event triggers

The last category of the triggers is *event trigger*, which can be associated with special events on the database server. Whereas such trigger fires only if the database is available, there are some restrictions on firing time – either just *before* or *after* the event associated or before finishing work of the DBS, signing in or out from the system.

Tab. 10.1: Triggering time & events

Event	Triggering time	Description
Startup	After	Instance starting
Shutdown	Before	Shutting down the instance
Servererror	After	Server error raising
Logon	After	Sign in of the user
Logoff	Before	Sign out of the user

Let's have the following table (*log_table*) consisting information about the user, time of event occurrence, IP address (IP address is obtained using *sys_context('userenv', 'ip_address')* function) and event. The table is created based on the *Select* statement and will be empty, whereas there is a condition that cannot be *True* at all.

```
create table log_table
as
  select user user_name, sysdate occur_date,
         sys_context('userenv', 'ip_address') as ip,
         'xxxxxx' event
  from dual
 where user is null;
```

Let's create triggers to monitor server activities. The first one monitors logons on the database and stores login, actual time, and IP address. The second one watches logoffs.

```
create or replace trigger logon_trigger
  after logon on database
begin
  insert into log_table
    select user, sysdate, sys_context('userenv', 'ip_address'), 'logon'
    from dual;
end;
/
```

```
create or replace trigger logon_trigger
  before logoff on database
begin
  insert into log_table
    select user, sysdate, sys_context('userenv', 'ip_address'), 'logoff'
    from dual;
end;
/
```

Notice that it is impossible to put both triggers together into a single one because one operation should be fired *before*, the second one should be fired *after*.

USER_NAME	OCCUR_DATE	IP	EVENT
KVET_ENG	06.02.2017	158.193.138.18	logon
KVET1	06.02.2017	158.193.138.18	logon
MATIASKO	06.02.2017	192.200.193.1	logoff
SYSTEM	06.02.2017	158.193.138.12	logon

10.14 Practice

This practice aims to create triggers and verify developed functionality and correctness of the results using *Insert*, *Update* and *Delete* statements. During the lab, focus on answering the following questions:

- What should trigger type be defined? *Row*, *statement*, or it does not matter.
- What trigger event should be used? *Before*, *after*, or it does not matter.
- Which record can be used (if available)? *New*, *old*, both, or none.

Be aware once again. Never catch the *exception* in the body of the trigger.

1. Extend the *study_subjects* table using these two attributes – *user* and *execution_date*. Then, create a trigger that stores information about the change to the defined attributes (*user*, *execution_date*). Ensure that those data cannot be directly changed.
2. Create trigger functionality to ensure that no student can register the same subject more than twice (operations *Insert* and *Update*). Verify the functionality. If it is ok, *drop the defined trigger*.
3. Create a *trigger* for cascade changing of the student identifier (*student_id*).
4. Create a *log table* consisting of operations (*Insert*, *Update*, *Delete*) performed in *study_subjects* tables. Information about the *user*, *date*, performed *operation*, and information about the original row should be stored (except for *Insert* statement).
5. Create a trigger, which prohibits *deleting* any row from the *study_results* table. Subsequently, try to remove some data from that table.
6. Deactivate defined trigger. Try to remove data from the *study_results* table. Enable defined trigger.
7. Drop trigger from the previous step.
8. Create a *log table* (*log_table2*) containing this information (*name of the table*, *owner*, *creator*, and *date of creation*). Define trigger to provide such data if a particular operation is executed. Use the following information:
 - *ora_sysevent* – which operation has been performed (in our case, it will be *Create*),
 - *ora_dict_obj_owner* – the owner of the table,
 - *ora_dict_obj_name* – the name of the defined table.
9. Map previous *trigger* solution also for *dropping* commands (use only one trigger to provide desired functionality).
10. Rewrite the previous trigger. It should be fired only if the table is *created* or *dropped* during the *weekend*.

Lab 11 – Relational integrity

Relational integrity is a core element of the validity and reliability of the database system itself. Relational integrity is commonly associated with data consistency. Transaction shifts the database from one consistent image to another, which is also consistent.

The reader of this book will get complex overview and the categorization of the relational integrity – entity, referential, user, column, and domain. When dealing with the user integrity, the focus is done on the whole hierarchy, from the superkey definition, through the primary key candidate and alternative key up to the primary key itself, as one element of the primary key candidate set. Related to the referential integrity, the reader will learn about the referential integrity check protocols, which can be done as part of the statement itself or moved to the end of the transaction. Shifted evaluation brings easier management of the referential integrity (like cascaded update operations) or is used in cases where the relationships between the tables form a reference cycle.

11.1 Introduction

Relational integrity in the area of databases is understood as a meaningfulness and data consistency supported by security and often associated with confidentiality. The integrity itself aims to provide data accuracy, correctness, and value for any changes in the database. Errors or subsequent data inconsistencies may arise from data input, operator errors, program errors, or deliberate damage to the database.

The concept of data consistency is closely related to the concept of integrity, and these terms are often considered synonymous. This is especially true in situations if there are changes in multiple database objects at the same time or when users in the multiuser system manipulate the same data set in parallel, which could ultimately lead to incorrect results. Therefore, the operator of the integrity definition is just a transactional management system that guarantees the transition of a database from one consistent state to another consistent state covered by the operations that manipulate the database's data.

Relational integrity is an inseparable part of the relational model and is currently considered one of the most elaborate areas of relational database systems. Values stored in the database must always represent reality modeled in the proposed system. Moreover, particular values must be correct and meaningful. This implies the need to define integrity rules that allow the DBS to work with real-system constraints.

11.2 Integrity constraints classification

Integrity constraints form design, implementation, and usage rules, which must be applied to data stored in the database. They are based on the conceptual model, as well as, covered by requirements for the modeled information system.

The following classification of integrity constraints represents relational integrity:

- Column integrity C
- User integrity U
- Referential integrity R
- Entity integrity E
- Domain integrity D

Some constraint definitions are optional and can evolve dynamically. However, each database system management must ensure at least these requirements – entity and referential integrity.

11.3 Entity integrity

Ensuring *entity* integrity is an essential requirement for database consistency. This integrity constraint defines the property of the **primary key** in the sense that the primary key must always have a defined value (each attribute forming the primary key must have a defined value – *NOT NULL*). For these purposes, the principle of the primary key definition and process of its selection is defined in the following section.

11.3.1 Primary key candidate

The primary key candidate (*cpk*, *kpk*) is a set of the attributes that meet these conditions:

- **Uniqueness** – there are no two or more data tuples with the same values of the attributes forming the primary key candidate.
- **Minimum** (no redundancy) – no subset of the primary key candidate attributes meets the requirement of uniqueness.

personal_data
personal_id
ICN
name
surname
street
town
zip
nationality

Fig. 11.1: *Personal_data* table

For that table, the following *candidates* can be identified:

CKP₁: personal_id

CKP₂: ICN (identification number of passport)

CKP₃: name, surname – only in assumptions that the pair is unique.

Notice that the pair *{personal_id, ICN}* is unique but is not considered as a *candidate* because of the *minimum* requirement.

The primary key itself is selected from the primary key candidate set to minimize storage requirements or based on the application usage.

In the relational scheme, we designate the primary key with #.

11.3.2 Primary key

Primary key (*PK*) can also be understood as the set of attributes $K = (A_1, A_2, \dots, A_i)$ of the *R* relation, selected from other such potential sets (primary key candidate set), which values uniquely determine the row of the *R* relation. *PK* is minimal (non-redundant).

PK can directly distinguish individual rows. Attributes that are part of a *PK* are called the *key*. Other attributes are called *non-key*.

Each table must be delimited just by only one primary key. Entities that have few attributes and cannot create a primary key are called *weak entities* (the subordinate entity is usually a *weak entity*, and then the primary key of the weak entity is defined as the primary key of the strong entity + discriminator to distinguish weak entities). Entities that have enough attributes are called *strong entities*. However, this problem needs to be addressed when creating a conceptual model.

11.3.3 Alternative key

Alternative key (*AK*) is formed by the set of attributes, which are *primary key candidates* but are not designated as the *primary key*.

11.3.4 Superkey

A super-key is a set of *R* relation attributes that contain a candidate for the primary key.

A super-key is a set of attributes that meet the condition of uniqueness but does not necessarily fulfill the condition of minimalism.

11.4 Referential integrity

The second important and inevitable part of the consistency definition is referential integrity, currently supported by *DDL* statements in most database systems. This constraint describes the relationship between data of two relations. It is based on the foreign key referencing the primary key –a connection between tables.

As already noted, the foreign key is an attribute (or group of attributes), which value is either undefined (*NULL*) or must contain the value of the primary key (unique index) of the referenced table. These tables are usually called *master (parent, principal)* and *slave (child, dependent)*. The *primary key* refers to the *master*. The *foreign key* is associated with *slave* relation.

Referential integrity groups individual cardinality possibilities – *1:1*, *1:N*, *M:N* – see [Lab 4 – Data modeling](#). A particular case of the referential integrity and foreign key definition is just self-relationship. In that case, the foreign key refers to the same table. Thus, just to remind you, self-relationship must always be non-identifying.

11.4.1 Referential integrity rule

The foreign key can acquire the value of the primary key of the referenced table or the undefined (*NULL*) value.

If we have two relations *R1* and *R2*, where the attribute *PK1* is the primary key of the relation *R1*, and the *FK* attribute is in relation *R2* that represents the connection between relations *R1* and *R2*, then the *FK* value is *PK1* or *NULL*. If the *FK* is part of the *primary key* in relation *R2*, then it is impossible to take an undefined value because of the *entity integrity* constraint.

11.4.2 Referential integrity consequences

To ensure database consistency, it is necessary to consider which operations (such as *DELETE*, *UPDATE*) should be rejected or accepted. There are two fundamental questions:

What to do if we try to delete a row for which reference (foreign key in another table) exists?

Let's try to delete the *subject* that some students have enrolled in. The solution could cover these three options:

- To allow such operation. In that case, it is necessary to ensure cascading cancellation of all the rows that refer to the deleted row of the base (master) table.
- To reject such operation completely.
- There may be situations that we want to delete a row from the base table but to keep all rows in the slave relation. How to do that? To comply with the *referential integrity*, the particular *foreign key* value is replaced by the undefined value (*NULL*).

A similar situation can occur if we attempt to change the value of the primary key, to which reference exists in the slave table. In general, there are two possibilities:

- Refuse execution of such operation.
- Allow cascade change based on referential integrity requirements.

In SQL, you can use one of the following options for *UPDATE* and *DELETE* operations to select an operating mode:

- *RESTRICTED*,
- *CASCADE*,
- *NULLIFIED*.

The ***RESTRICTED*** mode means that the operation will be rejected if there is at least one row with an *FK* equal to the *PK* value of the modified (corrected) relation row in the slave table.

If there is a reference path in the data model set in ***CASCADE*** mode for *UPDATE* or *DELETE* operation, then database changes will be reflected in all relations defined in the reference path.

NULLIFIED mode (in some literature, called only *NULL*) means that the operation will be enabled, but the *FK* value will be changed to *NULL*.

11.4.3 Cascade option example

Cascade option changes values of the foreign key in each table, which reference particular primary key value. It can be done directly using multiple DML statements (*Insert*, *Update*, *Delete*) or by the trigger. The following code shows the principles of changing the *student_id* value. It must also be reflected in the *study_subjects* table.

Based on referential integrity, the following operation will not work (it will be executed successfully only if there is no registered subject for a particular student).

```
update student
  set student_id = 550021
  where student_id = 550020;
```


To solve the problem, a new row is inserted into the *student* table with the same values (for a particular student), but the *student_id* value is replaced by a newer, corrected value:

```
insert into student
(select 550021, personal_id, field_id, specialization_id,
      class, st_group, final_date, status, first_date
 from student
 where student_id = 550020);
```

Then a connection can be made for the *study_subjects* table – reference is changed to the newly inserted student.

```
update study_subjects
set student_id = 550021
where student_id = 550020;
```

Finally, an original row in the *student* table is deleted.

```
delete from student
where student_id = 550020;
```

A similar solution can be obtained by trigger definition.

```
create or replace trigger trig_upd_student_id
before update on student
for each row
begin
  update study_subjects
  set student_id = :new.student_id
  where student_id = :old.student_id;
end;
/
```

When dealing with the change of the primary key of the *personal_data* table, similar principles are used. However, in that case, two tables must be managed based on referential integrity – table *contact* and *student*. The straightforward solution will, therefore, require four statements. The solution can look like the following:

```
insert into personal_data
select '841108/3456', name, surname,
      street, town, zip, nationality
from personal_data
where personal_id = '841106/3456';
```

```
update contact
set personal_id = '841108/3456'
where personal_id = '841106/3456';
```

```
update student
set personal_id = '841108/3456'
where personal_id = '841106/3456';
```

```
delete from personal_data
where personal_id = '841106/3456';
```

The trigger can provide an easier solution:

```
create or replace trigger trig_upd_personal_id
before update on personal_data
for each row
begin
  update contact
    set personal_id = :new.personal_id
    where personal_id = :old.personal_id;
  update student
    set personal_id = :new.personal_id
    where personal_id = :old.personal_id;
end;
/
```

11.4.4 Restricted option example

In this section, the *Restricted* option example is proposed. In that case, the *Update* statement operation will be executed successfully only if no reference to the particular primary key is used. We assume that *Cascade* operations have been executed, thus, particular data exist.

Changing the value of the *student_id* can be done, if there are no registered subjects for such student:

```
update student
  set student_id = 550020
  where student_id = 550021
     and student_id not in (select distinct student_id
                           from study_subjects);
```

A similar situation is used for changing the value of the *personal_id*. However, one more condition is used, whereas two table data must be checked.

```
update personal_data
  set personal_id = '841106/3456'
  where personal_id = '841108/3456'
     and personal_id not in (select distinct personal_id from contact)
     and personal_id not in (select distinct personal_id from student);
```

When defining a trigger, reference existence is checked and maintained.

```
create or replace trigger trig_restrict
before delete on personal_data
for each row
declare
  v_count integer;
begin
  select count(*) into v_count
  from contact
    where personal_id = :old.personal_id;
  if v_count <> 0 then
    RAISE_APPLICATION_ERROR(-20000, 'Operation refused - contact table.');
```

```

select count(*) into v_count
  from student
   where personal_id = :old.personal_id;

if v_count <> 0 then
  RAISE_APPLICATION_ERROR(-20000,'Operation refused - student table.');
```

```

end if;
end;
/
```

11.4.5 Nullified option example

The *nullified* option replaces values of the foreign key with *NULL* values if the row with the particular primary key is to be removed. Naturally, other constraints must allow such activity – *foreign key* cannot be denoted as *NOT NULL*.

For exemplary purposes, prepare the following data tables:

Table *subject2* is a copy of the *subject* table. Table *teacher2* is a copy of the *teacher* table. Moreover, attribute *supervisor* is added to the *subject2* table as a reference to the teacher. Corresponding values are loaded from the *study_subjects* table. Notice that the *supervisor* attribute can hold *NULL* values.

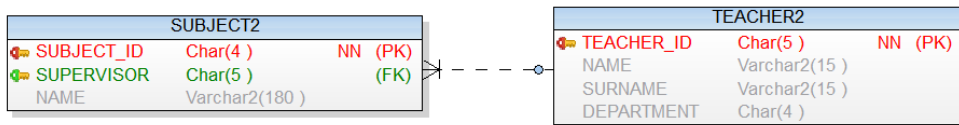


Fig. 11.2: Subject2, Teacher2 table

```

create table subject2 as select * from subject;
create table teacher2 as select * from teacher;

alter table subject2 add supervisor char(5);
alter table subject2 add foreign key (supervisor)
  references teacher2(teacher_id);

update subject2 s2
  set supervisor = (select lecturer
                    from study_subjects s
                   where s.subject_id=s2.subject_id and rownum=1);

commit;
```

The direct approach is reflected by two operations – *Update* statement, which changes foreign key values to *NULL* followed by the *Delete* statement (naturally, principles are same also for *Update* statements):

```

update subject2
  set supervisor = null
  where supervisor in (select teacher_id
                      from teacher
                     where name = 'Rachel'
                       and surname = 'Vargas');

delete from teacher2
  where name = 'Rachel' and surname = 'Vargas';
```

Solution with a trigger is following:

```
create or replace trigger trig_null
before delete on teacher2
for each row
begin
    update subject2 set supervisor = null
    where supervisor = :old.teacher_id;
end;
/
```

11.5 User integrity

User integrity allows defining integrity constraints that support *application logic*. This kind of integrity constraint can be ensured either *declaratively* or *procedurally*. A declarative way means that when the table is defined, *check* constraints of some attributes are defined that extend the *domain* and *column* integrity options. Usually, it is necessary to ensure *user* integrity procedurally because it involves multiple tables, and thus it is impossible to provide such requirement in a *declarative* way. Some examples of user integrity are following:

- The student must be older than 18 when registering to the university.
- A student cannot study more than 15 subjects a year.
- The discarded book cannot be lent anymore.
- The first date of the student registration must be lower than the final date, etc.

It is usually secured by the triggers.

```
create or replace trigger trig_age
before insert or update on student
for each row
begin
    if get_age(:new.personal_id) < 18 then
        RAISE_APPLICATION_ERROR('-20000','Too young person to be a student');
    end if;
end;
/
```

Notice that the *get_age* function is user-defined.

11.6 Column integrity

For each table attribute, it is possible to define *additional column constraints* besides the *domain* integrity constraints.

Column integrity constraints are following:

1. Additional constraints for a range of values that are a subset of the domain,
2. **NULL** or **NOT NULL**,
3. **DISTINCT** or **DUPLICATE**.

For each attribute, it is possible to define further limitations of the range of allowable values that *domain* integrity constraints have defined. Each column may or may not acquire a *NULL* value (undefined value). Moreover, it is also possible to determine whether attribute value should be unique or can hold duplicate value in the table scope.

The following example shows the constraint on the *study_subjects* table:

```
alter table study_subjects modify lecturer NULL;
```

11.7 Domain integrity

Domain integrity represents a set of integrity constraints that share all attribute values associated with this domain.

Domain integrity restrictions are:

- data type,
- set of permissible values,
- sortability – whether the relational operator $>$, $>=$, $<=$ or $<$ can be used to compare domain values.

Domain can be enhanced by the Check constraint discussed in [chapter 4](#).

11.8 Integrity constraints controlling and processing

Because of the result correctness necessity, it is necessary to process integrity constraints properly at each step of the processing. In relational databases, two stages can be distinguished:

- **Column** – the integrity constraint is never checked later than at the end of any relational type request processing. Typically, the request is part of the application program or is interactively specified by the user. Thus, it is managed automatically by the database manager.
- **Transactional** (multitable) – control mechanisms are launched at the end of the transaction, which includes the request.

Individual integrity constraint definitions are part of the system tables, managed automatically. The following algorithm is used:

1. At the beginning of the processing, the database manager determines which constraints and types are related to the request.
2. Column constraints concerned with the one table are identified.
3. Before the processing completion, the database manager determines whether the specified requirement matches defined column integrity constraints.
4. Database manager identifies and processes multitable constraints.
5. Before the change confirmation, such defined multitable constraints are checked.

11.9 Practice

1. Change the schema of the table *study_subjects*, that attribute *lecturer* can hold *NULL* values. Which integrity type is covered by that functionality?
2. Change the schema of the table *study_subjects*, that attribute *ects* cannot hold *NULL* values. Moreover, a particular value cannot be negative. Which integrity type is covered by that functionality?
3. Ensure that the number of reached *ects* for the student of a particular subject is the same as defined in the *subject_year* table (based on *school_year* and *subject_id*) or *zero* if such person has already passed that subject successfully during his previous study. Which integrity type is covered by that functionality?
4. Ensure that the student cannot register for the subject, which he passed sooner successfully. Which integrity type is covered by that functionality?

5. Ensure that the value of the *final_date* attribute is higher than a *first_date* attribute value. Which integrity type is covered by that functionality?
6. Ensure that the attribute *status* of the *student* can hold only these values – *S*, *E*, *A*, and *X*.
 - student.status:
 - S = student (actual),
 - E = ended successfully,
 - A = aborted,
 - X = fired due to disciplinary commission decision.

Which integrity type is covered by that functionality?

7. Change the value of the *subject_id* from “*BI06*” to “*BX06*” (notice that *BX06* does not exist). Is it possible to do it with only one *Update* statement? If not, why? Which integrity constraint type has been raised?
8. Change the value of the *subject_id* from “*BI06*” to “*BL06*” (notice that *BL06* exists). Is it possible to do it with only one *Update* statement? If not, why? Which integrity constraint type has been raised?
9. Solve the previous problem by trigger definition.
10. Try to *insert* a new person into the *personal_data* table without the *personal_id* value (it will be denoted as *NULL*). Is it possible? If not, why? Which integrity constraint type has been raised?

Lab 12 – Views

*The view is a named **Select** statement, which can be referenced as a common table. In this lab, the reader will get the **syntax** overview, usage in terms of producing only a subset of the original data and triggers, which can be associated with the views. Generally, views are complex and formed by multiple tables. Thus, **INSTEAD OF trigger** types are defined just to replace the original view change operation into multiple operations respecting the integrity.*

*By using views, it is, in principle, allowed to operate the data, which will not be visible through the view. Therefore, the reader will be emphasized by the **CHECK OPTION** clause, which checks the original and inherited **Where** clauses anytime the data are to be manipulated (Insert, Update or Delete).*

12.1 Introduction

The **view** is a logical data object associated with the **Select** statement (usually complex, dealing with multiple tables and aggregations). The **view** itself does not contain any data. Thus, when dealing with a view, it is automatically replaced by a defined **Select** statement during the execution. Characteristics are stored in the data dictionary ([Lab 14 – Data dictionary views](#)).

12.2 Syntax

```
CREATE [OR REPLACE] [FORCE | NOFORCE]
VIEW [schema.]name [(column_alias1, [, ... ])]
AS select_statement
   [WITH [READ ONLY | CHECK OPTION [CONSTRAINT constraint_name]]]
|
CREATE [OR REPLACE] VIEW [schema.]name [(column_alias1, [, ... ])]
AS select_statement
   [WITH [CASCADED | LOCAL] CHECK OPTION]
```

SCHEMA – defines the name of the schema under which the view will be created. If omitted, the current user schema is used.

OR REPLACE keyword forces the system to redefine view if existing. However, some database systems do not support that keyword. In that case, it is necessary to *drop* the existing view and *create* a new one. Notice that if the object has been created without this keyword, it is impossible to replace it later. Therefore, to redefine it – it must be dropped (*drop* command) and replaced by the new one (*create* command).

FORCE keyword allows you to create the view without raising errors during the compilation, even if the **SELECT** statement encapsulated in it includes references to objects (tables, views), which do not exist at that time, respectively the user has no particular privileges.

NOFORCE is an implicit value (inverse option to *Force*). In that case, it is possible to define view only if it passes all control mechanisms – used tables must exist and can be directly queried by the particular user.

READ ONLY keyword ensures that no destructive *DML* statements (*Insert*, *Update*, *Delete*) can be made using such a defined view.

CHECK OPTION – ensures passing conditions defined in the **WHERE** clause during executing destructive *DML* statements. Using this keyword, you cannot change data values that will not be visible using that view. Thus, it is impossible to manage not handled (invisible) data for the defined view.

CONSTRAINT keyword allows you to name the constraint.

CASCADE – provides condition checking by derived views.

LOCAL – condition control mechanism is restricted only for actual limitations defined in the view (regardless of the inherited conditions).

The following code shows a simple example – *name* and *surname* are selected from the *personal_data* table.

```
create view v1
as
  select name, surname
  from personal_data;
```

After its definition, it can be used as a standard table in the *Select* statement:

```
select * from v1;
```

The second example extends the view of the gender definition. Each attribute formed by the function must have its alias – the new name of the column for referencing in the query.

```
create view v2
as
  select name, surname,
         decode(substr(personal_id, 3, 1), 5, 'F', 6, 'F', 'M') as sex
  from personal_data;
```

If no alias is added, an exception will be raised, and no view will be created.

```
create view v2
as
  select name, surname,
         decode(substr(personal_id, 3, 1), 5, 'F', 6, 'F', 'M')
  from personal_data;
```

```
ERROR at line 3:
ORA-00998: must name this expression with a column alias
```

Moreover, if keyword **OR REPLACE** is used, if no alias is defined, the original view will remain valid.

12.3 Exceptions

Exception emphasis must be given to the functions, which can raise *exceptions*. Let's have a simple example. It creates a view consisting of three attributes – name, surname, and date of birth.


```
create view v3
as select name, surname,
        to_date(substr(personal_id, 1, 2)
                || mod(substr(personal_id, 3, 2), 50)
                || substr(personal_id, 5, 2), 'RRMMDD') as birth_date
from personal_data;
```

Such a view will be naturally created. However, an exception will be raised if the query based on this view will contain data that *personal_id* value cannot be transformed to *date of birth*. The exception itself will depend on properties and the current situation (invalid number, not a valid month, etc.).

```
SQL> select *      from v3;
select *      from v3

ERROR at      line 1:
ORA-01722:    invalid number
```

Fig. 12.1: Exception – invalid number

```
SQL> select *      from v3;
select *      from v3
              *

ERROR at      line 1:
ORA-01843:    not a valid month
```

Fig. 12.2: Exception – not a valid month

Now, we will highlight the problem of the invalid month. An *exception* can be raised. The following example shows and highlights the consequences of the implicit conversions. Zero values from the first positions are automatically removed if the *day*, *month*, and *year* elements are treated as numbers. Thus, the input does not contain six digits. However, how to convert it to the date subsequently? In our example, the first and second digits express year, the third and fourth characterize month (for the women, we have to *subtract* the value 50). The last two digits represent the day. But as you can see in the following example, the first zero value is removed (separately from the year, month, and day value). Thus, the length of input values is not 6.

To see the problem, let's create another view.

```
create view v4 as
select name, surname,
        substr(personal_id, 1, 2)
        || mod(substr(personal_id, 3, 2), 50)
        || substr(personal_id, 5, 2) as birth
from personal_data;
```

The query result is the following.

NAME	SURNAME	BIRTH
Michael	Smith	601224
Darl	Peterson	601224
Peter	Allison	74210
Paul	Casey	550947
Peter	Roger	781015
Jack	Robinson	791229
Mark	Bailey	80407
Thomas	Hall	81101

To convert values to *date* data type, an *exception* will be raised. The consequence of the raised exception in the query is the fact no data are provided (even though only one row is “corrupted” and the rest are correct). Simply, an exception has been raised, resulting in the query to rollback. Therefore, the question of thinking – how would you solve it? How would avoid raising exceptions? Is it even possible? Sure, it is.

In principle, we have various possibilities how to solve that problem. All of them are based on converting the value to the string format because they do not suffer such deficiency. The next two examples show the solution principles. The first one is based on elements of the date separation by characters (like dots, slashes, dashes, etc.). It prevents the possibility of automatic conversion to a numeric format. Another solution is to define conversion explicitly by calling the *to_char* method with two parameters – *input_value* to be converted and the *format* itself (see chapter [Conversion functions - TO_CHAR](#)):

```
to_char(input_value, [format])
```

Value “99” as format forces the system to use two digits in output format (number of “9” expresses the number of digits in the result set).

```
create view v5
as select name, surname,
        to_date(to_char(substr(personal_id, 1, 2), 99) ||
               to_char(mod(substr(personal_id, 3, 2), 50), 99) ||
               to_char(substr(personal_id, 5, 2), 99),
               'RRMMDD') as birth_date
from personal_data;
```

A similar solution will be obtained, if managing input date elements as strings by using character delimiters (in the following case, character “.” is used):

```
create view v5
as select name, surname,
        to_date(substr(personal_id, 1, 2) || '.' ||
               mod(substr(personal_id, 3, 2), 50) || '.' ||
               substr(personal_id, 5, 2), 'RR.MM.DD') as birth_date
from personal_data;
```

12.4 Managing data in views

Let’s have the following simple example:

```
create view v1
as select name, surname
from personal_data;
```

What will happen if you update data using a view? Will the table/view be updated? Of course.

```
update v1
  set name = 'Philippe'
  where name = 'Thomas' and surname = 'Hall';
```

```
1 row updated.
```

However, to be sure, check the values by querying view *v1*.

```
select * from v1 where surname = 'Hall';
```

NAME	SURNAME
Philippe	Hall

However, what about the table data? Will they be the same, or original value (“*Thomas*”) will be present? Why?

```
select name, surname
  from personal_data
  where surname = 'Hall';
```

NAME	SURNAME
Philippe	Hall

Sure, they must always be the same, whereas view itself is the only representation of the stored *Select* statement.

What will happen if you delete a row using a view? The principle is the same as the *Update* statement.

On the other hand, what about executing the *Insert* statement? Is it even possible? Why? Why not? Under what conditions is it possible? Remember the prerequisites for the *Insert* statements.

Solution – it is possible to add new data only if all constraints are met. Let’s have a practical example. Use the previously defined view (*v1*) based on the *name* and *surname* attributes of the *personal_data* table.

```
create view v1
  as select name, surname
  from personal_data;
```

Try to *Insert* a new row into the table using such a view. Is it possible? No, at all...

```
insert into v1 values('Michael', 'Flower');
```

```
ERROR at line 1:
ORA-01400: cannot insert NULL into
        ("KVET_ENG"."PERSONAL_DATA"."PERSONAL_ID")
```

However, if you define a view consisting of the *personal_id* attribute, the *Insert* statement can be executed without raising an exception (if *entity* and *domain* integrity constraints are passed).

Notice that the view must be *dropped* before redefinition, whereas it has not been created with the **Or Replace** keyword, nor it will not help us if we write **Create Or Replace** now. Simply, if there is no **Replace** keyword in the beginning, the only solution is to *drop* the object and *create* a new one.

```
drop view v1;
```

```
create view v1
as select personal_id, name, surname
from personal_data;
```

```
insert into v1 values('601224/6526', 'Mark', 'Flower');
```

```
1 row created.
```

Naturally, it is possible to add new data into a *personal_data* table using the defined view. Values for attribute *personal_id*, *name*, and *surname* are listed. The rest of the attribute values (*street*, *town*, *zip*, *nationality*) will hold *NULL* values, whereas they are not defined in the *Insert* statement. Notice that for attributes *name*, *surname*, *street*, *town*, *zip*, and *nationality*, *NULL* values are applicable).

```
insert into v1(personal_id) values('601224/6537');
```

```
1 row created.
```

As evidence, this is the control *Select* statement:

```
select personal_id, name, surname
from v1
where personal_id like '601224/%';
```

And the provided results:

<i>PERSONAL_ID</i>	<i>NAME</i>	<i>SURNAME</i>
601224/6526	Mark	Flower
601224/6537	(null)	(null)

Another critical question is whether it is possible to insert new data into the table using a defined view if it does not contain all *NOT NULL* attributes. The answer is undoubtedly positive. However, how would you do it?

Let's have the view consisting of actual students (*status* of the student is “S”). Such view will be based on *personal_id*, *field_id*, and *specialization_id* attributes.

```
create or replace view v_student
as select personal_id, field_id, specialization_id
from student
where status = 'S';
```

Using this view, it is impossible to add new data to the student table. Why? Because the value of the primary key is not provided. However, we can define a *trigger*, which will replace the value *student_id* automatically. Thanks to that, the *Insert* statement will pass.

See the following example. We will create a sequence for providing a new student identifier (maximal number of the *student_id* is **550945**, therefore, it will start with the consecutive value) and associate it with the trigger to automatize operation.

```
create sequence seq_student
start with 550946;
```

```
create or replace trigger trig_st
before insert on student
for each row
begin
:new.student_id := seq_student.nextval;
end;
/
```

```
insert into v_student values('740210/6525', 200, 2);
```

What about the real data in the *student* table? *Student_id* will be provided using the *sequence* and *trigger*. However, be aware, although a *view* has been created based on actual student condition (*status='S'*), such condition *IS NOT* copied to the new row image.

STUDENT_ID	PERSONAL_ID	FIELD_ID	SPECIALIZATION_ID	CLASS	ST_GROUP	FINAL_DATE	STATUS	FIRST_DATE
550947	740210/6525	200	2	(null)	(null)	(null)	(null)	(null)



12.5 Attribute name redefinition in views

As it has been partially mentioned, each attribute must have its name, by which it can be identified in queries. If the attribute is function-dependent, it must have its alias. Moreover, such defined views should be READ ONLY – you cannot change the output value of the function, can you? Definitely no.

However, it is possible to rename the column in the view definition using the same way as for function. Thus, it is possible to rename any attribute name.

```
create view v2
as select name, surname as last_name, personal_id as pid,
       decode(substr(personal_id, 3, 1), '5', 'female',
              '6', 'female',
              'male') as sex
from personal_data;
```

Another solution is a bit syntactically different. In that case, new attribute names are placed before the *Select* definition itself.

```
create view v2(name, last_name, pid, sex)
as select name, surname, personal_id,
       decode(substr(personal_id, 3, 1), '5', 'female',
              '6', 'female',
              'male')
from personal_data;
```

12.6 Check option clause

Let's have the following view:

```
create view v3
as select name, surname, personal_id
from personal_data
where surname like 'S%';
```

Is it possible to add new data to the table using this view? Sure, it is if it passes *personal_id* constraint checking (it is the primary key, so it should be *unique*

and *NOT NULL*). However, this view lists only persons with a surname passing the format: *'S%'*.

So, try it, insert these two rows. What will happen? Data will be inserted successfully.

```
insert into v3 values('Simone', 'Smith', '845210/6525');
```

```
insert into v3 values('John', 'Bush', '860412/6536');
```

However, what about *Select* statements? Will inserted rows be visible using a defined view? Whereas the view is an only predefined *Select* statement, the answer is NO. But data will be present in the table:

```
select name, surname, personal_id
from v3
where personal_id like '860412%';
```

no rows selected

```
select name, surname, personal_id
from personal_data
where personal_id like '860412%';
```

NAME	SURNAME	PERSONAL_ID
John	Bush	860412/6536

To prevent adding (or changing) “invisible” data of the table using view, it is possible to add keyword **WITH CHECK OPTION** to the view definition, ensuring managing only data, which pass conditions also after data changes:

```
create view v4 as select personal_id, name, surname
from personal_data
where surname like 'S%'
WITH CHECK OPTION;
```

It is possible to add a new person, whose surname starts with letter “S”:

```
insert into v3(personal_id, name, surname)
values('930930/7426', 'Frederico', 'Smith');
```

1 row created.

However, it is not possible to add a person whose surname is *'Ducato'*. It would raise the following exception: **ORA-01402: view WITH CHECK OPTION where-clause violation**.

```
insert into v3(personal_id, name, surname)
values('860712/6475', 'Frederico', 'Ducato');
```

Let’s have another example.

The first view will consist of *name*, *surname*, and *personal_id* of the people, whose surname starts with “S”:

```
create view view_person1
as select name, surname, personal_id
from personal_data
where surname like 'S%';
```

Using this view, any data respecting primary key constraint can be added. Thus, the *surname* can start with any letter:

```
insert into view_person1
values('Carol', 'Matiasco', '770724/2227');
```

```
1 row created.
```

Then, create a new view derived from the defined one (by using **WITH CHECK OPTION** keyword):

```
create view view_person2
as select * from view_person1
where personal_id like '75%'
WITH CHECK OPTION;
```

New data cannot be added if they do not meet specified conditions using a previously defined view. Specifically, there is one direct condition (*personal_id like '75%'*), and the second one is derived (*surname like 'S%'*). All of them must be passed to allow the user to add new data (a similar principle is also for *Update* statements).

Let's have the following examples, think, whether it is possible to *Insert* or not (solution is always above the statement).

```
insert into view_person2
values('Carol', 'Matiasco', '790501/2227');
```

Such a row cannot be added because it does not reflect the first (*surname*) nor the second condition (*year of birth*).

```
insert into view_person2
values('Carol', 'Smith', '770501/2227');
```

It will not be executed successfully due to *personal_id* restrictions.

```
insert into view_person2
values('Carol', 'Smith', '750501/2229');
```

These data will be inserted. No error will be raised.

Thus, remember that **WITH CHECK OPTION** clause controls also derived conditions.

12.7 Read only view

This keyword ensures, that no data can be changed using such a defined view. No *Insert*, *Update* and *Delete* statements execution is allowed:

```
create view v5 as select personal_id, name, surname
from personal_data
where surname like 'S%'
WITH READ ONLY;
```

Any attempt for destructive DML operation will fail:

```
insert into v5 values('900101/0095', 'Simone', 'Bris');
```

```
ERROR at line 1:
ORA-42399: cannot perform a DML operation on a read-only view
```

12.8 View based on multiple tables and triggers

The view can be based on data from multiple tables. However, is it possible to *Insert* new data using such a defined view?

Let's have the following example.

```
create or replace view view_student
as select name, surname, personal_id, student_id, field_id,
specialization_id
from personal_data join student using(personal_id);
```

```
insert into view_student
values('George', 'Smith', '440922/9220', 552312, 202, 0);
```

An *exception* has been raised because of dealing with multiple tables using one *Insert* statement. As we have described sooner, each destructive *DML* statement can manage only one table, thus, it is not possible. On the other hand, specialized tools can also be defined to provide desired functionality and cover that problem.

12.9 Triggers associated with views

Trigger is a specific functionality associated with the change data operations. It can also be correlated with the view replacing the original written statement. As stated, individual *Insert*, *Update* or *Delete* statements must deal just with one table, which is not common in complex views. Thus, original statement (referencing the view) is divided into multiple operation respecting the table structure, as well as referential integrity. *Trigger* associated with the view is replacing the original statement, therefore the firing option is delimited by the *Instead Of* keyword.

```
create or replace trigger trig_st_view_ins
INSTEAD OF INSERT on view_student
begin
insert into personal_data(name, surname, personal_id)
values(:new.name, :new.surname, :new.personal_id);
insert into student(student_id, personal_id, field_id,
specialization_id)
values(:new.student_id, :new.personal_id, :new.field_id,
:new.specialization_id);
end;
/
```

Integrity control mechanisms are launched after the *trigger* execution. Thus the order of operations is essential; reference for the *personal_data* table must be inserted after the value of the corresponding primary key in the *personal_data* table. Keyword *INSTEAD OF* reflects the replacement of real operation with the trigger body.

Similar to the previous example, also *Delete* statement based on more than one table must be replaced using trigger functionality:

```
create or replace view view_student
as select name, surname, personal_id,
student_id, field_id, specialization_id
from personal_data join student using(personal_id);
```

The solution is based on physically replacing the original statement with two physically executed *Delete* statements – from the table *student* and from the table *personal_data*

(the order of operations is also significant). Referencing to actual row is provided by the *:old.personal_id* value.

Notice that there is no **FOR EACH ROW** keyword, but it works correctly. The reason is that each trigger firing *instead of operation* on view is *automatically reflected* as **FOR EACH ROW**, so it is not necessary to define it explicitly (but you can if you wish).

```
create or replace trigger trig_st_view_del
  INSTEAD OF DELETE on view_student
begin
  delete from student
    where personal_id = :old.personal_id;
  delete from personal_data
    where personal_id = :old.personal_id;
end;
/
```

```
delete from view_student
  where personal_id = '440922/9220';
```

12.10 Summary

- If the *Select* statement forming the view contains the *primary key* and all *NOT NULL* attributes of the table, then *Insert* statement to the particular table can be executed successfully. Naturally, it must be based on only one table, otherwise, the *trigger* must be defined.
- If the view is defined as *READ ONLY*, no *Insert*, *Update* and *Delete* statements can be performed.
- If the view is defined using *WITH CHECK OPTION* keyword, then the particular data must meet all defined (and also derived) conditions (in the *Where* clause).
- If the *Select* statement forming the *view* consists of multiple tables, destructive operations must be replaced by several operations performed by the trigger type *INSTEAD OF*.

12.11 Practice

1. Define the view **view_st** containing the name, surname, study group, and the actual age of the *students*.
2. Define the view **view_tch** containing the name list (name, surname) of the *teachers* who taught some *subject* in the *school year 2007*. Try to add a new teacher and some *lectured subject* using a defined view, which passes the defined condition of the view.
3. Define the view **view_teacher** based on attributes name, surname. Is it possible to *Insert* new data into the particular table using that defined view? If not, restructuralize that view.
4. Define the view **view_person** based on *name*, *surname*, and *personal_id* attributes. Use only *personal_data* table.
5. Choose *one of the students* (whatever), remember his *name*, *surname*, *personal_id*, and *student_id*. Next, remove his *student* data (emphasizing all reflected tables to ensure consistency). Next, remove his data using the view **view_person**. What happened? What information about him can be found and where?
6. Define the view **view_student** based on *name*, *surname*, *personal_id*, and *student_id* attributes.

7. Choose a random *student*, remember his *student_id*, *name*, *surname*, and *personal_id*. Then, remove all his *studied subjects*. Then, *delete* his data using the view based on the *student_id* attribute. What happened? Check his data using the view and by querying the particular table.
8. Choose a random *student*, remember his *student_id*, *name*, *surname*, and *personal_id*. *Delete* all his studied subjects. Then, *delete* his data using the view based on the *personal_id* attribute. What happened? Check his data using the view and by querying the direct table.
9. Create a **trigger** to ensure consistent data removal from the view **view_student**. Then, check the correctness of the results.
10. Define the view **view_bachelor_subj** – list of subjects for the bachelor study (use the table *subject*, the first letter of the *bachelor subject* is “B”). **Do not use keyword WITH CHECK OPTION.**
11. Try to add a new subject using defined view – *XX01 – Database architectures*. Is it possible? Will defined data be visible using the view?
12. Define the view **view_bachelor_subj2** – list of subjects for the bachelor study (use the table *subject*, the first letter of the *bachelor subject* is “B”). **Use keyword WITH CHECK OPTION.**
13. Try to add a new subject using defined view **view_bachelor_subj2** – *BX01 – Database architectures*. Is it possible? Will defined data be visible using the view?
14. Try to add a new subject using defined view **view_bachelor_subj2** – *XX02 – Database architectures*. Is it possible? Will defined data be visible using the view?

Lab 13 – Date and Time value management

The final part related to the Select statement is covered by this lab introducing the Date and Time management complexity. It focuses on the Date and time data types and available existing functions, focusing on the limitations. After studying this lab, the reader will understand the complexity of Date and Time management, covered by the time zone management, regions, and NLS parameters. There is also discussion about the duration management, operated by two values or interval data types. The definitions are covered by really many examples focusing on individual problems related to Date and Time management.

Database Systems (DBS) currently offer comprehensive support for working with time in the form of data types and methods. In this chapter, we will describe the possibilities of the Oracle database system, which is most often used due to its complexity and offers of sophisticated solutions, by keeping SQL standards. We also highlight the main mistakes when dealing with *Date* attribute values. Notice that corresponding data types and methods may vary in other database systems, especially in performance, but the principles remain the same.

Generally, database systems provide four categories of data types – **Date**, **Timestamp**, **Time** (PostgreSQL), and **Interval**. Data type *Interval* represents the duration itself with no specific image at a time (we cannot determine the start point of the validity, only duration itself is maintained).

Data types **Date**, **Time**, and **Timestamp** are similar. However, they differ in the way of storing data and granularity. In comparison with other database systems, the data type **Date** in DBS Oracle includes not only the date itself but also the time up to the level of seconds. Thus, value consists of a component of the *year, month, day, hour, minute, and second*: *YYYY-MM-DD HH:MI:SS*. Other DBS have a specific data type for *date* and *time* (*elements are separated*). Notice that result of two *Date* values subtraction is the number of days between them (see chapter 13.9 Get the difference between Date values).

The **timestamp** data type can work with finer granularity – specifically with the second part (*fraction*) up to the level of *nine decimal places*. The optional parameter *n* in the declaration of an attribute or a variable of type **timestamp(n)** defines the scope and precision by a number of decimal places (*fraction*). If the value parameter is not specified, DBS automatically uses a parameter with a value of 6 (six decimal places for the second part), so data type **Timestamp** and data type **Timestamp(6)** are identical. The **timestamp** data type is stored as the time elapsed from a defined period – *1.1.1970* and therefore allows the definition of variable size.

It is also possible to define time zones, either as a **timestamp(n) with local time zone** or **timestamp(n) with time zone**. The differences are described in the following example.

Let's have 2 sessions (*S1* and *S2*). In one of them, set the time zone to "-7: 0". Then, create a simple table (*T1*) characterized by a single attribute – sequentially: **Timestamp**, **Timestamp(n) with local time**, and **Timestamp (n) with time zone**. Let's *insert* to such table the current value of the *systimestamp* from both sessions. Note the differences by executing the *Select* statement from both sessions. The order or individual commands to be executed is important.

```
-- session 1
alter session
  set nls_date_format='DD.MM.YYYY
    HH24:MI:SS';
alter session
  set nls_timestamp_format=
    'DD.MM.YYYY HH24:MI:SS.FF';
select sysdate from dual;
```

```
SYSDATE
-----
15.03.2021 13:16:09
```

```
create table T1 (val timestamp);

insert into T1
  values(systimestamp);

commit;
```

```
-- session 2
alter session
  set nls_date_format='DD.MM.YYYY
    HH24:MI:SS';

alter session
  set nls_timestamp_format=
    'DD.MM.YYYY HH24:MI:SS.FF';
```

```
alter session
  set time_zone='7:0';
```

```
insert into T1
  values(systimestamp);

commit;
```

By executing the *Select* statements, the following values are obtained – values are the same:

```
-- session 1

VAL
-----
15.03.2021 13:16:09.185371
15.03.2021 13:17:43.964973
```

```
-- session 2

VAL
-----
15.03.2021 13:16:09.185371
15.03.2021 13:17:43.964973
```

Now, repeat the previous example, but replace the data type *Timestamp* and use *Timestamp with time zone*. Before the processing, the previously defined table is *dropped*.

```
-- session 1

alter session
  set nls_date_format=
    'DD-MM-YY HH24:MI:SS';

alter session
  set nls_timestamp_format=
    'DD-MM-YY
      HH24:MI:SS.FF';

create table T1
(val timestamp
 with time zone);

insert into T1
  values(systimestamp);

commit;
```

```
-- session 2

alter session
  set nls_date_format=
    'DD-MM-YY HH24:MI:SS';

alter session
  set nls_timestamp_format=
    'DD-MM-YY
      HH24:MI:SS.FF';

alter session
  set time_zone='7:0';

insert into T1
  values(systimestamp);

commit;
```

```
-- session 1

VAL
-----
15-MAR-21 13:24:32.840097 +01:00
15-MAR-21 13:24:37.913932 +01:00
```

```
-- session 2

VAL
-----
15-MAR-21 13:24:32.840097 +01:00
15-MAR-21 13:24:37.913932 +01:00
```

Compared to the previous example, the *time zone value* is obtained with emphasis on time zone parameter settings. As you can see, if the time zone on the session is changed, the particular obtained value (time zone) is still the same.

By modifying the data type to *Timestamp with local time zone*, local session values are transformed using the time zone.

```
-- session 1

alter session
  set nls_date_format=
    'DD.MM.YYYY HH24:MI:SS';

alter session
  set nls_timestamp_format=
    'DD.MM.YYYY
      HH24:MI:SS.FF';

create table T1
(val timestamp
 with local time zone);

insert into T1
  values(systimestamp);

commit;
```

```
-- session 1

VAL
-----
15.03.2021 13:41:30.882184
15.03.2021 13:41:39.668943
```

```
-- session 2

alter session
  set nls_date_format=
    'DD.MM.YYYY HH24:MI:SS';

alter session
  set nls_timestamp_format=
    'DD.MM.YYYY
      HH24:MI:SS.FF';

alter session
  set time_zone='7:0';

insert into T1
  values(systimestamp);

commit;
```

```
-- session 2

VAL
-----
15.03.2021 19:41:30.882184
15.03.2021 19:41:39.668943
```

In this case, provided values reflect the local time zone. Thus, if changed, particular values are recalculated to current settings on the client-side.

The above examples show that the data type *Timestamp with local time zone* should be used if calendar data must be synchronized with different time zones. Suppose you are planning some consultations with people from other regions. In that case, such an attribute can ensure that everyone will get the correct result transformed using his defined time zone regarding the local time used by all of them.

As already mentioned, data types *Date* and *Timestamp* are similar and directly transformable by implicit conversion methods.

The result of the *sysdate* (or by calling *current_date*) function execution is the value of the *Date* data type.

```
create table t1 as select sysdate val from dual;
desc t1;
```

Name	Null	Type
VAL		DATE

If you want to create the table based on the *Select* statement using functions, do not forget to define column *alias* (for consecutive naming). Otherwise, an *exception* will be raised.

The output data type of the *systimestamp* or *current_timestamp* function is *Timestamp*.

```
create table t1 as select current_timestamp val from dual;
desc t1;
```

Name	Null	Type
-----	-----	-----
VAL		TIMESTAMP(6) WITH TIME ZONE

If the *current_timestamp* function is used in the *Select* statement, the value will also be extended by the time zone spectrum.

```
select current_timestamp from dual;
```

VAL

16.03.2017 04:34:15.669952 +02:00

Vice versa, if we want to get local value, the *localtimestamp* function should be used (value will be normalized base on client timezone settings).

```
select current_timestamp from dual;
```

VAL

16.03.2017 06:34:15.669952

Timestamp attributes can be transformed each other also in tables by changing the granularity and precision of the value. However, it can be done only if a particular table is empty or does not contain any *NOT NULL* value.

```
alter table t1 modify val timestamp(8);
```

Otherwise, one of these exceptions will be raised (based on performed activity).

```
SQL Error: ORA-01439: column to be modified must be empty to change the
datatype
01439. 00000 - "column to be modified must be empty to change datatype"
SQL Error: ORA-30082: datetime/interval column to be modified must be
empty to decrease fractional second or leading field precision
30082. 00000 - "datetime/interval column to be modified must be empty to
decrease fractional second or leading field precision"
*Cause: datetime/interval column with existing data is being modified
to decrease fractional second or leading field precisions.
*Action: Such columns are only allowed to increase the precisions.
```

Data types *Date* and *Timestamp* are similar, which means that the database system uses automatic conversion between them by increasing, respectively decreasing the granularity and format. Therefore, all these following cases will work correctly. The results of the first *Select* statement are formatted using the method *to_char* based on the granularity of seconds (limitation of the data type *Date* definition). The difference occurs when viewing the results based on the second part (*fraction*).

```
Create table T1 (val date);
Insert into T1 values(sysdate);
Select to_char(val, 'DD.MM.YYYY HH24:MI:SS') from t1;
```

16.03.2017 04:55:12

```
Insert into T1 values(systimestamp);
Select to_char(val, 'DD.MM.YYYY HH24:MI:SS') from t1;
```

```
16.03.2017 04:55:17
```

```
Create table T1 (val timestamp);
Insert into T1 values(sysdate);
Select to_char(val, 'DD.MM.YYYY HH24:MI:SS:FF') from t1;
```

```
16.03.2017 04:55:12.000000
```

```
Insert into T1 values(systimestamp);
Select to_char(val, 'DD.MM.YYYY HH24:MI:SS:FF') from t1;
```

```
16.03.2017 04:55:17.285000
```

13.1 NLS parameters & session format

NLS parameters define *National Language Support* and locale for server and also client environment determining format and language of the result set. There are four ways how *NLS parameter* values can be specified and set:

- Setting **initialization parameters** in the parameter file (*spfile*, *pfile*) specifying the default session *NLS* environment. These settings do not affect the client-side; they control the server's behavior and are the default for the client.
- Setting **environment variables** on the client-side influencing behavior of the client. It can override used default values for the session.
- **ALTER SESSION parameters**, which are used for changing session *NLS* parameters. It can override the initialization parameters as well as environment variables.
- **SQL function parameters** – *NLS parameter* values can be explicitly coded in the *SQL* function invocation to determine the provided result set format.

The following diagram shows the priorities and properties of overriding.

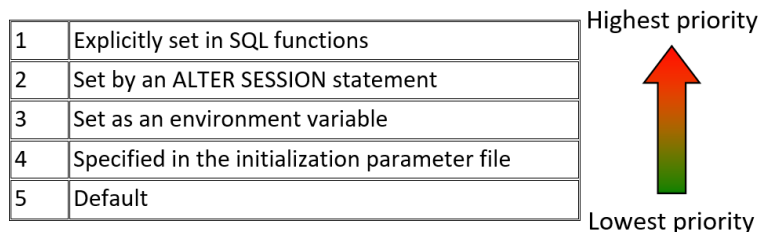


Fig. 13.1: Properties level priority

Session configuration and actual settings can be obtained using the following *Select* statement. Notice that the *nls_session_parameters* view consists of two attributes – *parameter* and *value*.

```
desc nls_session_parameters
```

```
Name      Null?  Type
-----
PARAMETER  VARCHA2 (30)
VALUE      VARCHA2 (64)
```


Offered language support parameters are following:

```
select * from nls_session_parameters;

NLS_LANGUAGE= 'AMERICAN';
NLS_TERRITORY= 'AMERICA';
NLS_CURRENCY= '$';
NLS_ISO_CURRENCY= 'AMERICA';
NLS_NUMERIC_CHARACTERS= '.,';
NLS_CALENDAR= 'GREGORIAN';
NLS_DATE_FORMAT= 'DD-MON-RR';
NLS_DATE_LANGUAGE= 'AMERICAN';
NLS_SORT= 'BINARY';
NLS_TIME_FORMAT= 'HH.MI.SSXF AM';
NLS_TIMESTAMP_FORMAT= 'DD-MON-RR HH.MI.SSXF AM';
NLS_TIME_TZ_FORMAT= 'HH.MI.SSXF AM TZR';
NLS_TIMESTAMP_TZ_FORMAT= 'DD-MON-RR HH.MI.SSXF AM TZR';
NLS_DUAL_CURRENCY= '$';
NLS_COMP= 'BINARY';
NLS_LENGTH_SEMANTICS= 'BYTE';
NLS_NCHAR_CONV_EXCP= 'FALSE';
```

The following sections will describe the main parameters for dealing with *Date* and *Timestamp* values. The complete specification can be found in the documentation (https://docs.oracle.com/cd/A84870_01/doc/server.816/a76966/ch2.htm#92653).

13.1.1 NLS_Language

NLS_Language specifies the default conventions for the following session characteristics:

- language for server messages,
- language for *day* and *month* names and their abbreviations (specified in the SQL functions *TO_CHAR* and *TO_DATE*),
- symbols for equivalents of *AM*, *PM*, *AD*, and *BC*,
- default sorting sequence for character data when *ORDER BY* is specified (*GROUP BY* uses a binary sort, unless *ORDER BY* is specified).

The value specified for *NLS_Language* in the initialization file is the default for all sessions in that instance.

The following example shows the text information reflecting the execution based on defined language. Database systems response is in selected language:

```
alter session set nls_language = 'Slovak';
```

```
relacia zmenena
```

```
drop table tab_non_existing;
```

```
ERROR v riadku 1:
ORA-00942: tabuľka alebo pohľad neexistuje
```

```
alter session set nls_language = 'English';
```

```
session altered.
```

```
drop table tab_non_existing;
```

```
ERROR at line 1:
ORA-00942: table or view does not exist
```

13.1.2 NLS_Territory

NLS_Territory specifies conventions for these date and numeric formatting characteristics:

- date format,
- decimal character and group separator,
- local currency symbol,
- ISO currency symbol,
- dual currency symbol,
- week start day,
- credit and debit symbol,
- ISO week flag,
- list separator.

Characteristics and limitations will be described later.

13.1.3 NLS_Date_Language

This parameter defines the language for the *Date* attribute value text format – spelling of the *day* and *month* names for the functions *To_char* and *To_date* by overriding *NLS_Language* parameter value. *NLS_Date_Language* has the same syntax as an *NLS_Language* parameter and can hold the value of any supported language.

Tab. 13.1: *NLS_Date_Language*

Parameter type:	String
Parameter scope:	Initialization parameter, Environment variable, and <i>ALTER SESSION</i>
Default value:	Derived from <i>NLS_Language</i>
Range of values:	Any valid language name

Let's have the following example. If the *NLS_Date_Language* parameter is changed, different values of the day and month in text format will be obtained.

```
alter session set nls_date_language = 'Slovak';

select to_char(sysdate, 'DD') as Day_num,
       to_char(sysdate, 'Day') as Day_text,
       to_char(sysdate, 'MM') as Month_num,
       to_char(sysdate, 'Month') as Month_text,
       to_char(sysdate, 'Mon') as Month_abr
from dual;
```

DAY_NUM	DAY_TEXT	MONTH_NUM	MONTH_TEXT	MONTH_ABR
03	Streda	05	Máj	Máj

```
alter session set nls_date_language = 'English';

select to_char(sysdate, 'DD') as Day_num,
       to_char(sysdate, 'Day') as Day_text,
       to_char(sysdate, 'MM') as Month_num,
       to_char(sysdate, 'Month') as Month_text,
       to_char(sysdate, 'Mon') as Month_abr
from dual;
```

DAY_NUM	DAY_TEXT	MONTH_NUM	MONTH_TEXT	MONTH_ABR
03	Wednesday	05	May	May

13.1.4 NLS_Date_format

NLS_Date_format defines the default format of the *Date* value used by calling implicit *To_char* and *To_date* functions. *NLS_Territory* determines the default value of this parameter. Any format mask can determine the definition. Moreover, if the constant string is added, such value must be enclosed with double-quotes.

Tab. 13.2: *NLS_Date_Format*

Parameter type:	String
Parameter scope:	Initialization parameter, Environment variable, and ALTER SESSION
Default value:	The default format for a particular territory
Range of values:	Any valid date format mask

```
alter session set nls_date_format='Current date:"
                                DD.MM.YYYY ", " HH24:MI ' ;

select sysdate as formatted_string from dual;
```

FORMATTED_STRING
Current date: 17.03.2017, 13:36

13.2 Transformation of the *personal_id* into the date of birth

Transformation of the *personal_id* attribute value to the date of birth (in *Date* data type) can be generally done in two ways. Both are based on *Select* statements. The first one is based on direct transformation in the *Select* statement. In this case, the input *personal_id* value must be separated into individual elements – *day*, *month*, and *year*. It cannot be done directly as the whole part, due to adding value 50 to the month definition for women. So, it is solved by executing a *substr* function for each element, managing the month definition, and consequently, by forming the string for *Date* value conversion using *concatenation* and *to_date* function. This approach is unhealthy due to no exception resistance. If any value cannot be transformed into a *Date* value due to an incorrect value (like month outside the range <1;12>), the whole operation is rolled back, resulting in providing no data result. Imagine that there can be thousands of people, and only one of them has incorrect *personal_id* value caused by typos.

Therefore, another solution should be introduced, isolating exceptions to the separate layer. Thanks to that, problems can be solved without raising exceptions externally. Therefore, for the next examples and definitions, reflect the following function code for *personal_id* transformation. If it is impossible to transform parameter value, an exception is raised, consequently returning the virtual date of birth. Thanks to that, incorrect data (typos of the *personal_id* values) can be evaluated and found easily.

For illustration purposes, a local variable is defined and loaded step by step using day, month, and year elements.

```

create or replace function PIDtoBirthDate(pid varchar2)
-- VARCHAR WITHOUT SIZE ELEMENT
return date
is
  v_str varchar2(10);
begin
  v_str := substr(pid, 5, 2) || '-';
  if (substr(pid, 3, 1) = 5 OR substr(pid, 3, 1) = 0) then
    v_str := v_str || '0' || substr(pid, 4, 1);
  else
    v_str := v_str || '1' || substr(pid, 4, 1);
  end if;

  -- MONTH, attention for female PID definition!
  v_str := v_str || '-19' || substr(pid, 1, 2);
  return to_date(v_str, 'dd-mm-yyyy');
  -- If the PID is not correct - cannot be transformed
  -- into date of birth, exception will be raised.
  -- In that case, virtual date of birth will be returned.
EXCEPTION WHEN OTHERS THEN
  return to_date('01-01-0001', 'dd-mm-yyyy');
end;
/

```

In a real environment, it would be done in one step. Even local variables can be omitted and directly transformed into *Date* data type values.

```

create or replace function PIDtoBirthDate(pid varchar2)
return date
is
  v_str varchar2(10);
begin
  v_str := substr(pid, 5, 2) || '-' || mod(substr(pid, 3, 2), 50) ||
    '-19' || substr(pid, 1, 2);
  return to_date(v_str, 'dd-mm-yyyy');
  -- If the PID is not correct - cannot be transformed into
  -- date of birth, exception will be raised.
  -- In that case, virtual date of birth will be returned.
EXCEPTION WHEN OTHERS THEN
  return to_date('01-01-0001', 'dd-mm-yyyy');
end;
/

```

13.3 Get the list of persons who celebrate a birthday today

Managing and comparing *Date* values is often a problem and source of errors. The aim is to get the list of the persons, who celebrate a birthday today. The previously defined function can be used.

We will list typical mistakes made by students or programmers, which leads to processing incorrect data if some data portions are returned.

Let's evaluate the following code. What about the results? Is there any problem?

```

select name, surname
from personal_data
where PIDtoBirthDate(personal_id) = sysdate;

```

Syntactically not, but no data will be returned. Can you explain the reason? Reflect on the following example. A simple table is created with only one attribute. The current value of the date is *inserted* and consequently *selected* based on the *sysdate* function. No data will be returned because of the time spectrum of the *Date* attribute.

Thus, the previous example managing date of birth would list only persons born today during midnight (00:00:00).

```
select name, surname, PIDtoBirthDate(personal_id)
from personal_data;
```

	NAME	SURNAME	PIDTOBIRTHDATE(PERSONAL_ID)
1	Michael	Pearce	06.11.1984 00:00:00
2	Jack	Smith	12.03.1984 00:00:00
3	John	Young	07.09.1986 00:00:00

To move to the next step, remove the time spectrum when comparing. What about the result? Are they correct?

```
select name, surname
from personal_data
where PIDtoBirthDate(personal_id) = trunc(sysdate);
```

No, at all. The result set consists of the people born today (during this day regardless of the time). The correct solution is based on comparing only day and month elements:

```
select name, surname
from personal_data
where to_char(PIDtoBirthDate(personal_id), 'DD.MM')
      = to_char(sysdate, 'DD.MM');
```

13.4 Get the list of students who passed the exam this month

This section points out the facts of the month evaluation with regards to the current date. Several solutions will be listed with false solutions to achieve the correct answer at the end. The aim is to list the students who passed the exam this month (similar to the last 30 days). Many times, only the month element itself is evaluated, which is, of course, incorrect.

```
select name, surname, student_id
from study_subjects join student using(student_id)
join personal_data using(personal_id)
where to_char(exam_date, 'MM') = to_char(sysdate, 'MM');
```

NAME	SURNAME	STUDENT_ID
Carol	Pearce	550545
Peter	Roger	550020
Jack	Robinson	501103
Tom	Moore	501201
Tom	Moore	501201
Peter	Murphy	500427
Milan	Clarke	500426
Milan	Clarke	500426

Let's see the whole *exam_date* attribute value with *Date* elements (day, month, and year).

```

select name, surname, student_id,
       to_char(exam_date, 'DD.MM.YYYY')
from study_subjects join student using(student_id)
                  join personal_data using(personal_id)
where to_char(exam_date, 'MM') = to_char(sysdate, 'MM');

```

NAME	SURNAME	STUDENT_ID	TO_CHAR(EXAM_DATE, 'DD.MM.YYYY')
Carol	Pearce	550545	12.06.2009
Peter	Roger	550020	20.06.2002
Jack	Robinson	501103	23.06.2003
Tom	Moore	501201	26.06.2001
Tom	Moore	501201	10.06.2003
Peter	Murphy	500427	06.06.2006
Milan	Clarke	500426	01.06.2006
Milan	Clarke	500426	05.06.2006

Thus, to get correct results, also year must be evaluated. Both following solutions are right.

```

select name, surname, student_id,
       to_char(exam_date, 'DD.MM.YYYY')
from study_subjects join student using(student_id)
                  join personal_data using(personal_id)
where to_char(exam_date, 'MMYYYY') = to_char(sysdate, 'MMYYYY');

```

```

select name, surname, student_id,
       to_char(exam_date, 'DD.MM.YYYY')
from study_subjects join student using(student_id)
                  join personal_data using(personal_id)
where to_char(exam_date, 'MM') = to_char(sysdate, 'MM')
and to_char(exam_date, 'YYYY') = to_char(sysdate, 'YYYY');

```

NAME	SURNAME	STUDENT_ID	EXAM_DATE
Mark	Bailey	501402	25.06.2017
Jack	Robinson	501103	25.06.2017
Jack	Robinson	501103	25.06.2017
John	Young	550127	25.06.2017

13.5 Get the list of students who passed the exam previous last month

However, think of getting the list of students who passed the exam last month. The solution described in chapter 13.4 Get the list of students who passed the exam this month does not provide sufficient power because individual date elements (month, year) are compared separately. Let's have the following solution. Is it correct? What about the provided limitations?

```

select name, surname, student_id,
       to_char(exam_date, 'DD.MM.YYYY')
from study_subjects join student using(student_id)
                  join personal_data using(personal_id)
where to_char(exam_date, 'MM') - 1 = to_char(sysdate, 'MM')
and to_char(exam_date, 'YYYY') = to_char(sysdate, 'YYYY');

```

Naturally, it cannot manage transitions over the years. So, a natural question arises, how to get data correctly? How to solve that definition? First of all, it is necessary to observe that it is impossible to evaluate month and year elements separately. Thus, to provide a suitable solution giving correct results, several opportunities are available to be covered by provided functions of the DBS.

The first two solutions are based on the *month_between* function, which checks the range of the value inside the range <1; 2>. In this case, last month is delimited by subtracting one month from a current date. The following figure shows the execution principle followed by the *Select* statement definitions. *We assume that the current date is 17.3.2017.*

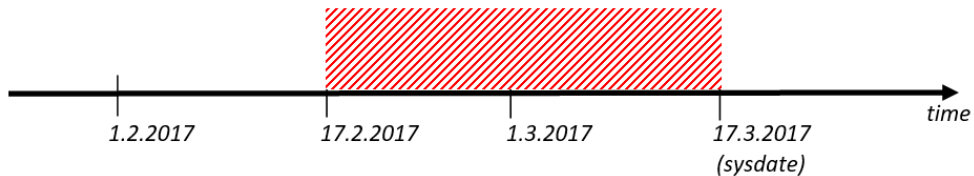


Fig. 13.2: Validity interval definition

```
select name, surname, student_id,
       to_char(exam_date, 'DD.MM.YYYY')
from study_subjects join student using(student_id)
                   join personal_data using(personal_id)
where months_between(sysdate, exam_date) between 0 and 1;
```

Another solution is based on a combination of multiple provided functions – *trunc* and *last_day* encapsulating the function *add_month*. In this case, last month definition is limited by the first and last date of the month.

The following figure shows the principle followed by the *Select* statement definition. Compare the principle with the previous solution. Different reflections and meanings of the term “last month” should be noticed. *We assume that the current date is 17.3.2017.*

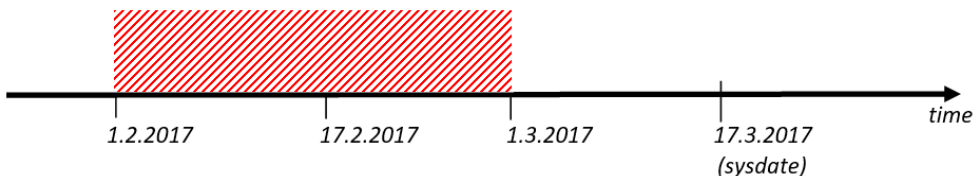


Fig. 13.3: Validity interval definition

```
select name, surname, student_id,
       to_char(exam_date, 'DD.MM.YYYY')
from study_subjects join student using(student_id)
                   join personal_data using(personal_id)
where exam_date BETWEEN TRUNC(ADD_MONTHS(sysdate, -1), 'MM')
      AND LAST_DAY(ADD_MONTHS(sysdate, -1));
```

If the whole time spectrum (also time) should be evaluated, it must be extracted from the current date and put together with used functions.

13.6 Get the list of the persons, who will celebrate their birthday next Sunday

Date value evaluating and shifting based on the day definition can be done using multiple ways. We will use the explicit definition by invoking the *to_char* method. In the following code of the function, we will highlight the limitation of usage based on server or session settings. It is often impossible to *alter* the system or even the session itself, whereas it can be connected to the rest part of the application assuming a specific *Date* or *Time* format. Moreover, it would end the transaction! Therefore, we will show the problems and propose solutions based on the following example. The aim is to get the *Date* value of the nearest *Sunday* based on the current date (*sysdate*). The first solution of the function definition can look like the following. Subsequently, a number of days is added to the current date by checking whether the result *Date* value is *Sunday* or not.

```
create or replace function GetNearestSunday return date
is
    v_day varchar2(10);
begin
    for i IN 1..7 loop
        select to_char(sysdate + i, 'DAY') into v_day from dual;
        if v_day like 'SUNDAY%' THEN
            return sysdate + i;
        end if;
    end loop;
    return null;
end;
/
```

As we can see, the solution is not very effective due to the cycle inside. Thus, the defined *Select* statement is performed at least once (which is naturally ok) but can even be executed 7 times, which is unnecessary. Therefore, there is also a better solution. Text format of the day is extracted and evaluated with the list of days in the *case* command. As a consequence, the *Select* statement is executed only once. The return value of the function is provided using the *Case* function.

```
create or replace function GetNearestSunday return date
is
    v_day varchar2(10);
begin
    select to_char(sysdate, 'DAY') into v_day from dual;
    case trim(v_day)
        when 'MONDAY'      then return sysdate + 6;
        when 'TUESDAY'     then return sysdate + 5;
        when 'WEDNESDAY'  then return sysdate + 4;
        when 'THURSDAY'   then return sysdate + 3;
        when 'FRIDAY'      then return sysdate + 2;
        when 'SATURDAY'    then return sysdate + 1;
        when 'SUNDAY'      then return sysdate + 7;
        else return null;
    end case;
end;
/
```

It is far more effective. However, is it also robust? Unfortunately, not. Let's evaluate the following conditions. The result of the proposed method is strictly delimited by the server

or session format definition, causing problems in scalability and deployability. It namely provides a correct solution only if the *English* branch of the language is used. If the language is changed, the defined function will return a *NULL* value.

```
select sysdate, TO_CHAR(sysdate, 'DAY') as day from dual;
```

<i>SYSDATE</i>	<i>DAY</i>
17.03.2017	FRIDAY

```
alter session set nls_date_language='English';
select GetNearestSunday from dual;
```

<i>GETNEARESTSUNDAY</i>
19.03.2017

```
alter session set nls_date_language='Slovak';
select GetNearestSunday from dual;
```

<i>GETNEARESTSUNDAY</i>
(null)

There is a significant problem, isn't it? However, once again, it is not suitable to change the format of the session's language. Thankfully, a solution exists without influencing *server* or *session* format by shifting the evaluation to the command level. The *to_char* function generally uses two parameters – *date_val* and *format_mask*. However, the syntax of the method *to_char* allows you to use also the third parameter – *nls_language*, by which the language settings can be influenced, but only for such method invocation. Thus, no other systems are affected.

```
select to_char( date_val [, format_mask] [, nls_language] ) from dual;
```

Setting *Date format* to *American* inside the *to_char* function invocation, a robust solution is provided regardless the current set language. Thanks to that, no problem with function result can occur. A complex solution of the *GetNearestSunday* function definition will look like following:

```
create or replace function GetNearestSunday return date
is
  v_day varchar2(10);
begin
  select to_char(sysdate, 'DAY', 'American') into v_day from dual;
  case trim(v_day)
    when 'MONDAY' then return sysdate + 6;
    when 'TUESDAY' then return sysdate + 5;
    when 'WEDNESDAY' then return sysdate + 4;
    when 'THURSDAY' then return sysdate + 3;
    when 'FRIDAY' then return sysdate + 2;
    when 'SATURDAY' then return sysdate + 1;
    when 'SUNDAY' then return sysdate + 7;
    else return null;
  end case;
end;
/
```

And provided results:

```
alter session set nls_date_language='English';
select GetNearestSunday from dual;
```

GETNEARESTSUNDAY
19.03.2017

```
alter session set nls_date_language='Slovak';
select GetNearestSunday from dual;
```

GETNEARESTSUNDAY
19.03.2017

A similar problem with *Date* format and location definition can occur when the order of days in the week should be provided. As we know, some countries consider the first day of the week as *Sunday*, and the rest reflect *Monday*. Therefore, the fundamental question is how to solve it. The *nls_territory* parameter definition namely influences such behavior. Let's notice the examples:

```
alter session set nls_territory='Slovakia';
select to_char(sysdate, 'D') from dual;
```

TO_CHAR(SYSDATE,'D')
5

```
alter session set nls_territory='America';
select to_char(sysdate, 'D') from dual;
```

TO_CHAR(SYSDATE,'D')
6

Remember that execution of the *to_char* method cannot be influenced by the *nls_territory* parameter. The only *nls_language* can be used, if necessary. Robust solution irrespective of the *nls_territory* parameter value is based on transforming the text value of the day into a desired numerical value regarding the *nls_language* parameter of the *to_char* function. Thus, the following *Select* statement will always reflect *Mondays* as the first day of the week.

```
select decode(trim(to_char(sysdate, 'DAY', 'nls_date_language=American')),
             'MONDAY',      1,
             'TUESDAY',     2,
             'WEDNESDAY',   3,
             'THURSDAY',    4,
             'FRIDAY',      5,
             'SATURDAY',    6,
             'SUNDAY',      7)
from dual;
```

Notice that parameter "DAY" of the function *to_char* provides an uppercase result, whereas using "Day" would offer a lowercase result (the first letter is uppercase).

13.7 Get the Date of the second Sunday of the month

Database maintenance operation planning should be selected precisely to be executed during the specific period (during low workload). Therefore, it is necessary to get desired time borders based on defined requirements. Let's assume that update operations and

statistics evaluation refreshing management should be done once a month, namely every second *Sunday* of the month. Whereas it should be planned automatically, it is necessary to evaluate the *Date* value dynamically. In principle, how to get the required *Date* value based on provided input date? We will describe the solution step by step:

1. Get the *Date* of the first day of the month and check for the week of the day.
2. If it reflects the *Sunday*, add 7 days and end the processing. If not, continue with step 3.
3. Find the first *Sunday* of the month.
4. Add 7 days.

The complete solution can look like following:

```
select case
  when to_char(first_day, 'D') = 1 then
    -- first day = Sunday
    to_char(first_day + 7, 'DD.MM.YYYY')
  else
    -- first day is not Sunday
    to_char(next_day(first_day, 1) + 7, 'DD.MM.YYYY')
  end as second_sunday
from
  (select trunc(sysdate, 'MM') as first_day from dual);
```

Notice that the solution is *nls_territory* dependent – try to create a robust solution based on it.

13.8 Get the list of the persons, who will celebrate their birthday next week

Listing the persons who will celebrate a birthday next week requires two parts to be handled.

Left limitation reflects the first day of the following week (*Monday*). The right limitation is the *Sunday* of the next week. Thus, these values can be obtained like this:

The first day of the next week – result of the function *next_day* is used.

```
select next_day(sysdate, 'MONDAY') from dual;
```

Last day of the next week – processing must be shifted to *Sunday* of the actual week or even any day (except *Sunday*) of the next week. Thus, the value of 6 days is added to the current date value. Then, the *next_day* function is used.

```
select next_day(sysdate + 6, 'SUNDAY') from dual;
```

Result:

```
select name, surname, to_char(PIDtoBirthDate(personal_id), 'DD.MM')
from personal_data
  where to_char(PIDtoBirthDate(personal_id), 'MMDD')
        between to_char(next_day(sysdate, 'Monday'), 'MMDD')
              and to_char(next_day(sysdate + 6, 'Sunday'), 'MMDD');
```

The main disadvantage of the previously defined solution is language dependency. If the *English* language branch is used, no problem can occur. However, other languages will not provide data for the *next_day* function result definition – the condition would not be

able to be evaluated. For the robust and immune solution, a special trick can be used. It is based on the knowledge that the *1.1.1900* was *Monday*. From that information, the text form of the day value can be provided concerning actual language settings. Then, invoking the *next_day* function will get desirable results, whereas format value is delimited by obtained day text format value from the *Date* – 1.1.1900. Consider the following example for getting the *Date* value of the next *Monday*.

```
declare
  v_monday_text varchar2(50) := to_char(to_date('19000101', 'yyyymmdd'),
                                         'Day');
  v_result date;
begin
  v_result := next_day(sysdate, v_monday_text);
  dbms_output.put_line(v_result);
end;
/
```

The database administrator cannot change the *nls_date_format* parameter value during the execution, whereas such parameter is static and would require a server restart. So, the complex and immune solution is provided. Naturally, for use in a real environment, it would be encapsulated by the function definition.

13.9 Get the difference between Date values

The result of the two *Date* values subtracting is the number of days between. Let's consider the result of the following example. The result is 9 days, decimal part (*0.08333*) reflects two hours (*2/24*).

```
select
  to_date('21.3.2017 12:00:00', 'DD.MM.YYYY HH24:MI:SS')
- to_date('12.3.2017 10:00:00', 'DD.MM.YYYY HH24:MI:SS')
from dual;
```

If you want to get the result in another form (like hours, minutes, ...), the result set value must be processed into the required format. Chapter [13.10 Get the difference between Date values – a sophisticated solution](#) describes the complex approach highlighting multiple granularity levels.

13.10 Get the difference between Date values – a sophisticated solution

Think of another solution, which will list the difference more sophisticatedly, reflecting a number of years, months, days and hours, minutes and seconds between two *Date* values. Such a function does not exist automatically. Therefore, we will show how to code it. In the following part, we will use two parameters of the function – *p_date1*, *p_date2* and assume that *p_date1* <= *p_date2* (it can be tested inside the function values can be interchanged, if necessary).

The optimal solution for getting years between two *Date* values provides function *months_between* divided to 12 (number of months delimiting the whole year). Whereas

the decimal part of the result is treated with lower granularity (months, days, ...), only truncated value for the year between definitions will be used:

```
year_count := trunc(months_between(v_date2, v_date1) / 12);
```

From the rest part, the month component is extracted. Original values are processed using the *month_between* function *subtracted* by the number of years (multiplied by value 12).

```
v_months_count :=
    trunc(months_between(v_date2, v_date1) - 12 * v_year_count;
```

To get a number of the days, the auxiliary variable will be used (*v_temp_date*) for the illustration purposes. It will store the value of the lower parameter value (*v_date1*) with the added *year* (multiplied by value 12) and *month* spectrum (evaluated by the *add_month* function result):

```
v_temp_date := add_months(v_date1, 12 * v_year_count + v_months_count);
```

The number of days between these two values can be obtained by subtracting the original value (*v_date2*) and defined auxiliary variable (*v_temp_date*). If the result set should also contain a time spectrum, the number of days is truncated.

```
v_day_count := trunc(v_date2 - v_temp_date);
```

Complete solution and example of the received results are following:

```
create or replace function Get_difference_date
(p_date1 date, p_date2 date)
-- must be p_date1 <= p_date2
return varchar2
is
    v_date1 date;
    v_date2 date;
    -- for changing, if the parameter order is not suitable
    -- v_temp double precision;
    v_temp_date date;
    v_temp integer;
    -- YYYY, MM, DD
    v_year_count integer;
    v_months_count integer;
    v_day_count number;
begin
    if p_date1 > p_date2 then
        v_date1 := p_date2;
        v_date2 := p_date1;
    else
        v_date1 := p_date1;
        v_date2 := p_date2;
    end if;

    -- get the number of months between two dates
    v_temp := months_between(v_date2, v_date1);
    -- year count is calculated as truncated value
    -- division of number of months between divided by 12
    v_year_count := trunc(v_temp / 12);
    -- the rest part expresses
    -- the number of months in the year
    v_temp := v_temp - 12 * v_year_count;
```

```

-- value is truncated,
-- the rest part expresses the number of days
v_months_count := trunc(v_temp);
-- to get number of days
-- (with regards on obtained number of years and months)
-- processed elements(year, months) are subtracted
-- from higher parameter value (v_date2)
v_temp_date := add_months(v_date1, 12 * v_year_count + v_months_count);
v_temp := v_date2 - v_temp_date;

-- if time elements are not processed,
-- particular day value is rounded, otherwise truncated.
v_day_count := round(v_temp, 2);
return v_year_count || ' years, ' || v_months_count || ' months, ' ||
       v_day_count || ' days.';
end;
/

```

```

select GET_DIFFERENCE_DATE(
        to_date('21.3.2017 15:10:22', 'DD.MM.YYYY HH24:MI:SS'),
        to_date('6.2.2013 11:00:11', 'DD.MM.YYYY HH24:MI:SS'))
       as difference
from dual;

```

```
4 years, 1 months, 15 days.
```

A similar approach can also be defined for the time spectrum itself.

13.11 YY vs. RR

In chapter [2.3.4 Conversion functions](#), the format of the *YYYY*, *RRRR*, and *YY*, *RR* has been introduced for getting the year element from the *Date* value. Transformation of the *string* value to *Date* using the *to_date* method is significant. If four value format of the year is used, results are the same regardless of using format *YYYY*, respectively *RRRR*. However, for *Insert* and *Update* statement execution, there is a significant difference if the century of the year value is omitted – only two values for the year representation are used. *YY* value always represents the current century, so the value is always larger (or equal) than the millennium 2000. *RR* value works differently. The provided result depends on the value of the year (two values). In principle, if the value is covered by the range <0 ; 49>, 21st century is used. In other cases (<50 ; 99>), reflection is made to the 20th century.

Let's create a simple table *T1* consisting of only one attribute (*val*) using *Date* type. Then, insert these two *Date* values using *YY* format. Afterward, get the full *Date* format (at least the whole year element). What about the provided data? Both of them reflect the current century (21st century). For each executed *Insert* statement, we assume that rollback is executed after evaluation by the *Select* statement.

```

create table TAB1(val date);
insert into TAB1 values(to_date('1-1-15', 'DD-MM-YY'));
select to_char(datum, 'DD-MM-YYYY') from TAB1;

```

```
01-01-2015
```

```
select to_char(datum, 'DD-MM-RRRR') from TAB1;
```

```
01-01-2015
```

```
insert into TAB1 values(to_date('1-1-60', 'DD-MM-YY'));
select to_char(datum, 'DD-MM-YYYY') from TAB1;
```

```
01-01-2060
```

```
select to_char(datum, 'DD-MM-RRRR') from TAB1;
```

```
01-01-2060
```

Repeat the same commands. However, now, use the *RR* format. As you can see from the year element part, the first one will reflect the current century (21st century), but the second solution refers to the 20th century. Be familiar with it.

```
insert into TAB1 values(to_date('1-1-15', 'DD-MM-RR'));
select to_char(datum, 'DD-MM-YYYY') from TAB1;
```

```
01-01-2015
```

```
select to_char(datum, 'DD-MM-RRRR') from TAB1;
```

```
01-01-2015
```

```
insert into TAB1 values(to_date('1-1-60', 'DD-MM-RR'));
select to_char(datum, 'DD-MM-YYYY') from TAB1;
```

```
01-01-1960
```

```
select to_char(datum, 'DD-MM-RRRR') from TAB1;
```

```
01-01-1960
```

Notice, *Select* statement is not influenced. The value stored in the database is essential.

13.12 Actual employees

An employment contract is a relation of the *person* (defined by the *personal_id* value in our case) and *employer* covered by the *employer_id* as the primary key. Moreover, it must also be delimited by the time range (*date from*, *date to*). Therefore, the *primary key* of the table *employee* is composite and consists of three attributes – *personal_id*, *employer_id*, and *date_from*.

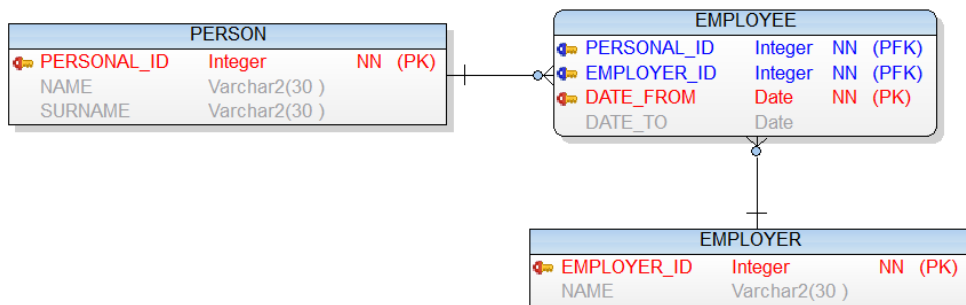


Fig. 13.4: An employment model

The aim is to get a *list of actual employees*. In principle, two situations can occur regarding time elements:

1. Fixed-term employment – attribute *date_to* > *sysdate*
2. Employment for an indefinite period – *date_to* IS NULL

Moreover, also *date_from* value should be evaluated and must hold the following expression value: *date_from* <= *sysdate*. The reason is that contracts, which will start in the future, can be inserted into the database sooner than the validity point starts. The solution can, therefore, look like the following:

```
select name, surname
  from personal_data JOIN employee using(personal_id)
 where date_from <= sysdate
       and (date_to >= sysdate OR date_to IS NULL);
```

The second condition group (*date_to* >= *sysdate* OR *date_to* IS NULL) evaluating expiration date (*date_to*) can be together, forming only one condition. In that case, an undefined value is replaced using the *NVL* function.

```
select name, surname
  from personal_data JOIN employee using(personal_id)
 where date_from <= sysdate
       and (NVL(date_to, sysdate) >= sysdate);
```

13.13 Period models and Allen relationships

The validity of the contract, respectively time interval modeled by two attributes (characterizing left and right border), can use four structural types. Definition and approach must be determined during the table creation, respectively, before the first *Insert* to that table.

In the past, there were several attempts for modeling time duration using *period* data type expressing begin and end point of the validity. Unfortunately, such an approach has not been approved as a standard resulting in the complete abolition of this concept in 2001.

Therefore, explicit modeling must be used highlighting these representations:

- Closed – closed,
- Closed – open,
- Open – open,
- Open – closed.

These representations determine whether the border *Date* (or *Timestamp* based on used granularity) belongs to the interval or not. In principle, only the closed type of the left border (begin point) is used in practice because of the necessity for strict limitation of the beginning point of the validity interval. Therefore, two approaches are used to represent the end point of the validity – either open or closed.

Undefined state management is easier to be distinguished by the *Closed – open* representation. Moreover, such a solution is robust, immune to the changing granularity (nowadays, the granularity of the processing is moving to fine precision grade more and more). Closed – open representation is modeled in the following table consisting of four attributes:

- ID – identifier of the object itself, *part of the primary key*,
- BD, ED – attributes characterizing validity, *part of the primary key*,
- Data – attribute values themselves are modeled by using a common naming “*data*” for illustration simplicity.

Tab. 13.3: Closed-open model

ID	BD	ED	Data
1	September 2012	July 2013	123
2	January 2013	December 2014	555
1	October 2013	January 2014	456

The undefined state is characterized by situations where *ED* does not meet consecutive *BD* in time.

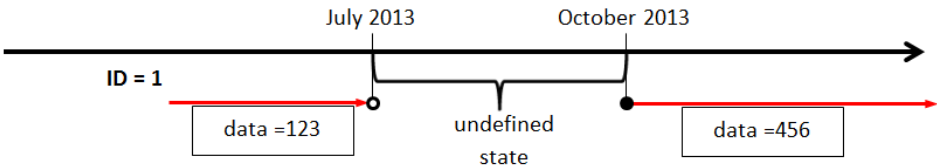


Fig. 13.5: Closed-open model

Individual period representations can be transformed by each other. The following table shows the mapping for such solution transformation.

Representation	Predicate
$[a_1, a_2] \text{ equals } [b_1, b_2]$	$a_1 = b_1 \text{ AND } a_2 = b_2$
$[a_1, a_2] \text{ equals } [b_1, b_2)$	$a_1 = b_1 \text{ AND } a_2 + 1 = b_2$
$[a_1, a_2] \text{ equals } (b_1, b_i]$	$a_1 = b_1 + 1 \text{ AND } a_2 + 1 = b_1 + b_i$
$[a_1, a_2] \text{ equals } [b_1, b_2]$	$a_1 = b_1 \text{ AND } a_2 = b_2 + 1$
$[a_1, a_2] \text{ equals } [b_1, b_2)$	$a_1 = b_1 \text{ AND } a_2 = b_2$
$[a_1, a_2] \text{ equals } (b_1, b_i]$	$a_1 = b_1 + 1 \text{ AND } a_2 = b_1 + b_i$
$(a_1, a_i] \text{ equals } [a_2, b_1]$	$a_1 + 1 = a_2 \text{ AND } a_1 + a_i = b_1 + 1$
$(a_1, a_i] \text{ equals } [a_2, b_1)$	$a_1 + 1 = a_2 \text{ AND } a_1 + a_i = b_1$
$(a_1, a_i] \text{ equals } (a_2, a_{i_2})$	$a_1 = a_2 \text{ AND } a_{i_1} = a_{i_2}$

Fig. 13.6: Time definition model transformation; source: Tom Johnston, Randall Weis: Managing Time in Relational Databases: How to Design, Update and Query Temporal Data

Allen relationships describe all possible positional relationships between two time periods along the common timeline. There are 13 *Allen relationships* in total. One of them does not have an inverse relationship.

Names of relationships are standardized, defined in 1983. They are part of the ordinary and query language, so when we refer to the technical language, we will write their names separated by parentheses.

In general, the two-time intervals on a common axis can be either separated (*exclude*) or may have at least one common point in time (*intersect*). Namely, it can be a relationship *fills* or *overlaps*. If one interval is in relation *fills* with another, any of its subintervals are correlated *fills* to the second interval. However, it is not necessarily true conversely. In the case of overlapping intervals *overlaps*, each has at least one common point in time and at least one that the second does not contain.

Relationship *exclude* indicates that the intervals do not have any common point. We can define two types. If there is at least one point in time between them, we use the relationship *before*. Otherwise, we define the relationship *meets* – one interval is immediately following the second.

If one interval fills (relationship [*fills*]) the second, two situations can occur. They are both identical – [*equals*]. Thus, there is no time point belonging to one, which does not belong to the second interval. The second situation occurs when the first interval is a subset of the second interval. In this case, we are speaking about the relationship [*occupies*]. Again, the opposite relationship does not apply. The general relationship [*occupies*] can be divided into the relationship [*aligns*] and [*during*]. The relation's name [*aligns*] defines its properties – two intervals have the common beginning or end time of the interval (exclusively – only one of them is true). In this case, we are talking about the relationship [*starts*], where intervals have a common beginning. Otherwise, we use the relationship [*finishes*]. However, if the intervals have a common beginning and end point of the interval, it represents the relationship [*equals*].

If the relationship is defined as [*occupies*] and the intervals do not have a common beginning and end time of the interval, the relationship [*during*] is defined – beginning of the first period occurs later than the start of the second, end of the first period occurs before the end of the second interval. The following figure shows the positional relationships based on the relationship [*fills*].

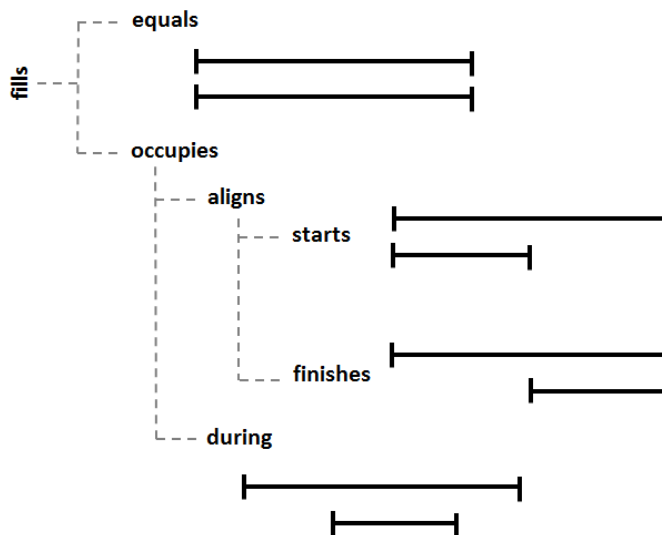


Fig. 13.7: Allen relationships (submodel)

A particular case is a time interval containing precisely one point in time. There are two possibilities in comparison with other time intervals (including at least two points in time). The first case – the time point is part of the second interval [*intersects*]. In this case, the relationship is defined as [*fills*] and [*occupies*]. If the time point is also an extreme point of the interval, it is the relationship [*starts*] or the relationship [*finishes*]. In other cases, the point is inside the interval and is defined by the relationship [*during*]. The second case compared to the time point, and the interval is when the point is not included in the second interval. In this case, the relationship between the intervals is called [*excludes*]. If there is no time between defined time intervals – it is a relationship [*meets*]. Otherwise, it is a relationship [*before*] – we assume that the time point precedes the second interval.

The latest case is the comparison of intervals, which contain just one element-time point. The first type is that the values are equal – relationship [*equals*]. If the values are not identical,

it is important whether there is a time point (or more) between them or not. If so, then it is a relationship [*meets*]. Otherwise, we talk about the relationship [*before*].

As we mentioned in the previous section, these relationships are essential elements for processing and comparison. The area of temporal databases focuses mainly on these three interval relations:

1. Relationship [*intersects*] is vital for the transactions that add new records to the database. Time interval defining the validity of the record must be disjoint with all already defined intervals of a given object. There cannot be valid two or more versions (episodes) at the same time. *Update* and *Delete* transactions again use this method to find the record, the validity of which contains a user-specified time.
2. Relationship [*before*] is used to distinguish and sort the individual episodes.
3. The basis for versions comparison and unification is the relationship [*meets*].

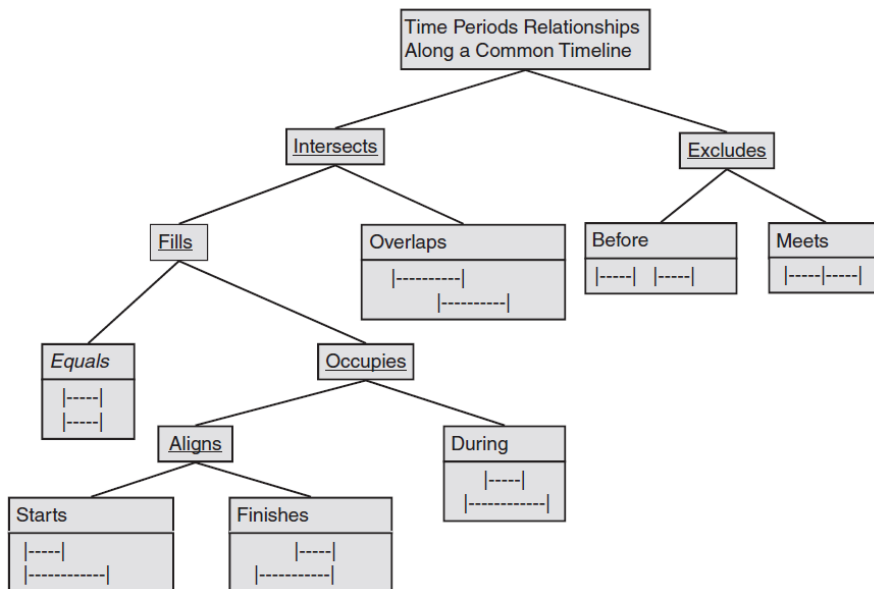


Fig. 13.8: Allen relationships!; source: Tom Johnston, Randall Weis: *Managing Time in Relational Databases: How to Design, Update and Query Temporal Data*

13.14 Unlimited validity definition

Undefined values are usually modeled using the *NULL* values. For the *Date* attribute management, *MaxValueTime* notation has been introduced, which is modeled by the latest date, that can be stored in current database systems. Naturally, it can be used only for the end date of the validity. The meaning is “later than now”. Although we do not know the exact time of the end border, it is evident that such a moment has not come yet. Undefined values modeled by the *NULL* characteristics are commonly used for unknown data, but in this case, some time position (future) can be defined, although not strict. It is one of the main reasons why not to use *NULL* values for the time definition.

Moreover, previously explained *Allen relationships* for time interval comparison would not be possible to be used at all due to no *NULL* value comparison opportunities. Last but not least aspect is just the performance. Database system consists of multiple performance enhancers – index structures. The main stream forming the index approaches of the current database systems is the *B+ tree* index type, which cannot, however, deal

with *NULL* values at all. The real representation of *MaxValueTime* notation used is **31.12.9999 (DD.MM.YYYY)**.

13.15 Data type Interval management

Consequently, since the data type *period* was not accepted as the norm, the need for modeling time in some other way became significant. As we already mentioned, one possibility is based on interval limitation by two values characterizing begin and end date of the validity with regards to representation (*CC*, *CO*, *OO*, *OC*). In principle, another data type category can be defined by characterizing the duration – data type ***Interval Year To Month*** and ***Interval Day To Second***. Be aware. They indicate only time duration, not the position in the time sphere (there is no information about the start, nor end position, only the duration itself). Thus, mapping the interval representation by two *Date* values (or *Timestamp* values based on granularity) can be shifted to only one *Date* value (or *Timestamp* value based on granularity) followed by another attribute defining duration – data type *Interval*.

13.15.1 Interval Year to Month data type

This data type can hold the value in the range of months and years. It uses the following syntax model:

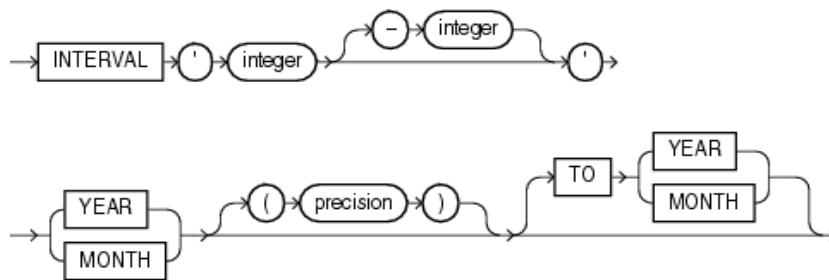


Fig. 13.9: Interval year to month

The value in the parentheses expresses the precision (maximal number of numeric places for year definition). The default value is 2. Then, the keyword is listed, characterizing the meaning of the proposed value.

Tab. 13.4: Interval

Form of Interval Literal	Interpretation
INTERVAL '123-2' YEAR(3) TO MONTH	An interval of 123 years, 2 months. You must specify the leading field precision if it is greater than the default of 2 digits.
INTERVAL '123' YEAR(3)	An interval of 123 years 0 months.
INTERVAL '300' MONTH(3)	An interval of 300 months.
INTERVAL '4' YEAR	Maps to INTERVAL '4-0' YEAR TO MONTH and indicates 4 years.
INTERVAL '50' MONTH	Maps to INTERVAL '4-2' YEAR TO MONTH and indicates 50 months or 4 years 2 months.
INTERVAL '123' YEAR	Returns an error because the default precision is 2, and '123' has 3 digits.

13.15.2 Interval Day to Second data type

This data type can hold the value in the range of days up to the finest granularity – seconds. It uses the following syntax model:

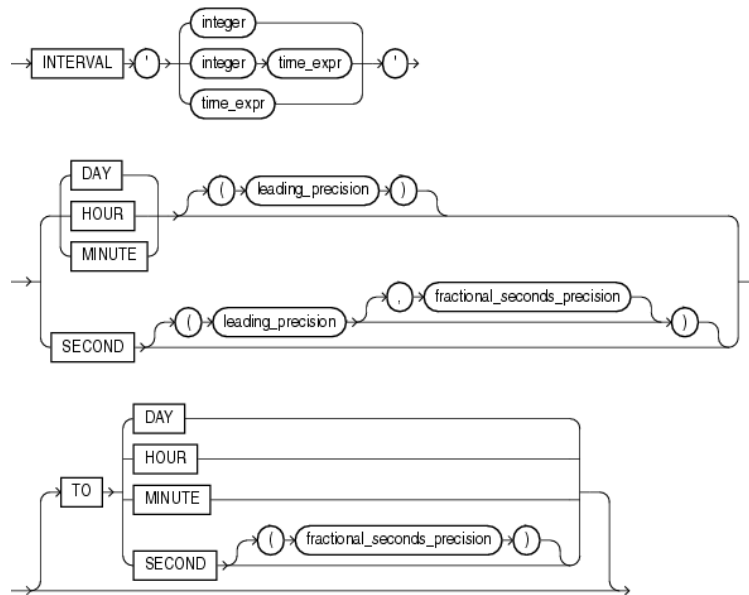


Fig. 13.10: Interval day to second

Principles are similar to the previously defined *Interval* type of the data structure type. In that case, the value in the parentheses also expresses the precision based on the specific element.

Tab. 13.5: Interval

Form of Interval Literal	Interpretation
INTERVAL '4 5:12:10.222' DAY TO SECOND(3)	4 days, 5 hours, 12 minutes, 10 seconds, and 222 thousandths of a second.
INTERVAL '4 5:12' DAY TO MINUTE	4 days, 5 hours, and 12 minutes.
INTERVAL '400 5' DAY(3) TO HOUR	400 days 5 hours.
INTERVAL '400' DAY(3)	400 days.
INTERVAL '11:12:10.2222222' HOUR TO SECOND(7)	11 hours, 12 minutes, and 10.2222222 seconds.
INTERVAL '11:20' HOUR TO MINUTE	11 hours and 20 minutes.
INTERVAL '10' HOUR	10 hours.
INTERVAL '10:22' MINUTE TO SECOND	10 minutes 22 seconds.
INTERVAL '10' MINUTE	10 minutes.
INTERVAL '4' DAY	4 days.
INTERVAL '25' HOUR	25 hours.
INTERVAL '40' MINUTE	40 minutes.
INTERVAL '120' HOUR(3)	120 hours.
INTERVAL '30.12345' SECOND(2, 4)	30.1235 seconds. The fractional second '12345' is rounded to '1235' because the precision is 4.

13.15.3 Examples – Interval data types

Let's have the following example of the *INTERVAL YEAR TO MONTH* mapping. Expression '14' month reflects one year and 2 months.

```
select interval '14' month from dual;
```

If the current date was 17.3.2017, the output would be 17.5.2018 (also with time spectrum). So notice the automatic mapping possibilities of the *Interval* to a *Date* value.

```
select sysdate + interval '1-2' year(3) to month from dual;
```

A similar situation occurs if *INTERVAL DAY TO SECOND* data type value is used. In that case, also fractions can be processed. Therefore, the mapping example will be associated with the *Timestamp* value in our case (it can also be associated with *Date* values based on automatic conversion methods. In that case, however, the fraction part will be removed).

```
INTERVAL '4 5:12:10.222' DAY TO SECOND(3)
-- 4 days, 5 hours, 12 minutes, 10 seconds and 222 thousands of seconds.
```

```
select systimestamp + INTERVAL '4 5:12:10.222' DAY TO SECOND(3)
from dual;
```

SYSTIMESTAMP	SYSTIMESTAMP+INTERVAL'45:12:10.222'DAYTOSECOND(3)
17.03.17 10:19:13, 945000000 +01:00	21.03.17 15:31:24, 167000000 +01:00

13.15.4 Update validity definition based on Interval data value

Validity modeled using two attributes defining begin and end point can be transformed to *Interval* definition and vice versa. In this section, we will describe the principles and show one example. The aim is to update *date_to* attribute value for employees of "ZU". Limit the employment contract to 3 years for all actual employees. Add the condition that if the value of the *date_to* would be in the past, the reflected output value should be set to the current date (*sysdate*).

To get the correct results, several steps must be done by creating conditions and set the correct value of the *date_to* attribute. Thus, first of all, define the condition characterizing the *employees*, whose *employer* name is *ZU*. It can be done using an inner *Select* statement from the *employer* table:

```
employer_id IN (select employer_id
                 from employer
                 where name = 'ZU')
```

Then, *date_to* value to be processed must also be limited only to the actual *employees* (see chapter 13.12 Actual employees):

```
date_to IS NULL or date_to > sysdate
```

So now, the conditions are defined. However, how to set the correct value of the *date_to* attribute? The employment contract should last 3 years. Comparison with the current date

must be highlighted. Therefore, if the value to be set is lower than the *sysdate* value, it should be replaced by the current date. Conditional processing can be done using *Case*:

```
date_to = case when (date_from + interval '36' month) < sysdate
              then sysdate else (date_from + interval '36' month) end
```

The complete solution can, therefore, look like following:

```
update employee
  set date_to = case when (date_from + interval '36' month) < sysdate
                    then sysdate else (date_from + interval '36' month) end
  where (date_to IS NULL or (date_to > sysdate and
                           date_to > date_from + interval '36' month)
        ) and employer_id IN (select employer_id
                              from employer
                              where name = 'ZU');
```


Lab 14 – Data dictionary views

When dealing with the database systems, it is not only about the data, but also overall architecture, structures, and privileges must be highlighted. All such metadata describing the objects are stored in the system tables and are available through the data dictionary views in a user-friendly format. In this lab, the reader will learn the categorization of the system tables focusing on object owners and accessibility. Querying data dictionary section highlights the most significant structures – list of tables, their structure (attributes with data types), primary key definition and foreign key reference, associated table triggers, method headers, and sequences.

By studying this lab, the reader will get a complex overview on data dictionary views by understanding the formats and available data. Thanks to that, any additional structural information can be obtained easily.

14.1 Introduction

Database system resources are continuously monitored, and individual changes are stored in the specific data structures. In Central Europe, these data structures are called “System tables”, and the actual state is obtained by querying “System tables”, however, to be honest, representation refers to *views*, not the *tables*. Never mind, such structures can be divided into two groups based on their characteristics and way of saving and updating. The first group is formed by *Dynamic Performance Views* (also referred to as “*Vee dollar*”). In general, there are more than 300 *dynamic performance views*, and they characterize the status of the instance and the database covering actual settings and approaches. They also cover some information, which can be found in the *data dictionary*, as well (described later). These *views* are created at the startup, updated according to settings, and dropped at the shutdown. *Some essential views for parameter settings and tuning at the instance level are V\$INSTANCE (status of the instance), V\$SGA (summary information about the shared memory structures and size), V\$SYSSTAT (instance statistics).* The physical database is covered by e.g., *V\$DATABASE* or *V\$DATAFILE*. The interim layer between instance and database is just *tablespace*, which can be queried using the *V\$TABLESPACE* view.

The session is also characterized by multiple *dynamic performance views*. As a representative, *V\$SESSION* can be mentioned listing current session information. *All views consist of multiple attributes, which can be listed using the already described DESC command. Further information can be found in Oracle documentation, or feel free to ask the teacher.*

This lab focuses on the second group defined by *Data Dictionary Views (DDV)*, which refers to the *metadata – data about data*. They describe the database and its contents. User definitions, security information, integrity constraints, and performance monitoring information (from release 10g onward) are part of the data dictionary views. They characterize all data objects defined in the database. As the naming notation implies, they deal with *views* defined on internal data structures (*Internal Tables; Base Tables*). Such *Internal Tables* are generated during the database creation process, management and loading are provided automatically.

14.2 Data dictionary – structure

Data Dictionary (abbreviation of the *Data Dictionary Views*) is one of the essential components of the database. It is *read-only* and provides information about the structure, objects, type, and much other helpful information for database management. In general, it provides the following data:

- **all schema object definitions** (tables, views, indexes, clusters, synonyms, sequences, procedures, functions, packages, triggers, etc.),
- allocated and free space for the particular object,
- integrity constraint definition information,
- users and their settings and properties,
- granted privileges and roles,
- audit information,
- other general database information.

As mentioned, it references **Internal Tables**, which contain all information. However, they are often encrypted and stored in specific data structures due to normalization, and it is complicated to get required data from them. Thus, the *data dictionary* can be considered as the middle layer between physical representation and users and provide desired data in user-friendly form. However, also **Internal Tables** can be queried directly.

The user **SYS** user owns **Internal Tables**.

Be aware, never try to change any data using a data dictionary or internal table (although it is possible, it is always a license breach). Only Oracle can do it automatically. Moreover, such activity can compromise data integrity and completely destroy the database.

Nowadays, granted privileges **Insert Any Table**, **Update Any Table** and **Delete Any Table** allows the user to manage “any” table in the system. In the past, these privileges also covered **Internal Tables**, so granted user would be able to irretrievably damage the structure, which was a significant security risk. Fortunately, current DBS versions do not allow it. Thus, nowadays, these privileges do not cover and allow destructive operations in **Internal Tables**.

When any query is provided, the *data dictionary* is also used for database systems activities, such as finding information about the user, schema objects, and physical data storage. Another example provides execution of any **DDL** command – all information about the performed activity is visible and accessible via these structures. For a fast approach, a significant data amount from the *data dictionary* is cached in the memory.

The *data dictionary* can be divided into three groups depending on what data are accessible or to whom they are available. They can be easily distinguished thanks to the same category of the prefix:

- **User** – view associated with the particular user, it contains information about objects owned by the particular user.
- **All** – these views extend the user views category by objects accessible to the particular user (which have been *granted* to the particular user).
- **Dbu** – these views consist of information about all user schemas (so this view is extended by the **Owner** attribute). Moreover, some **DBA** views have additional columns containing valuable information to the administrator.

Let's have the following example.

User **KVET** creates new table **TAB1** (assuming that he does not have any table in his schema for better illustration):

```
-- KVET
create table TAB1(id integer);
```

To get a list of all tables owned by user **KVET**, a view with the prefix **USER** can be used. It contains a wide range of attributes. However, for illustration purposes, we will use only some of them:

```
SQL> desc user_tables
```

Name	Null?	Type
TABLE_NAME	NOT NULL	VARCHAR2 (30)
TABLESPACE_NAME		VARCHAR2 (30)
CLUSTER_NAME		VARCHAR2 (30)
IOT_NAME		VARCHAR2 (30)
STATUS		VARCHAR2 (8)
....		

All attributes can be obtained by executing the description (DESC) command.

Tables owned by the currently logged user can be found in the **user_tables** data dictionary view.

```
select table_name from user_tables;
```

TABLE_NAME
TAB1

List of tables, which the particular user does not own, but they have been granted access to him, is accessible via **all_tables** data dictionary view. Let's grant **Select** privilege of the table **TAB1** owned by **Kvet** to **Kmat**. The owner of the table is stored in the **owner** attribute of the data dictionary view.

```
-- KVET
grant select on TAB1 to KMAT;

-- KMAT
select table_name from all_tables
where owner like 'KVET';
```

TABLE_NAME
TAB1

Such information is not accessible using the **User** data dictionary view type, whereas **Kmat** is not the owner of the table:

```
select table_name from user_tables
where table_name like 'TAB1';
```

```
no rows selected
```

When using a data dictionary, always remember that all values are UPPERCASE because of querying system (internal, base) tables!

See the following example executed by user *Kvet* highlighting the mentioned problem:

```
select table_name
  from user_tables
     where table_name like 'TAB1';
```

```
TABLE_NAME
-----
TAB1
```

```
select table_name
  from user_tables
     where table_name like 'tab1';
```

```
no rows selected
```

```
select table_name
  from user_tables
     where lower(table_name) like 'tab1';
```

```
TABLE_NAME
-----
TAB1
```

As mentioned, *data dictionary views* contain multiple views, which are formed by various attributes. Their names can be obtained using the description (*DESC*) function. The important factor is just the meaning and values, which can hold. These characteristics can be found either in documentation (*docs.oracle.com*), but also *data dictionary* can provide them. For these purposes, use the *dict_columns* data dictionary view. Realize, it is not prefixed by the user / all / dba, because it describes the meaning and comments of the attributes of data dictionary views. Table, which will be described, is delimited in the *Where* clause defining *table_name* (it reflects the name of the *data dictionary view*, although the name refers to the table).

```
select column_name, comments
  from dict_columns
     where table_name = 'USER_TABLES';
```

	COLUMN_NAME	COMMENTS
1	TABLE_NAME	Name of the table
2	TABLESPACE_NAME	Name of the tablespace containing the table
3	CLUSTER_NAME	Name of the cluster, if any, to which the table belongs
4	IOT_NAME	Name of the index-only table, if any, to which the overflow or mapping table entry belongs
5	STATUS	Status of the table will be UNUSABLE if a previous DROP TABLE operation failed, VALID otherwise
		...

If you want to get all *views* (including also *dynamic performance views*), which are part of the data dictionary, you can query *Dictionary* view, which consists of two attributes – *table_name* and *comments*.

```
desc dictionary
```

```
Name          Null?          Type
-----
TABLE_NAME          VARCHAR2 (30)
COMMENTS            VARCHAR2 (4000)
```

There are almost 2 000 data dictionary views and 600 dynamic performance views.

14.3 Querying data dictionary

This chapter will describe the essential *data dictionary views* and queries to obtain desired data about data modeling.

14.3.1 List of tables owned actual user

Either *user_tables* or *all_tables* delimited by the actual user (*where owner = user*) can be used. In general, *DBA* can also use the *dba_tables* view.

```
select table_name
  from user_tables;
```

```
select table_name
  from all_tables
  where owner = user;
```

14.3.2 List of table attributes

To get attributes of the table, the *Desc* command can be used. However, another approach is to query the *User_Tab_Cols* data dictionary view:

```
select column_name
  from user_tab_cols
  where table_name = 'STUDY_SUBJECTS';
```

	COLUMN_NAME
1	SCHOOL_YEAR
2	STUDENT_ID
3	SUBJECT_ID
4	LECTURER
5	RESULT
6	EXAM_DATE
7	SIGN_DATE
8	ECTS

14.3.3 Get attribute data type and characteristics

To get the attribute's data type, the previous query based on *User_Tab_Cols* can be extended by the *data_type* attribute. Such a result, however, does not contain the size of the attribute, if applicable (e.g., only information about data type *Varchar2* is provided with no size element (*Varchar2(30)*)).

```
select column_name, data_type
  from user_tab_cols
  where table_name = 'STUDY_SUBJECTS';
```

	COLUMN_NAME	DATA_TYPE
1	SCHOOL_YEAR	NUMBER
2	STUDENT_ID	NUMBER
3	SUBJECT_ID	VARCHAR2
4	LECTURER	CHAR

	<i>COLUMN_NAME</i>	<i>DATA_TYPE</i>
5	RESULT	VARCHAR2
6	EXAM_DATE	DATE
7	SIGN_DATE	DATE
8	ECTS	NUMBER

To get information about the precision and length of the particular attribute, use the following attributes of the *User_Tab_Cols* data dictionary:

- **Data_Length** – the length of the column (in bytes).
- **Data_Precision** – decimal precision for *Number* data type; binary precision for *Float* data type, *NULL* for all other data types.
- **Char_Length** – displays the length of the column in characters. This value can be relevant to *Char* and *Varchar* data types.

```
select column_name, data_type, data_length, data_precision,
       decode(nvl(char_length, 0), 0, ' ', char_length) as ch_length,
       nullable
from user_tab_cols
where table_name = 'STUDY_SUBJECTS';
```

	<i>COLUMN_NAME</i>	<i>DATA_TYPE</i>	<i>DATA_LENGTH</i>	<i>DATA_PRECISION</i>	<i>CH_LENGTH</i>	<i>NULLABLE</i>
1	SCHOOL_YEAR	NUMBER	22	4	(null)	No
2	STUDENT_ID	NUMBER	22	6	(null)	No
3	SUBJECT_ID	VARCHAR2	30	(null)	30	No
4	LECTURER	CHAR	5	(null)	5	No
5	RESULT	VARCHAR2	1	(null)	1	Yes
6	EXAM_DATE	DATE	7	(null)	(null)	Yes
7	SIGN_DATE	DATE	7	(null)	(null)	Yes
8	ECTS	NUMBER	22	2	(null)	Yes

Value of the attribute *Nullable* specifies whether a column can hold *NULL* values or not. Value is *N* if there is a *NOT NULL* constraint on the column. Remember that it is also applicable for the *primary key*, which cannot also hold *NULL* values. Otherwise, the value is *Y*.

To get data type information similarly as provided using *Desc* functionality, the query will be a bit more complicated. One character string for the *Nullable* sign is replaced by the text format. However, when dealing with the data type, two attributes must be evaluated to get correct results (*char_length*, *data_precision*). First of all, *char_length* is evaluated. Such values are, however, applicable only for string data types. Otherwise, value “0” is obtained – in that case, the second attribute – *data_precision* is evaluated, which can reflect the real value or *NULL* (for *Date* attributes). If a *NULL* value is provided, it is replaced by an empty string. In all other cases, the numerical output value is transformed

into a string (*to_char* method) and surrounded by parentheses. The query to get the solution can look like this:

```
select column_name,
       decode(nullable, 'Y', ' ', 'N', 'NOT NULL') as "NULL",
       data_type || decode(char_length, 0,
                           decode(to_char(data_precision), null, ' ',
                                   '(' || to_char(data_precision) || ')'),
                           '(' || to_char(char_length) || ')') as "Type"
from user_tab_cols
where table_name = 'STUDY_SUBJECTS';
```

The result will look like this:

	COLUMN_VALUE	NULL	TYPE
1	SCHOOL_YEAR	NOT NULL	NUMBER(4)
2	STUDENT_ID	NOT NULL	NUMBER(6)
3	SUBJECT_ID	NOT NULL	VARCHAR2(30)
4	LECTURER	NOT NULL	CHAR(5)
5	RESULT		VARCHAR2(1)
6	EXAM_DATE		DATE
7	SIGN_DATE		DATE
8	ECTS		NUMBER(2)

It provides the same results as the *Desc* function:

Name	NULL	TYPE
-----	-----	-----
SCHOOL_YEAR	NOT NULL	NUMBER (4)
STUDENT_ID	NOT NULL	NUMBER (6)
SUBJECT_ID	NOT NULL	VARCHAR2 (30)
LECTURER	NOT NULL	CHAR (5)
RESULT		VARCHAR2 (1)
EXAM_DATE		DATE
SIGN_DATE		DATE
ECTS		NUMBER (2)

14.3.4 Get system identifier and definition of the primary key

To get the system identifier of the primary key, use the following query based on the *User_Constraints* data dictionary view. *Constraint_Type* value “P” refers to the primary key of the table. An example is based on the *Study_Subjects* table.

```
select constraint_name
from user_constraints
where table_name = 'STUDY_SUBJECTS'
and constraint_type = 'P';
```

Two *data dictionary views* must be joined if you want to get attributes consisting of a primary key. *User_Constraints* contains *constraint_name* and references to the particular *table_name*. The attribute list itself is reached from *User_Cons_Columns*. However, such a *data dictionary view* does not include table name information. In this case, also the order (attribute *position*) of the attributes is significant because it influences the associated index.

```
select ucc.column_name, ucc.position
from user_cons_columns ucc
join user_constraints uc using(constraint_name)
where ucc.table_name      = uc.table_name
and uc.constraint_type    = 'P'
and uc.table_name        = UPPER('study_subjects');
```

COLUMN_NAME	POSITION
SCHOOL_YEAR	1
STUDENT_ID	2
SUBJECT_ID	3

So, the primary key of the *Study_Subjects* table is composite covering *school_year*, *student_id*, and *subject_id* (in this order).

Think about the consequences of changing the order of the attributes in the primary key definition.

14.3.5 Get system identifier and definition of the foreign key

In the *Study_subjects* table, we also have 3 references, three foreign keys – pointers to tables *teacher*, *student*, and *subject*. To get the information about foreign keys, use the previous query. For now, *constraint_type* attribute should contain value “R” (reference):

```
select constraint_name
from user_constraints
where table_name = 'STUDY_SUBJECTS' and constraint_type = 'R';
```

This is the structure of the output result set. For you, it should contain 3 rows. However, the values themselves will differ, whereas they are system generated based on actual server conditions.

CONSTRAINT_NAME
SYS_C007231
SYS_C007232
SYS_C007233

However, by using the previous query, you do not know the original table, which is referenced. To get references, use the following query. We will now use *data dictionary views* prefixed by the *All* and join with the *On* clause for demonstration purposes. We will use two *data dictionary views*, each of them will be used twice (thus, they must be aliased). The constraint information and table determination are stored in *All_constraints*, whereas the attributes defining the relationship are in *All_Cons_Columns*. *Join* operation between *All_Cons_Columns* and *All_constraints* is provided by composition – *owner* and *constraint_name* (if prefix *User* would be used, *Join* operation would reflect only *constraint_name* attribute). *Where* clauses conditions ensure that we deal with references (*c_fk.constraint_type = 'R'*) and the processed table is *Study_Subjects* (*a_c_fk.table_name = 'STUDY_SUBJECTS'*). The result set contains these attributes (in the left-right order):

1. Column name of the attribute in the table with a primary key.
2. Column name of the attribute in the table with a foreign key (*Study_Subjects*).
3. Name of the primary key constraint.
4. Name of the foreign key constraint.

5. Owner of the referenced table (referenced table is referenced by foreign key).
6. Name of the reference table.

```
select a_c_pk.column_name column_name_pk,
       a_c_fk.column_name column_name_fk,
       a_c_fk.constraint_name constraint_name_pk,
       c_pk.constraint_name constraint_name_fk,
       c_fk.r_owner owner_pk,
       c_pk.table_name table_name_pk
from all_cons_columns a_c_fk
     JOIN all_constraints c_fk ON a_c_fk.owner = c_fk.owner
       AND a_c_fk.constraint_name = c_fk.constraint_name
     JOIN all_constraints c_pk ON c_fk.r_owner = c_pk.owner
       AND c_fk.r_constraint_name = c_pk.constraint_name
     JOIN all_cons_columns a_c_pk ON a_c_pk.owner = c_fk.owner
       AND a_c_pk.constraint_name = c_fk.r_constraint_name
WHERE c_fk.constraint_type = 'R'
     AND a_c_fk.table_name = 'STUDY_SUBJECTS';
```

COLUMN_NAME_PK	COLUMN_NAME_FK	CONSTRAINT_NAME_PK	CONSTRAINT_NAME_FK	OWNER_PK	TABLE_NAME_PK
TEACHER_ID	LECTURER	SYS_C007231	SYS_C007212	STUDENT_ENG	TEACHER
SUBJECT_ID	SUBJECT_ID	SYS_C007232	SYS_C007209	STUDENT_ENG	SUBJECT
STUDENT_ID	STUDENT_ID	SYS_C007233	SYS_C007196	STUDENT_ENG	STUDENT

Let's separate the previously defined statement into two parts – primary key and foreign key management.

In the following example, a list of system identifiers of the foreign key is obtained. **Constraint_name** attribute expresses foreign key, **r_constraint_name** refers to the primary key when joining. Example deals with *Study_subjects* table. The highlighted name is the system identifier of the primary key in the *Student* table.

```
select uc.r_constraint_name
from user_constraints uc
     join user_cons_columns ucc
       on (uc.r_constraint_name = ucc.constraint_name)
where uc.constraint_type = 'R'
     and uc.table_name = 'STUDY_SUBJECTS'
order by uc.table_name, uc.r_constraint_name,
         ucc.table_name, ucc.column_name;
```

R_CONSTRAINT_NAME
SYS_C007196
SYS_C007209
SYS_C007212

To check it, list the system identifier of the primary key in *Student* table:

```
select constraint_name
from user_constraints
where table_name = 'STUDENT' and constraint_type = 'P';
```

CONSTRAINT_NAME
SYS_C007196

14.3.6 Listing triggers for a particular table

Before getting the list of the triggers associated with the particular table, create a new one for dealing with the new value of the attribute *student_id* of the *student* table. Then, create a new one controlling the correctness of the *first_date* attribute value – it must express the actual date of the new row insert and cannot be changed later (review of the [Lab 10 – Triggers](#)). Subsequently, list the names of the developed triggers? How many triggers are created? What about their structure and characteristics?

The correct answer is 2 or 3 depending on developing methods, one is *Update* trigger, and 2 triggers can be created for *Insert* (which can also be grouped, whereas the condition of firing is the same).

Significant attributes of the data dictionary are the following:

- *Trigger_name*.
- *Trigger_type* – defines the time of firing – *BEFORE* / *AFTER* and *STATEMENT* / *ROW*.
- *Triggering_event* – statement that will fire the trigger – *INSERT*, *UPDATE* and/or *DELETE*.
- *Table_name* – the name of the table that defined trigger is associated with.
- *Column_name* – the name of the column on which the trigger is defined.

A special attribute influencing firing is just the *status*. If the trigger is *Disabled*, the particular trigger is not fired at all.

```
select trigger_name, trigger_type, triggering_event,
       table_name, column_name, status
from user_triggers
where table_name = 'STUDENT';
```

14.3.7 Listing developed methods (procedures, functions)

Developed methods are grouped, forming *User_objects data dictionary views*. Method type is delimited by the *Object_type* attribute values – *Procedure*, *Function*. Any developed object information can be obtained, like *Sequence*, *Table*, *Index*, *View*, *Package*, *Package Body*, etc., using such a view. However, for simplicity, we will highlight only methods. The query for listing objects owned by the particular user is following:

```
select object_name, object_type, created
from user_objects
where object_type in ('FUNCTION', 'PROCEDURE');
```

OBJECT_NAME	OBJECT_TYPE	CREATED
DROP_JOB_PROC	PROCEDURE	10.09.2013
DYN_CUR	PROCEDURE	25.03.2013
DYN_CUR_PR	PROCEDURE	25.03.2013
FUNCONTAINS	FUNCTION	28.03.2013
FUNC_DV	FUNCTION	22.10.2015

For clarity, it is helpful to describe also the differences between three-time attributes of the *User_objects data dictionary views*. Attribute *Created* reflects the timestamp of the object creation, whereas attribute *Last_DDL_time* expresses timestamp for the last *DDL* or *DCL* change. The last attribute name is *Timestamp* and delimits the timestamp of the object specification.

Each object is internally represented by the **Object_ID**, part of the **User_objects** data dictionary view.

Let's create a simple function, which will return the gender of the person using the **personal_id** attribute. The solution can look like this (try to develop it alone).

```
create or replace function Func_gender(p_id char)
return char
is
begin
  case
    when substr(p_id, 3, 1) in (5, 6) then return 'female';
    when substr(p_id, 3, 1) in (0, 1) then return 'male';
    else return 'unknown';
  end case;
end func_gender;
/
```

Notice that the developed method should cover all cases. Therefore, do not forget to deal with incorrect data, which can be present in the table. If no Else clause is added, the following exception would be raised. Moreover, do not forget that string parameters do not contain the size definition in the method header.

```
Error at line 1:
ORA-06592: CASE not found while executing CASE statement
ORA-06512: at "KVET.FUNC_GENDER", at line 5
```

Parameters and properties of the developed method can be obtained using multiple ways. If you want to get the structure – parameters and return data type of the function – command **Description**, respectively **Desc** can be used resulting in the following output:

```
desc func_gender
```

Argument Name	Type	In/Out	Default
<return value>	CHAR	OUT	unknown
P_ID	CHAR	IN	unknown
P_NAME	CHAR	OUT	unknown
P_SURNAME	CHAR	OUT	unknown

How does it work? Where are the parameters stored? Naturally, in the **data dictionary** – **User_Arguments** view.

There are several attributes. The most important are:

- **Object_name**.
- **Argument_name** – the name of the parameter.
- **Position** – the order of the parameters in the definition.
- **Data_type** – information about the data type characteristic of the parameter.

```
select object_name, argument_name, position,
       data_type, data_length
from user_arguments
where object_name = 'FUNC_GENDER';
```

OBJECT_NAME	ARGUMENT_NAME	POSITION	DATA_TYPE	DATA_LENGTH
FUNC_GENDER	(null)	0	CHAR	(null)
FUNC_GENDER	P_ID	1	CHAR	(null)

When dealing with functions, the first position (value of the *position* attribute is *1*) reflects the first parameter of the function defined by its name and data type, etc. The return data type is expressed using *position = 0*, which does not have the argument name (*argument_name* is *NULL*).

For a clear explanation, create a function that is more complex by changing the previously defined function. Extend it by the name and surname of the person as output parameters. *In this case, the Select statement must be used inside the function. Therefore, do not forget to check whether the defined parameter value exists if the Select-Into statement is used.* The solution can look like the following (two solutions are mentioned). The gender of the person is obtained from the *personal_id* value using the *decode* and *substr* function. If the value cannot be obtained, the particular *Select* statement will return *no data*. The *exception* handler is used to solve the situation by returning “unknown” data.

The first solution uses an *exception* handler. The second solution checks the number before the processing itself.

```
create or replace function Func_gender(p_id char,
                                     p_name out char,
                                     p_surname out char)
    return char
is
    v_gender varchar2(7);
begin
    select decode(substr(p_id, 3, 1), 5, 'female', 6, 'female',
                                0, 'male', 1, 'male',
                                'unknown'),
           name, surname into v_gender, p_name, p_surname
    from personal_data;
    return v_gender;
exception
    when no_data_found then
        p_name := 'unknown';
        p_surname := 'unknown';
        v_gender := 'unknown';
end;
/
```

```

create or replace function Func_gender(p_id char,
                                     p_name out char,
                                     p_surname out char)

    return char
is
    v_gender varchar2(7);
    v_count integer;
begin
    select count(*) into v_count
    from personal_data
    where personal_id = p_id;
    if v_count = 0 then
        p_name := 'unknown';
        p_surname := 'unknown';
        v_gender := 'unknown';
    else
        select decode(substr(p_id, 3, 1), 5, 'female', 6, 'female',
                        0, 'male', 1, 'male',
                        'unknown'),
               name, surname into v_gender, p_name, p_surname
        from personal_data;
    end if;
    return v_gender;
end;
/

```

Which of the previous solutions will be more effective and less time demanding? Explain why?

The result of the previous *User_arguments* query will be the following (it has been extended by attribute *IN_OUT* characterizing the mode of the parameters). So now, we get complex information about parameters.

```

select object_name, argument_name, position,
       data_type, data_length, IN_OUT
from user_arguments
where object_name = 'FUNC_GENDER';

```

OBJECT_NAME	ARGUMENT_NAME	POSITION	DATA_TYPE	DATA_LENGTH	IN_OUT
FUNC_GENDER	(null)	0	CHAR	(null)	OUT
FUNC_GENDER	P_ID	1	CHAR	(null)	IN
FUNC_GENDER	P_NAME	2	CHAR	(null)	OUT
FUNC_GENDER	P_SURNAME	3	CHAR	(null)	OUT

The second solution is the result of the *Description* command (parameters are ordered based on the *position* attribute of *User_arguments* data dictionary view).

```

ARGUMENT_NAME      TYPE      IN/OUT  DEFAULT
-----
<return_value>     CHAR      OUT      unknown
P_ID                CHAR      IN        unknown
P_NAME              CHAR      OUT      unknown
P_SURNAME           CHAR      OUT      unknown

```

Procedures cannot return value using the *Return* command (only *OUT* parameters are available). Therefore, there is no row with *position* = 0. Procedure *Proc_sex* functionality is the same, but the gender is returned by the *OUT* parameter (header is listed).

```
create or replace procedure proc_sex(p_id char, p_name out char,
                                     p_surname out char, p_sex out char)
...
```

OBJECT_NAME	ARGUMENT_NAME	POSITION	IN_OUT	DATA_LENGTH
PROC_SEX	P_SEX	4	OUT	(null)
PROC_SEX	P_SURNAME	3	OUT	(null)
PROC_SEX	P_NAME	2	OUT	(null)
PROC_SEX	P_ID	1	IN	(null)

14.3.8 Managing sequences

Information about the sequence status can be obtained using the *User_Sequences* data dictionary view. It contains the following attributes (all possibilities of the definition):

Tab. 14.1: *User_sequences*

Column	Datatype	NULL	Description
SEQUENCE_OWNER	VARCHAR2(30)	NOT NULL	Name of the owner of the sequence.
SEQUENCE_NAME	VARCHAR2(30)	NOT NULL	Sequence name.
MIN_VALUE	NUMBER		The minimum value of the sequence.
MAX_VALUE	NUMBER		The maximum value of the sequence.
INCREMENT_BY	NUMBER	NOT NULL	Value by which sequence is incremented.
CYCLE_FLAG	VARCHAR2(1)		Does sequence wrap around on reaching the limit?
ORDER_FLAG	VARCHAR2(1)		Are sequence numbers generated in order?
CACHE_SIZE	NUMBER	NOT NULL	Number of sequence numbers to cache.
LAST_NUMBER	NUMBER	NOT NULL	Last sequence number written to disk. If a sequence uses caching, the number written to the disk is the last number placed in the sequence cache. This number is likely to be greater than the last sequence number that was used.

For principle demonstration, create the following sequence:

```
create sequence seq_id
start with 1000
increment by 1
minvalue 1000
maxvalue 2000
cache 10;
```

For getting actual state of the sequence, *User_sequences* data dictionary view will be used:

```
select sequence_name, min_value, max_value,
       increment_by, cycle_flag, cache_size, last_number
from user_sequences
where sequence_name = 'SEQ_ID';
```

SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	CYCLE_FLAG	CACHE_SIZE	LAST_NUMBER
SEQ_ID	1000	2000	1	N	10	1000

If you get the next value of the sequence, the *last_number* attribute value will be incremented by 10.

SEQUENCE_NAME	MIN_VALUE	MAX_VALUE	INCREMENT_BY	CYCLE_FLAG	CACHE_SIZE	LAST_NUMBER
SEQ_ID	1000	2000	1	N	10	1010

14.4 Practice

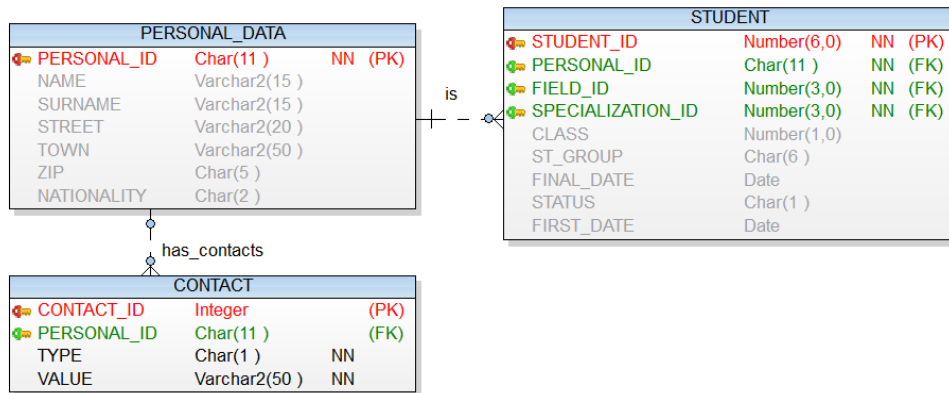


Fig. 14.1: Student submodel – practice

Consider the provided figure (fig. 14.1). For the next practice, try to get all information from the data dictionary, use the listed figure only for checking the correctness of the result.

1. Get the list of attributes, data types, and *NULL* flags for the table *STUDENT*.
2. Get the list of attributes, data types, and *NULL* flags for the table *PERSONAL_DATA*.
3. Get the system identifier of the *primary key* of the table *STUDENT*.
4. Get the attributes forming the *primary key* in the table *STUDENT*.
5. Get the attributes forming the *primary key* in the table *STUDY_SUBJECTS*.
6. Get the constraint name in the table *STUDENT* referencing *PERSONAL_DATA* table.
7. Drop the *primary key* in the table *PERSONAL_DATA*. Consider the prerequisites for dropping based on *referential integrity*.
8. Create the *primary key* again with the non-system-generated name.
9. Check the provided name for the *primary key*.
10. Interconnect table *PERSONAL_DATA* and *STUDENT* once again with the non-system generated name.
11. Get the constraint name for the reference between table *PERSONAL_DATA* and *STUDENT*.
12. List all *procedures* and *functions* owned by you.
13. List all *procedures* and *functions*, which are accessible to you, but not owned by you.
14. Create a *sequence* for getting the *STUDENT_ID* attribute value. Set the actual *sequence* position to the correct value.
15. Create a *trigger* for setting the *STUDENT_ID* attribute automatically.
16. Add the cache (5 values) for the defined sequence.
17. Get the information about the sequence (like *increment*, *cycle*, *actual position*, ...) and *trigger* from the *data dictionary*. What about the *triggering event*?

Lab 15 – Reports

The output of the `Select` statement is commonly “table” formatted. In this lab, we will expand the technology by the reports providing various output layouts and formats. By this lab, the reader will be able to present results in the table, graph or correlated report styles. He will understand how to configure reports, providing the outputs in PDF documents, HTML formats or XML. He will also be able to export the reports into Excel, CSV, general delimited format, textual or XML types.

Remember that it is not enough to have the data and know how to get the required outputs. The design and the presentation of the results are crucial.

15.1 Overview

The *report* is an output of the developing tool based on data stored in the database. In our case, we will deal with the *Report* extension of the *SQL developer* tool. Defined report outputs can be sent to the printer directly or saved in many formats:

- HTML
- PDF
- XML
- CSV
- Microsoft Excel formats (XLS, XLSX)
- ...

In principle, the *report* can be considered the user-friendly formatted output of the *Select* statement in the form of a table, graph, binding child tables, and export (e.g., input for *SQL Loader*). Thus, the report forms the layer between data stored in the database and presentations for the management of the company. Whereas each report is currently evaluated *Select* statement, each data change is automatically reflected. Therefore, report can be considered as dynamic performance output. Looking at the history, two report management streams can be perceived. The first system was based on a report generated in the *SQL console*. Report presentation could be done only in the text form, either as formatted text or *XML*. Although the result might be saved (e.g., using *SPOOL* commands), subsequent processing, export, and publishing were too complicated due to editing layout necessity and design problems. Fig. 15.1 shows the example of the report generated in the console using *TITLE*, *COLUMN*, *COMPUTE* and *BREAK ON* commands encapsulating the *Select* statement itself. Notice that the following report has been developed by only one *Select* statement. Whereas console solutions are not used anymore, a more sophisticated approach has been proposed.

Student report							21.03.2022
Class	Student ID	Full name	Form	Avg	Best	Worst	Count
=====	=====	=====	=====	=====	=====	=====	=====
1	501448	Andrej Janci	Bc.	02.83	02.00	04.00	03
	550127	Rudolf Kovac	Ing.	03.00	02.00	04.00	02
*****				-----			
Avg-class				02.92			
2	500425	Jaroslav Cipak	Ing.	03.32	01.50	04.00	38
	500426	Alojz Gazo	Ing.	02.83	01.00	04.00	32
	500431	Zoltan Sim	Ing.	02.54	01.00	04.00	41
	500432	Zdenko Olzbut	Ing.	02.68	01.00	04.00	36
	500438	Miroslav Gmuca	Ing.	01.98	01.00	04.00	24
	501319	Branislav Balaz	Bc.	02.70	02.00	04.00	05
	501559	Rastislav Kontros	Bc.	04.00	04.00	04.00	01
	550807	Lubos Lehotsky	Bc.	02.63	01.00	04.00	04
*****				-----			
Avg-class				02.83			
Page:		1					

Fig. 15.1: Report in a console environment

For this lab and consecutive presentations and publishing on the web, we will use the *SQL Developer* tool. Individual reports will be managed locally on the client-side but based on server data (*cloud, localhost or on-premise*).

15.2 Environment settings, background

To allow using *Oracle reports*, it is necessary to enable its processing in the *SQL developer* tool by clicking on the **View** tab and selecting **Reports**.

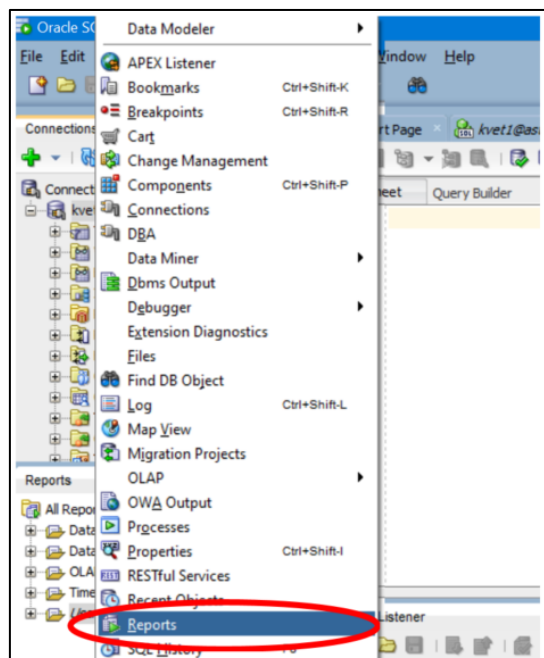


Fig. 15.2: Report navigation in SQL developer

Individual report characteristics with the name of created ones are visible in the separate window.

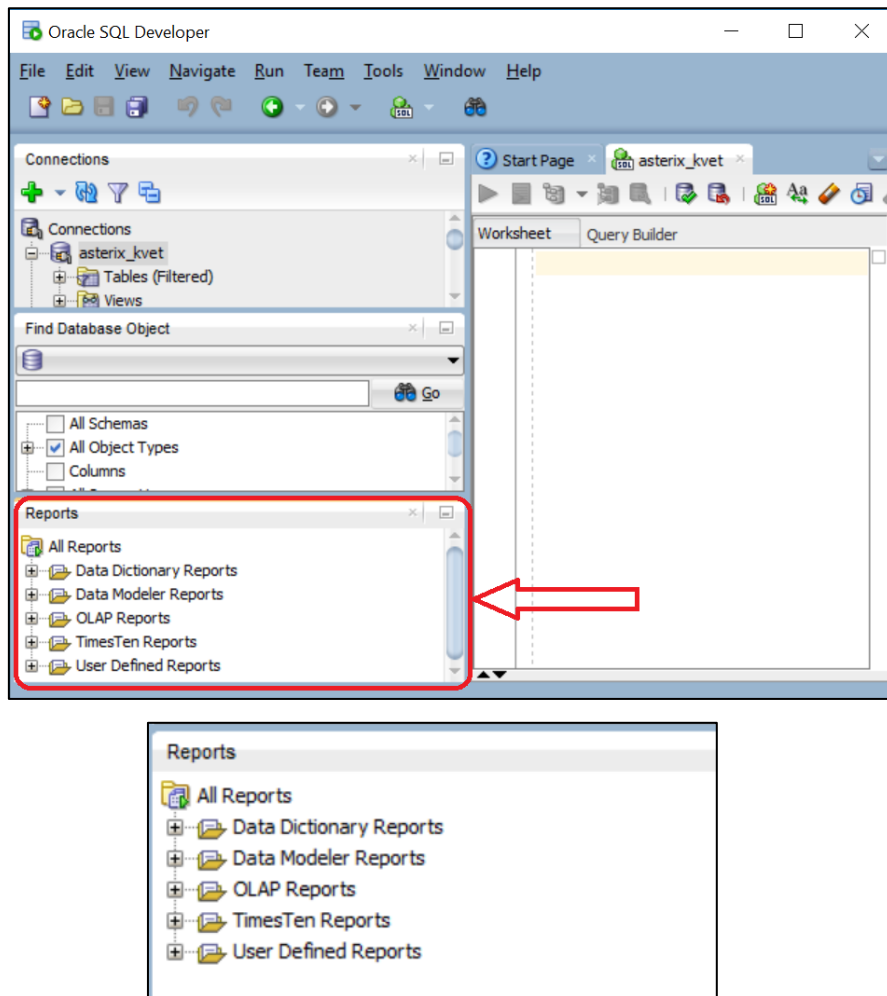


Fig. 15.3: Reports window in SQL developer

There are some pre-prepared reports available for you to highlight the opportunities and power of the reports. There are many categories, and we will deal with only some of them because our focus is mostly on user report definition creation. The layer **Data Dictionary Reports** consists of reports based on the database system characteristics and defined objects like *constraints*, *indexes*, *triggers*, *tables*, etc. Let's see the information about the *tables* defined by the connected user in the *SQL developer* session. These data are accessible using **Data Dictionary Reports** => **Table** => **User_tables** view.

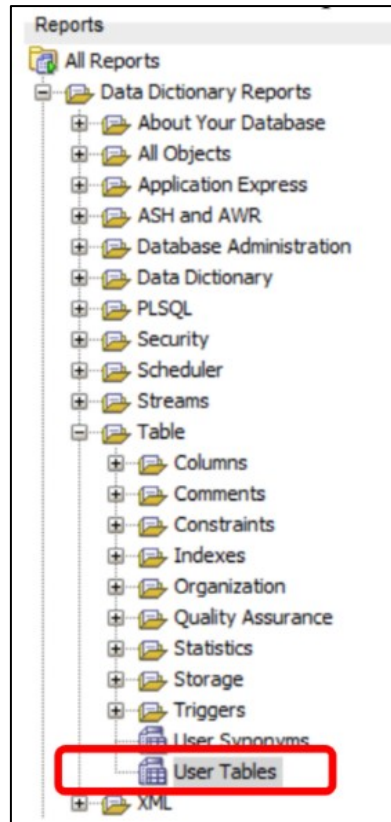


Fig. 15.4: Report tree in SQL developer

Each table is physically located in the defined *tablespace* (if not written explicitly, *default tablespace* for user objects will be used).

The defined report also consists of the name of all tables created by the particular user, many characteristics, and associated statistics (e.g., *num_rows*, *blocks*, *empty_blocks*, ...).

Statistics themselves provide a really powerful apparatus used by the optimizer to make decisions about data access. In the past, *Analyze* functionality has been used. However, nowadays, it is supported only for *backward* compatibility. It has been replaced by the **DBMS_STATS** package, which can be launched either manually or automatically during the maintenance window. **DBMS_STATS** package is characterized by these methods:

```
DBMS_STATS.GATHER_INDEX_STATS
-- Index statistics
DBMS_STATS.GATHER_TABLE_STATS
-- Table, column, and index statistics
DBMS_STATS.GATHER_SCHEMA_STATS
-- For all objects in a schema
DBMS_STATS.GATHER_DATABASE_STATS
-- For all objects in a database
DBMS_STATS.GATHER_SYSTEM_STATS
-- CPU and I/O statistics for the system
```

Obtaining statistics for the whole *schema* can be done using the following code. *Kvet_eng* is the name of the *schema* (user). “*Cascade => true*” option forces the database to collect statistics for all *indexes* on *table/schema*.

```
execute DBMS_STATS.GATHER_SCHEMA_STATS('KVET_ENG', cascade => true);
```

Obtaining statistics only for one table is reflected by the following code. *Person* delimits the table name to be evaluated (*tab_name*). Parameter *ownname* characterizes the owner of the table.

```
execute DBMS_STATS.GATHER_TABLE_STATS(ownname => 'KVET_ENG',
                                       tabname => 'PERSON',
                                       cascade => true);
```

Actual statistics are stored in the data dictionary views (see [Lab 14 – Data dictionary views](#)), like *DBA_TABLES*, *DBA_TAB_COLUMNS*, *DBA_TAB_STATISTICS*, *DBA_TAB_COL_STATISTICS*, *DBA_TAB_MODIFICATIONS*, etc.

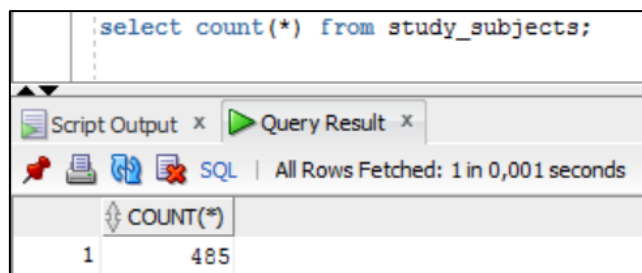
Generated statistics include the following information:

- Table statistics
 - Number of rows
 - Number of blocks
 - Average row length
- Column statistics
 - Number of distinct values (*NDV*) in column
 - Number of NULL values in column
 - Data distribution (histogram)
- Index statistics
 - Number of leaf blocks
 - Levels
 - Clustering factor
- System statistics
 - I/O performance and utilization
 - CPU performance and utilization

Auxiliary statistics (e.g., extended histograms) specific to a *SQL* statement can be obtained using an *SQL profile*.

It is important to emphasize that the data are based on statistics, which are evaluated periodically, so the change in the table will be reflected only after the new statistics processing, not immediately. It will also be imaged using the following example.

Let’s have the table **study_subjects**. The cardinality of it is **485** (your results can vary based on executed data operations).



The screenshot shows a SQL query window with the query: `select count(*) from study_subjects;`. Below the query, there are tabs for 'Script Output' and 'Query Result'. The 'Query Result' tab is active, showing a table with one row and one column. The column is labeled 'COUNT(*)' and the value is '485'. Above the table, it says 'All Rows Fetched: 1 in 0,001 seconds'.

COUNT(*)
485

Fig. 15.5: Cardinality of the table *Study_subjects* (real value)

If you delete all the data from the table and end the transaction, the cardinality of such a table will be 0.

```
delete from study_subjects;
commit;
```

However, existing (original) statistics are still used, so the amount of data in the table is still **484** (after the statistics collecting, one more row has been inserted. Afterwards, all data have been deleted. If statistics have not been recollected, they store original values – cardinality: **484**). Consequently, the optimizer will get incorrect data for decision-making.

Therefore, if many changes are performed, it is useful and recommended to set up new statistics to reflect significant data changes. So, without reconstructing statistics, we will still get old data, optimizer decisions can be inappropriate.

Tab. 15.1: Table report

TABLE_NAME	TABLESPACE_NAME	NUM_ROWS	BLOCKS	DATE_LAST_ANALYZED	LAST_ANALYZED	...
STUDY_SUBJECTS	USERS	484	%	21.03.2022 14:50:55	2.21 days ago	...

TABLE_NAME	TABLESPACE_NAME	LOGGING	NUM_ROWS	BLOCKS	EMPTY_BLOCKS	AVERAGE_ROW_LENGTH	CACHE	DATE_LAST_ANALYZED	LAST_ANALYZED
1 PERSONAL_DATA	USERS	YES	35	5	0	63	N	21.03.2022 14:50:54	3 minutes ago NO
2 STUDENT	USERS	YES	37	5	0	46	N	21.03.2022 14:50:54	3 minutes ago NO
3 STUDY_SUBJECTS	USERS	YES	484	5	0	35	N	21.03.2022 14:50:55	3 minutes ago NO
4 ST_FIELD	USERS	YES	9	5	0	34	N	21.03.2022 14:50:55	3 minutes ago NO
5 ST_PROGRAM	USERS	YES	637	5	0	21	N	21.03.2022 14:50:55	3 minutes ago NO
6 SUBJECT	USERS	YES	218	5	0	34	N	21.03.2022 14:50:55	3 minutes ago NO
7 SUBJECT_YEAR	USERS	YES	372	5	0	23	N	21.03.2022 14:50:55	3 minutes ago NO
8 TEACHER	USERS	YES	32	5	0	25	N	21.03.2022 14:50:55	3 minutes ago NO

Fig. 15.6: Table report

Notice that reports are dynamic. Thus, after obtaining new statistics, particular data are automatically replaced also in reports.

To focus, evidence rebuild statistics for the current user by executing the *gather_schema_stats* procedure of the *dbms_stats* package. Explanation of the package methods and parameters is out of the scope of this subject.

```
exec dbms_stats.gather_schema_stats(
    ownname => '&ownname',
    estimate_percent => 20,
    method_opt => 'for all columns size auto',
    options => 'Gather',
    cascade => true,
    degree => 4);
```

When executing a previous procedure, you will be prompted to get the *username* for who statistics should be recollected. It is based on parameters defined after the **&** symbol. In this case, your username should be written to substitute **&ownername** parameter with a real value.

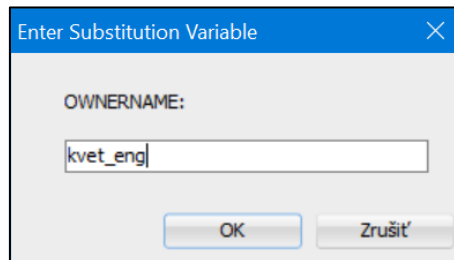


Fig. 15.7: Parameter substitution

Then, when looking at the generated report, data will be updated automatically and will express correct values.

Tab. 15.2: Table report

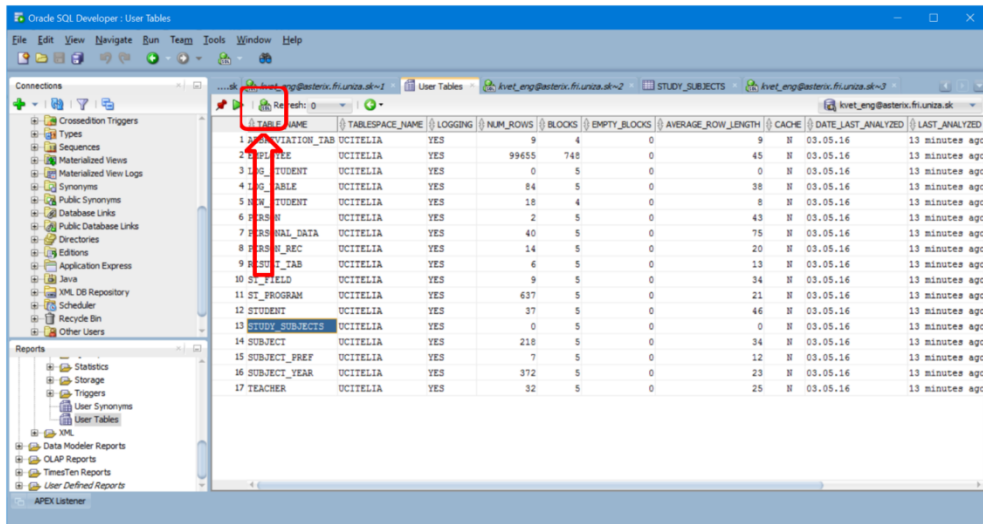
<i>TABLE_NAME</i>	<i>TABLESPACE_NAME</i>	<i>NUM_ROWS</i>	<i>BLOCKS</i>	<i>EMPTY_BLOCKS</i>	<i>AVERAGE_ROW_LENGTH</i>
STUDY_SUBJECTS	USERS	0	5	0	0

When you double-click on the table in the report, you can get the schema of such table (it is provided by the binding techniques, which will be described later).

	<i>COLUMN_NAME</i>	<i>DATA_TYPE</i>	<i>NULLABLE</i>	<i>DATA_DEFAULT</i>	<i>COLUMN_ID</i>	<i>COMMENTS</i>
1	SCHOOL_YEAR	NUMBER(4,0)	No	(null)	1	(null)
2	STUDENT_ID	NUMBER(6,0)	No	(null)	2	(null)
3	SUBJECT_ID	VARCHAR2 (30 BYTE)	No	(null)	3	(null)
4	LECTURER	CHAR (5 BYTE)	No	(null)	4	(null)
5	RESULT	VARCHAR2 (1 BYTE)	Yes	(null)	5	(null)
6	EXAM_DATE	DATE	Yes	(null)	6	(null)
7	SIGN_DATE	DATE	Yes	(null)	7	(null)
8	ECTS	NUMBER(2,0)	Yes	(null)	8	(null)

Fig. 15.8: Report

To get the statement, which forms the report, click on the SQL button to run the report in *SQLWorksheet*.



	TABLE_NAME	TABLESPACE_NAME	LOGGING	NUM_ROWS	BLOCKS	EMPTY_BLOCKS	AVERAGE_ROW_LENGTH	CACHE	DATE_LAST_ANALYZED	LAST_ANALYZED
1	ADVISATION_TAB	UCITELIA	YES	9	4	0	9	N	03.05.16	13 minutes ago
2	APPRAISER	UCITELIA	YES	99655	745	0	45	N	03.05.16	13 minutes ago
3	APPRAISER_TAB	UCITELIA	YES	0	5	0	0	N	03.05.16	13 minutes ago
4	APPRAISER_TAB	UCITELIA	YES	84	5	0	38	N	03.05.16	13 minutes ago
5	APPRAISER_TAB	UCITELIA	YES	18	4	0	9	N	03.05.16	13 minutes ago
6	APPRAISER_TAB	UCITELIA	YES	2	5	0	43	N	03.05.16	13 minutes ago
7	APPRAISER_DATA	UCITELIA	YES	40	5	0	75	N	03.05.16	13 minutes ago
8	APPRAISER_REC	UCITELIA	YES	14	5	0	20	N	03.05.16	13 minutes ago
9	APPRAISER_TAB	UCITELIA	YES	6	5	0	13	N	03.05.16	13 minutes ago
10	APPRAISER_TAB	UCITELIA	YES	9	5	0	34	N	03.05.16	13 minutes ago
11	ST_PROGRAM	UCITELIA	YES	637	5	0	21	N	03.05.16	13 minutes ago
12	STUDENT	UCITELIA	YES	37	5	0	46	N	03.05.16	13 minutes ago
13	STUDENT	UCITELIA	YES	0	5	0	0	N	03.05.16	13 minutes ago
14	STUDENT	UCITELIA	YES	218	5	0	34	N	03.05.16	13 minutes ago
15	SUBJECT_PREF	UCITELIA	YES	7	5	0	12	N	03.05.16	13 minutes ago
16	SUBJECT_YEAR	UCITELIA	YES	372	5	0	23	N	03.05.16	13 minutes ago
17	TEACHER	UCITELIA	YES	32	5	0	25	N	03.05.16	13 minutes ago

Fig. 15.9: Getting query forming report

15.3 Filtering, sorting

Let's go back to the main topic of this lab – user-defined reports. They are managed by the *User Defined Reports* part at the end of the *Reports* segment. The new report is defined after right-clicking on the item and choosing *New Report*.

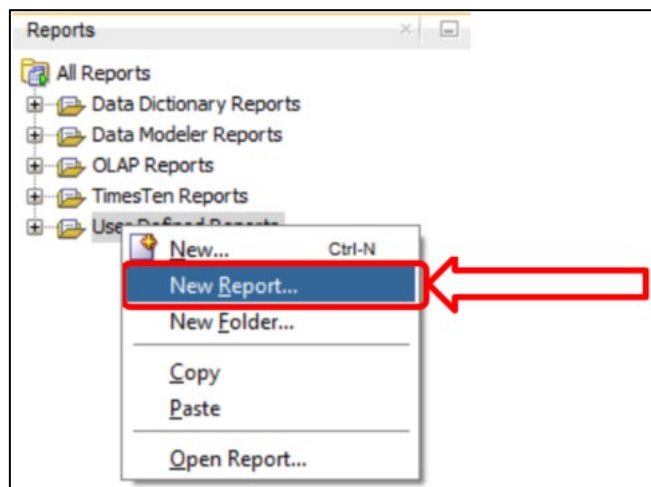


Fig. 15.10: New report

The window for the report definition consists of these parts, which are described:

- **Name** – each report must have a unique name, by which you can find and reference it in the system.
- **Style** – *table, chart, gauge, code, script, PL/SQL DBMS Output*. For this lab, we will deal with the tables and chart reports. Now, select the *Table* option.
- **Description** – optional, description of the provided functionality.
- **SQL statement** forming the input data for the report.

So, let's create the first simple table report, which will consist of personal and student data using the following *Select* statement.

```
select personal_id, name, surname, student_id, class, status
from personal_data JOIN student using(personal_id);
```

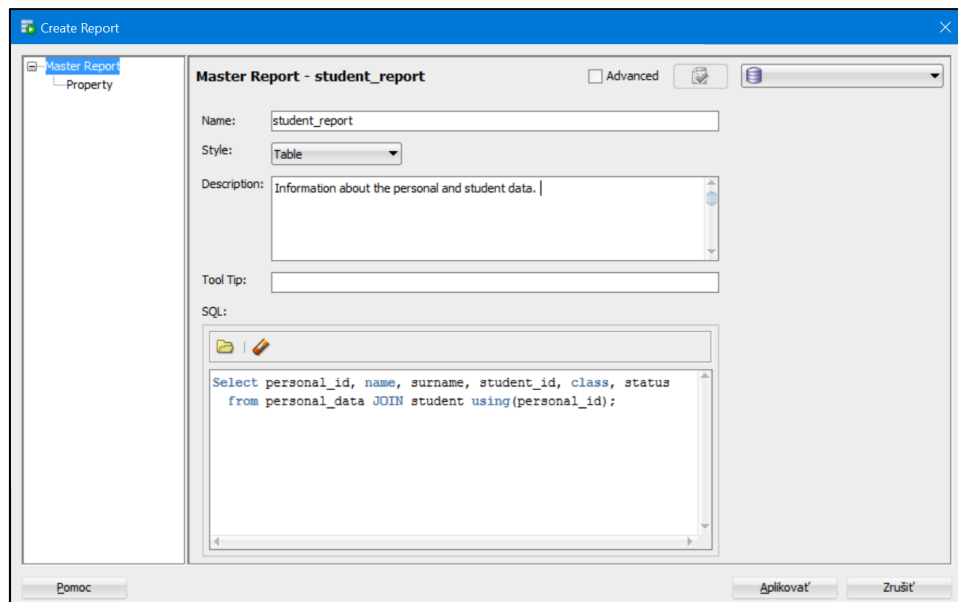


Fig. 15.11: Report definition

After settings confirmation, a new report will be created, which is visible in the **Report** section.

The output of such report is in table form:

	<i>PERSONAL_ID</i>	<i>NAME</i>	<i>SURNAME</i>	<i>STUDENT_ID</i>	<i>CLASS</i>	<i>STATUS</i>
1	781015/4431	Peter	Roger	550020	3	S
2	791229/5431	Jack	Robinson	501333	1	S
3	791229/5431	Jack	Robinson	501103	0	K
4	791229/5431	Jack	Robinson	501096	0	V
5	800407/3522	Mark	Bailey	501402	2	S
6	800407/3522	Mark	Bailey	501555	1	S

Fig. 15.12: Report output

Data can be user-managed in the grid, so they can be sorted by choosing attribute names for sorting criteria definition (ascending, descending). The symbol in the *Surname* expresses the selected sorting criterion based on that column.

<i>PERSONAL_ID</i>	<i>NAME</i>	↑ A Z	<i>SURNAME</i>	<i>STUDENT_ID</i>	<i>CLASS</i>	<i>STATUS</i>
--------------------	-------------	-------	----------------	-------------------	--------------	---------------

Fig. 15.13: Data sorting in report

Disadvantage if such sorting criteria definition, only one attribute be used. Thus, if you select another attribute (e.g., *name*), it will be sorted by *name*, not the combination of the *surname* and *name*. That opportunity can be provided by right-clicking on the header and by choosing the **Sort** option. In that case, a complex sort criterion can be defined

(in the first example, it is sorted by two attributes – *surname* and *name*, all values are sorted ascendant. Vice versa, the second example sorts the result set based on *personal_id* (*desc*) and *student_id* (*asc*)).

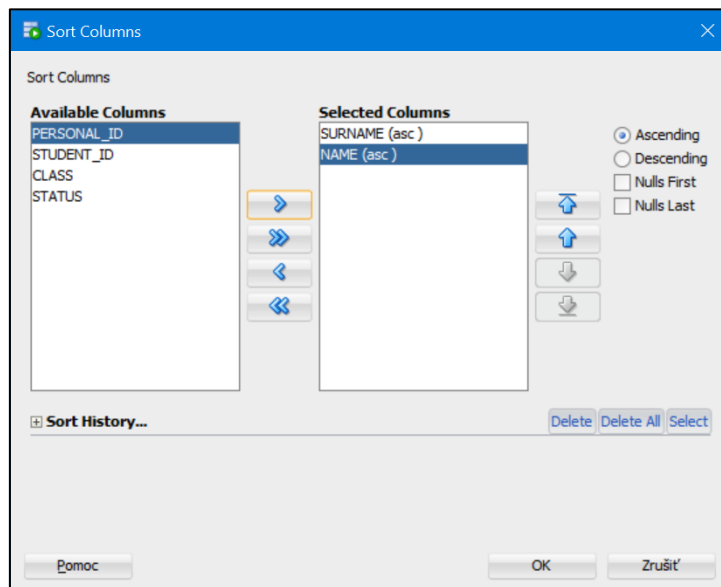


Fig. 15.14: Data sorting in report

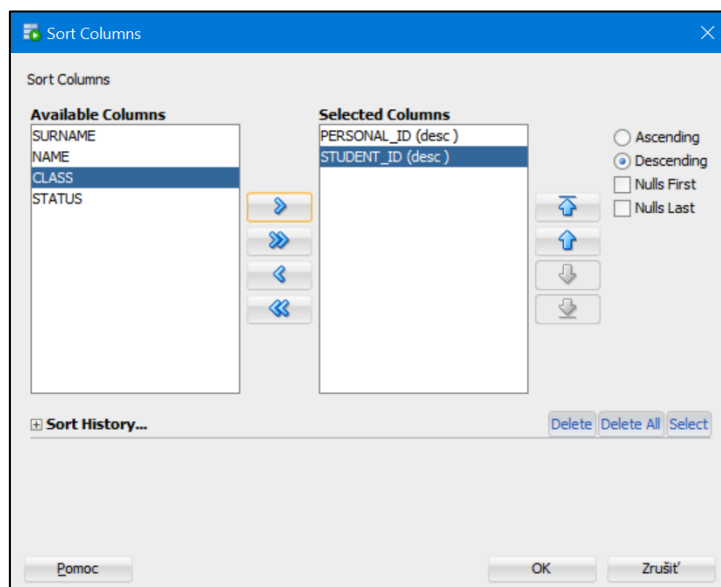


Fig. 15.15: Data sorting in report

↓ z A 1 PERSON...	NAME	SURNAME	↓ 2 1 2 STUDEN...	CLASS	STATUS
921225/7452	Sim	Eas	501559	2	S
911001/3623	Mark	Vox	501448	1	S
901130/4454	Jack	Clever	501003	2	S
900913/3326	Jacob	Murgas	550945	0	S
896123/5471	Suzanne	Walker	550123	1	S
890608/4543	Jacob	Homm	550807	2	S
890310/2145	Arnas	Mitchell	501345	2	S
871203/5472	Tom	Moore	501201	3	A
870913/3326	Jacob	Murgas	501381	2	S
860907/1259	John	Young	501414	2	S
860103/2238	John	Young	550127	1	S
855122/8569	John	Pearce	550698	2	S
850130/3695	Carol	Pearce	550545	1	A
841106/3456	Michael	Pearce	501512	3	S
840821/8027	Hugo	Davis	500425	2	S
840410/6777	Milan	Clarke	500426	2	K

Fig. 15.16: Data sorting in report

Moreover, you can filter the result set and search for particular data. It can be done by clicking on a particular column header. Then, choose the **Filter Column** item.

↓ z A 1 PERSON...	NAME	SURNAME	↓ 2 1 2 STUDEN...	CLASS	STATUS
1 921225/7452	Sim	Auto-fit All Columns Auto-fit Selected Column Columns... Sort... Delete Persisted Settings... Copy Selected Column Header(s) Filter Column...	501559	2	S
2 911001/3623	Mark		501448	1	S
3 901130/4454	Jack		501003	2	S
4 900913/3326	Jacob		550945	0	S
5 896123/5471	Suzanne		550123	1	S
6 890608/4543	Jacob		550807	2	S
7 890310/2145	Arnas		501345	2	S
8 871203/5472	Tom		501201	3	A
9 870913/3326	Jacob		501381	2	S
10 860907/1259	John		501414	2	S

Fig. 15.17: Filtering (1)

In the input box, write conditions based on equality or wildcard. Let's the filter form the students in the **class = 2**.

Filter:CLASS	X
2	
	2

Fig. 15.18: Filtering (2)

Another filter can be based on *status* = 'S'.

Filter:STATUS	X
= 'S'	
S	

Fig. 15.19: Filtering (3)

In that case, the result set will be based only on actual students of the second class. The funnels express defined filters.

	↓ ↕ ↗ PERSON...	↑ NAME	↑ SURNAME	↓ ↕ ↗ ² STUDEN...	↑ ...	↓ ST...
1	921225/7452	Sim	Eas	501559	2	S
2	901130/4454	Jack	Clever	501003	2	S
3	890608/4543	Jacob	Hoom	550807	2	S
4	890310/2145	Arnas	Mitchell	501345	2	S
5	870913/3326	Jacob	Murgas	501381	2	S
6	860907/1259	John	Young	501414	2	S
7	855122/8569	John	Pearce	550698	2	S
8	840821/8027	Hugo	Davis	500425	2	S
9	840312/7845	Jack	Smith	501469	2	S
10	830703/7486	Charlie	Lewis	500429	2	2
11	830514/5341	Wiliam	Whitel	501319	2	S
12	800407/3522	Mark	Bailey	501402	2	S

Fig. 15.20: Report output (1)

String values can also be filtered based on a wildcard, e.g., to get only the actual second class student list, whose first name starts with the “J” letter, another filter can be added. In this case – *wildcard* type is used. The condition would be following:

where name like 'J%'

Therefore, in the input box, the right part of the condition is written – *like 'J%'*

Filter:CLASS	X
like 'J%'	

Fig. 15.21: Filtering (4)

The solution looks like this:

	↓ ↕ ↗ PERSON...	↑ N...	↑ SURNAME	↓ ↕ ↗ ² STUDEN...	↑ ...	↓ ST...
1	901130/4454	Jack	Clever	501003	2	S
2	890608/4543	Jacob	Hoom	550807	2	S
3	870913/3326	Jacob	Murgas	501381	2	S
4	860907/1259	John	Young	501414	2	S
5	855122/8569	John	Pearce	550698	2	S
6	840312/7845	Jack	Smith	501469	2	S

Fig. 15.22: Report output (2)

Individual filters can be removed at once (right-click on the header and choose **Remove All Filters**) or individually based on the header of the particular column (right click of the particular column header and choose **Filter column...**):

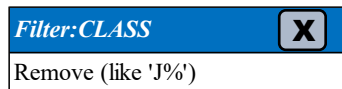


Fig. 15.23: Removing filtering option

If only one filter is removed, other ones will continue to be defined so that the result set will consist of actual (**status = 'S'**) *second-class* students:

	↓ z A 1 PERSON...	↑ NAME	↑ SURNAME	↓ z 2 STUDEN...	↑ ...	↓ ST...
1	921225/7452	Sim	Eas	501559	2	S
2	901130/4454	Jack	Clever	501003	2	S
3	890608/4543	Jacob	Hoom	550807	2	S
4	890310/2145	Arnas	Mitchell	501345	2	S
5	870913/3326	Jacob	Murgas	501381	2	S
6	860907/1259	John	Young	501414	2	S
7	855122/8569	John	Pearce	550698	2	S
8	840821/8027	Hugo	Davis	500425	2	S
9	840312/7845	Jack	Smith	501469	2	S
10	830703/7486	Charlie	Lewis	500429	2	S
11	830514/5341	Wiliam	Whitel	501319	2	S
12	800407/3522	Mark	Bailey	501402	2	S

Fig. 15.24: Report output

Dealing with *NULL* values in the filter and sorting criteria is another characteristic, which can, however, cause some problems. Therefore, we will explain it using the example. In the previous text, a report has been based on **INNER JOIN** between *personal_data* and *student* table. Thus, if the person is not referenced in the student table, such person will not be part of the result set at all. To remove this restriction, create a new report based on **OUTER JOIN** – all *personal_data* should be listed regardless of the row existence in the *student* table.

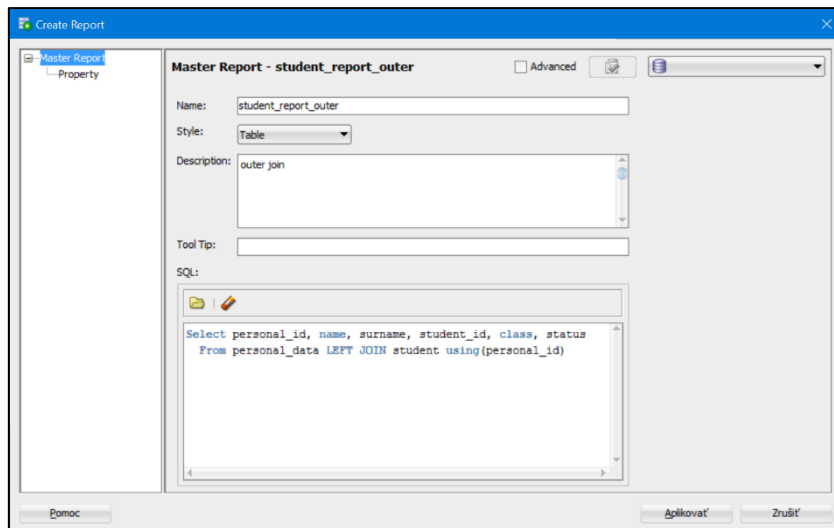


Fig. 15.25: Report definition

The result set consists of all personal data (used attributes – *personal_id*, *name*, *surname*), and if possible, it will also contain student data. Thus, if the interconnection cannot be done, it will have *NULL* values in the student part.

	<i>PERSONAL_ID</i>	<i>NAME</i>	<i>SURNAME</i>	<i>STUDENT_ID</i>	<i>CLASS</i>	<i>STATUS</i>
1	601224/6526	Michael	Flower	(null)	(null)	(null)
2	601224/6537	(null)	(null)	(null)	(null)	(null)
3	740210/6525	Carol	Matiasako	(null)	(null)	(null)
4	740210/6536	Michael	Flower	(null)	(null)	(null)
5	781015/4431	Peter	Roger	550020	3	S
6	791229/5431	Jack	Robinson	501333	1	S
7	791229/5431	Jack	Robinson	501103	0	K
8	791229/5431	Jack	Robinson	501096	0	V

Fig. 15.26: Report output – managing *NULL* values

However, how to sort such data with *NULL* values? Remember that such *NULL* values cannot be compared directly, so they must be handled separately. In that case, it is possible to choose whether *NULL* values are processed at the beginning or just at the end. Selection is made using radio buttons in the sorting definition area (*in the application, it is modeled by the checkboxes, but the functionality is the same as radio button functionality, so no more than one option can be selected*). By default, *NULL* values data are processed at the end of the list for ascending sorting.

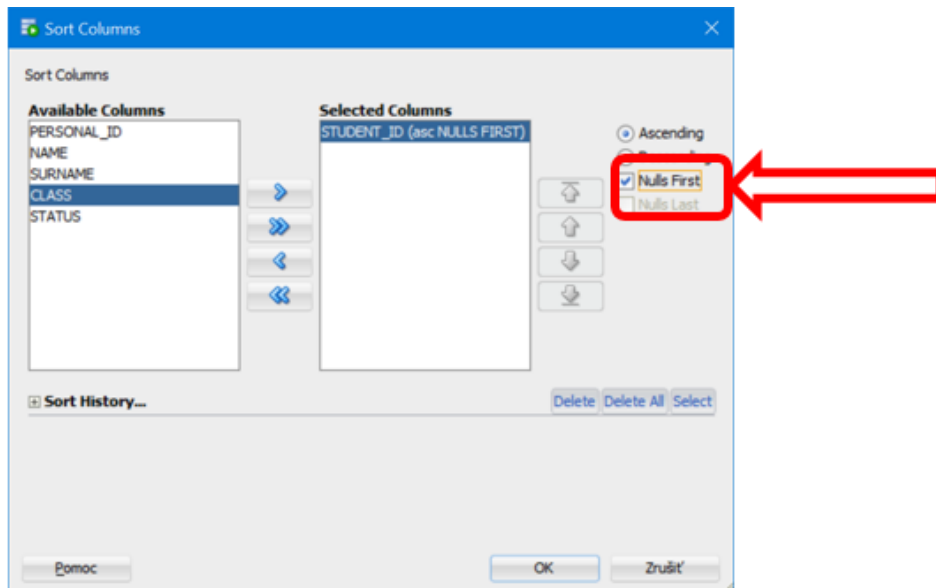


Fig. 15.27: Managing NULL values

15.4 Hidden columns

The report is associated with the data provided by the *Select* statement. We can define table columns aliases, which will be reflected as the name of the attributes in the result set:

```
select personal_id as pid, name, surname, student_id, class, status
from personal_data JOIN student using(personal_id);
```

	PID	NAME	SURNAME	STUDENT_ID	CLASS	STATUS
1	78105/5431	Peter	Roger	550020	3	S
2	79129/5431	Jack	Robinson	501333	1	S
3	79129/5431	Jack	Robinson	501103	0	K
4	79129/5431	Jack	Robinson	501096	0	V

Fig. 15.28: Report output

We recommend you use aliases for used functions. However, for reports, it is not necessary.

Hidden columns are part of the *Select* statement associated with the report but are not displayed in the result set. They can be used for output formatting or binding to another report. Columns to be hidden can be removed from the result set by clicking on the data grid header and choosing the *Columns...* option.

	↓ z A 1 PERSON...	NAME	SURNAME	↓ z 1 2 STUDEN...	CLASS	STATUS
1	921225/7452	Sim	Eas	Auto-fit All Columns		
2	911001/3623	Mark	Vox	Auto-fit Selected Column		
3	901130/4454	Jack	Clever	Columns...		
4	900913/3326	Jacob	Murgas	Sort...		
5	896123/5471	Suzanne	Walker	Delete Persisted Settings...		
6	890608/4543	Jacob	Homm	Copy Selected Column Header(s)		
7	890310/2145	Armas	Mitchell	Filter Column...		
8	871203/5472	Tom	Moore			
9	870913/3326	Jacob	Murgas			
10	860907/1259	John	Young	501414	2	S

Fig. 15.29: Hidden column definition

By this choice, you can manage the order of columns in the result set (left part) and determine attributes to be hidden (right part). Notice that using this option does not influence the original *Select* statement at all. So, in the following example, the *personal_id* attribute will be hidden.

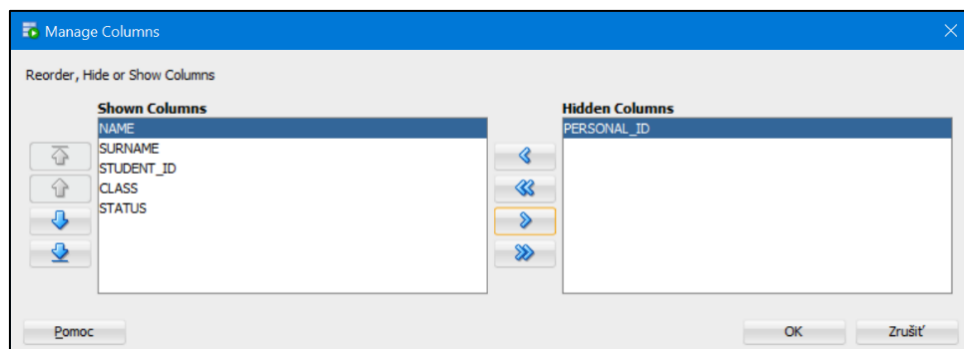


Fig. 15.30: Hidden column definition

15.5 Binding multiple reports – Master – Child

As partially mentioned, hidden columns are used for binding results to another report or its part (child). Let's have the following task. One report should contain personal data. The second one should collect information about the student if the person has already been part of the student data. In this case, the second report is called the *child*, whereas the results are dependent on the actual choice (actual selection) in the first report, which is called *master*. How does it work? You should create a master report and its child report, which will be interconnected by the bindings:

Master report – let's create a *person_binding* report consisting of *name*, *surname*, and *personal_id* attributes.

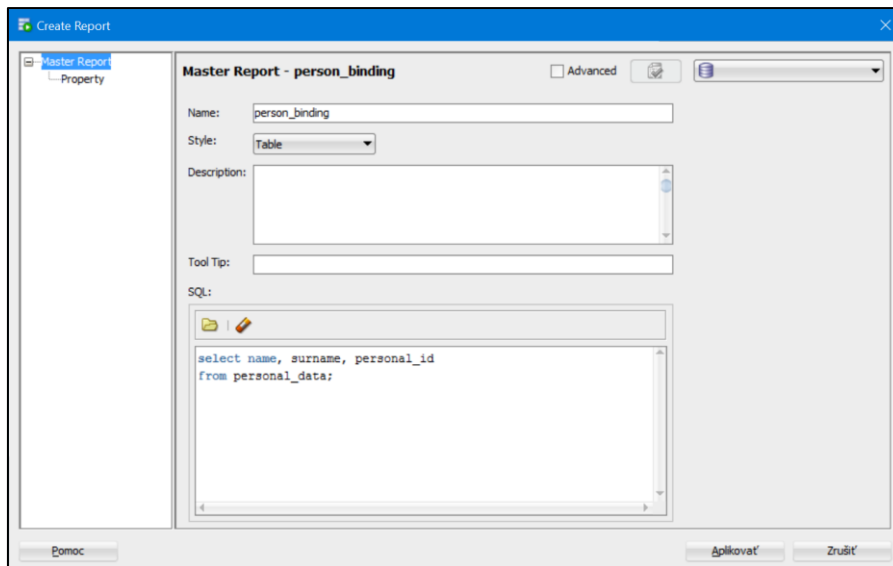


Fig. 15.31: Master report

Now, you can **Edit** it and add a child report to that defined – right-click on the defined report and choose the **Edit...** option.

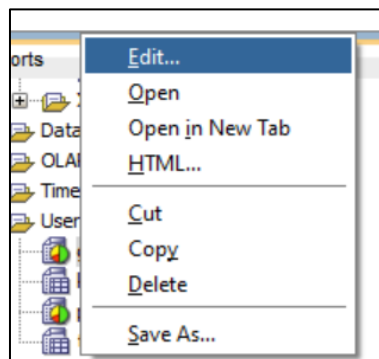


Fig. 15.32: Master report – edit option

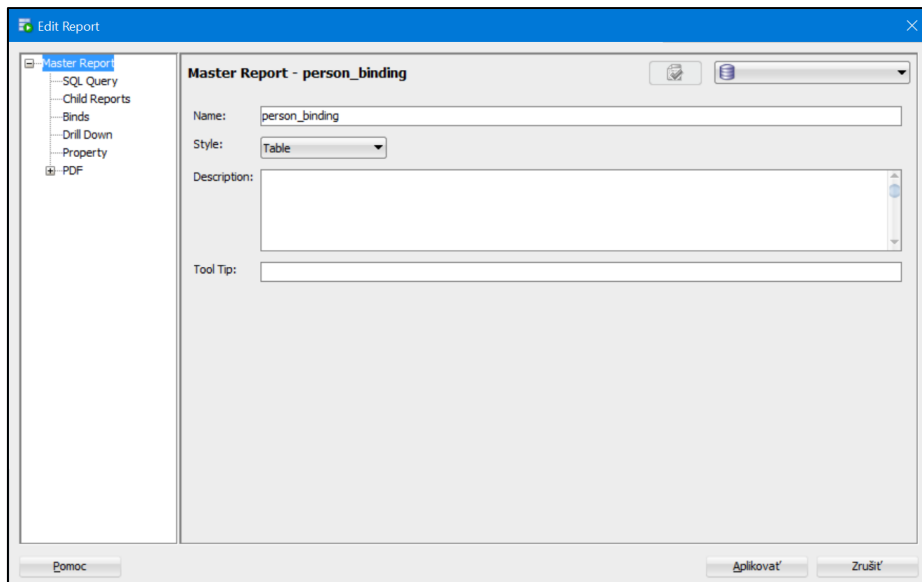


Fig. 15.33: Master report – edit option

In this window, you can edit the individual setting of the report. The defined *Select* statement can be found in the **SQL Query** option of the **Master report** branch. However, now, we will mainly highlight the **Child Reports** option.

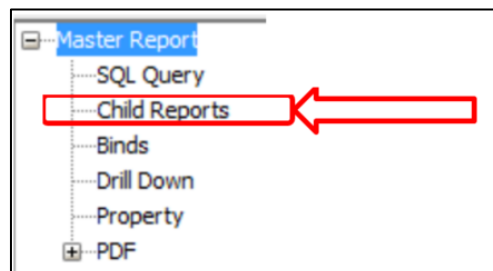


Fig. 15.34: Child report

Child report must also have a unique name (*student_binding*) and will consist of student data in our case. Whereas it is associated with the master one, performance should be dynamic. Thus, if the selection in the master report is changed, it must be automatically reflected and synchronized with the child report.

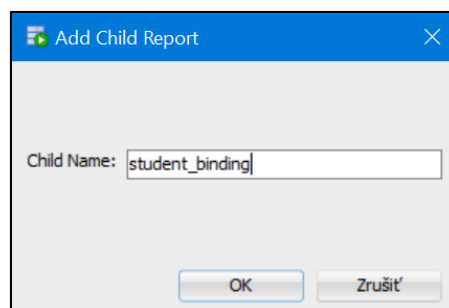


Fig. 15.35: Naming child report

Thus, the child report *Select* statement definition will look like this:

```
select student_id, class, status, first_date
from student
where personal_id = :PERSONAL_ID;
```

In this case, the referential integrity provides binding, so the binding is ensured by the **:PERSONAL_ID**, defined in the master report. Notice that the binding variable is prefixed by the colon.

Select statement definition for the child report can be set in this branch direction: **Master report => Child reports => name_of_the_child_report => SQL Query.**

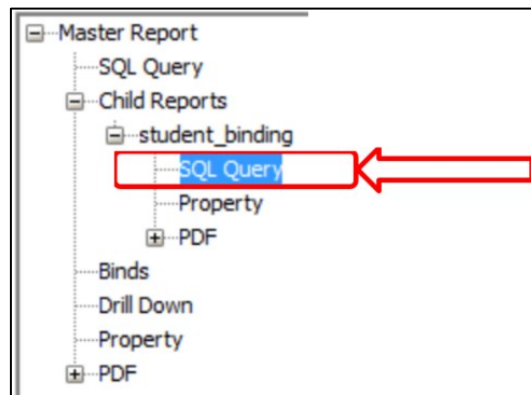


Fig. 15.36: Child report query

The child report definition window will then look like this:

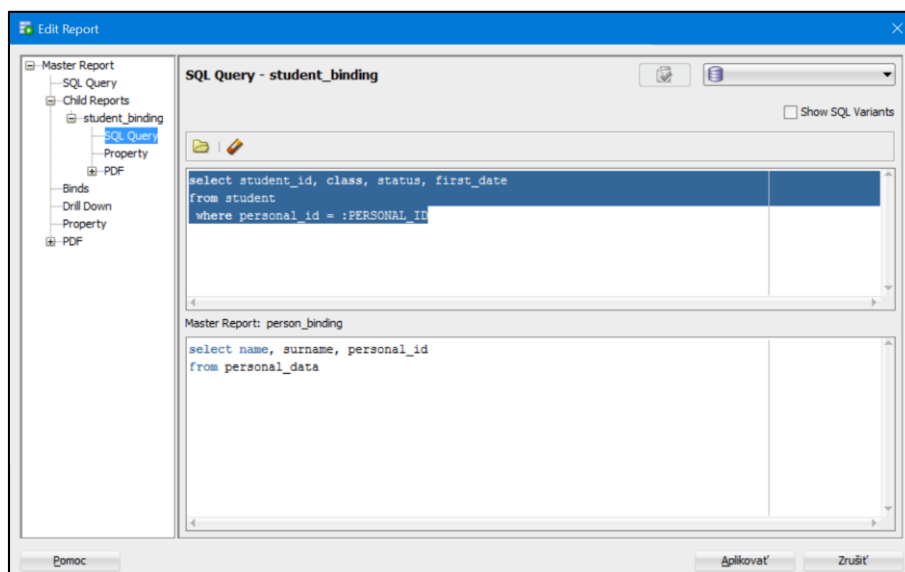


Fig. 15.37: Child report query

When the definition is applied, and you choose some cell in the master report, particular data based on bindings will be shown in the child report.

	NAME	SURNAME	PERSONAL_ID	
1	Michael	Pearce	841106/3456	
2	Jack	Smith	840312/7845	
3	John	Young	860907/1259	
4	Carol	Pearce	850130/3695	
5	Carol	Pearce	841201/1248	
6	William	Whittel	830514/5341	
7	Peter	Roger	781015/4431	
▲▼				
student_binding				
🔄 Refresh: 0 ▼				
	STUDENT_ID	CLASS	STATUS	FIRST_DATE
1	501567	0	E	31.08.06
2	501319	2	S	(null)

Fig. 15.38: Report binding

In the previous example, person *William Whittel* has been studied twice (with **student_id** = **501567** and **501319**). On the other hand, when we choose e.g., person *Milan Clarke*, he has studied only once.

	NAME	SURNAME	PERSONAL_ID	
17	Jacob	Murgas	870913/3326	
18	Jacob	Hoom	890608/4543	
19	John	Young	860103/2238	
20	Suzanne	Walker	896123/5471	
21	John	Pearce	855122/8569	
22	Peter	Murphy	830914/7748	
23	Milan	Clarke	840410/6777	
▲▼				
student_binding				
🔄 Refresh: 0 ▼				
	STUDENT_ID	CLASS	STATUS	FIRST_DATE
1	500426	2	K	12.06.08

Fig. 15.39: Report binding

The evaluation is done automatically.

Just in the binding, we see the significant importance of the hidden column definition. Indeed, attribute **personal_id** must be provided by the master *Select* statement but does not need to be visible in the result set.

The Hidden column can provide sufficient power. It will be part of the *Select* statement but not displayed in the master report result set. *Master SQL query* will, therefore, contain **personal_id** attribute values. However, such an attribute will be invisible to the user report.

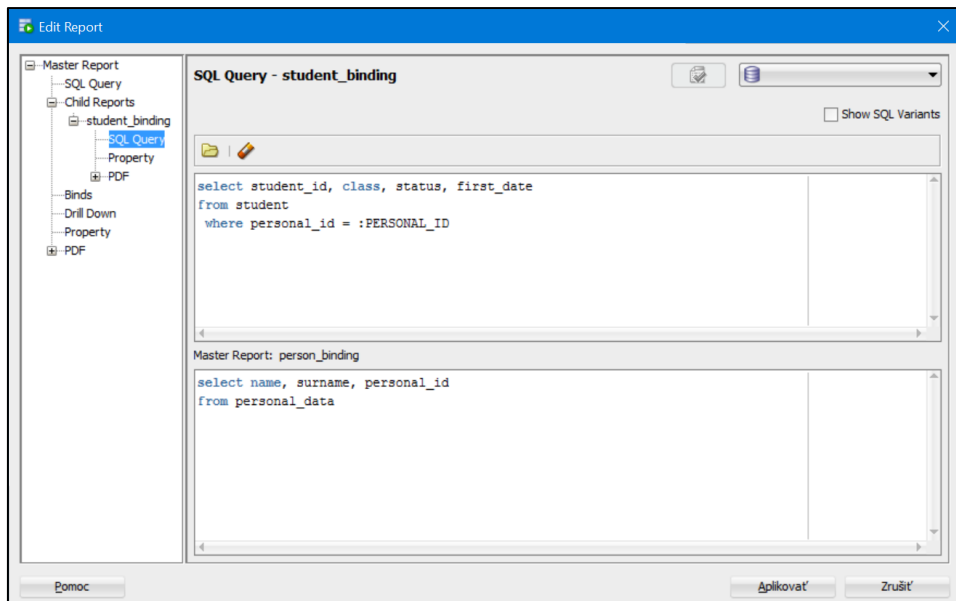


Fig. 15.40: Report binding

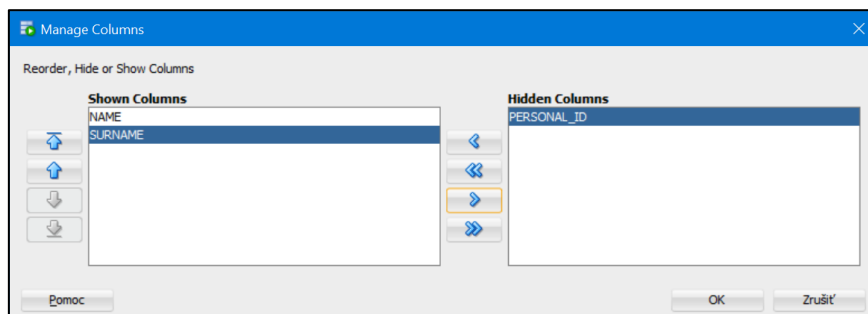


Fig. 15.41: Managing report

So, the result set (master report) will not contain the primary key of the *personal_data* table. However, the binding will be correct.

	NAME	SURNAME
1	Michael	Pearce
2	Jack	Smith
3	John	Young
4	Carol	Pearce
5	Carol	Pearce
6	William	Whittel
7	Peter	Roger

student_binding				
Refresh: 0				
	STUDENT_ID	CLASS	STATUS	FIRST_DATE
1	501512	3	S	

Fig. 15.42: Managing binding

Be aware, no warning nor exception will be raised to inform the user that binding cannot be done. However, child report will always be empty as a consequence of impossible binding.

	NAME	SURNAME
1	Michael	Pearce
2	Jack	Smith
3	John	Young
4	Carol	Pearce
5	Carol	Pearce
6	William	Whittel
7	Peter	Roger

▲▼

student_binding

Refresh: 0

▼

NAME

SURNAME

STUDENT_ID

CLASS

STATUS

FIRST_DATE

Fig. 15.43: Managing binding

It is possible to define multiple child reports. However, they must be associated with the same master report – it is impossible to chain multiple child reports (associate child report to another child report cannot be done).

Let's have the master report consisting of teacher information. One child report can contain information about lectured subjects of a particular teacher. The second one can deal with guaranteed subjects.

The master report will be based on the following *Select* statement:

```
select teacher_id, name, surname, department from teacher;
```

The first report (*lecturer_report*) will be created using this *Select* statement:

```
select distinct school_year, subject_id, name
from study_subjects join subject using(subject_id)
where lecturer = :TEACHER_ID;
```

The second report (*guarantee_report*) will be created using this *Select* statement:

```
select distinct school_year, subject_id, name
from subject_year join subject using(subject_id)
where guarantee = :TEACHER_ID;
```

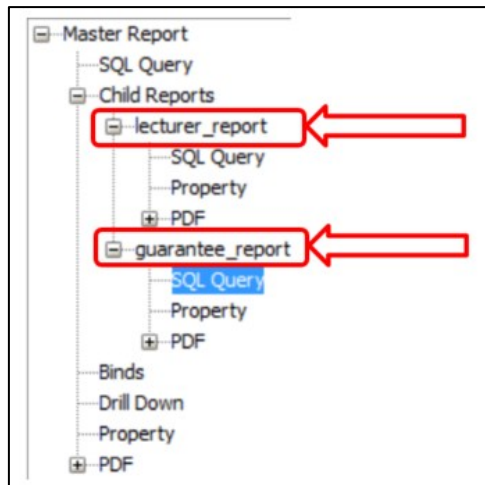


Fig. 15.44: Multiple child reports


The selection on the master report will be synchronized in the child reports.

Lecturer:

	⬆️	TEACHER_ID	⬆️	NAME	⬆️	SURNAME	⬆️	DEPARTMENT
10		KI003		Rachel		Vargas		DI
11		KI005		Mathias		Fortin		DI
12		KTKO2		Jacob		Demers		DTK
13		KDS04		Bill		Rosario		KTN
14		KTKO3		Suzanne		Perreault		DTK
15		KTKO4		Owen		Boudreau		DTK
16		KMT01		Edie		St-Pierre		DMT
17		KDS03		Michael		Rosario		KTN
18		KMME1		John		St-Pierre		DMME
19		EX001		Peter		Frank		EX
20		KIS01		Mathias		Ouellet		DIN

▲▼

lecturer_report guarantee_report

 Refresh: 0 ▼

	⬆️	SCHOOL_YEAR	⬆️	SUBJECT_ID	⬆️	NAME
1		2002		BI10		Java
2		2002		BS01		Operation systems
3		2006		II03		Database systems - administration
4		2002		BA12		Graphs theory
5		2001		BI30		Language C++
6		2006		BI06		Database systems - the best subject :)


Fig. 15.45: Lecturer report

Guarantee:

	↑	TEACHER_ID	↑	NAME	↑	SURNAME	↑	DEPARTMENT
10		KI003		Rachel		Vargas		DI
11		KI005		Mathias		Fortin		DI
12		KTKO2		Jacob		Demers		DTK
13		KDS04		Bill		Rosario		KTN
14		KTKO3		Suzanne		Perreault		DTK
15		KTKO4		Owen		Boudreau		DTK
16		KMT01		Edie		St-Pierre		DMT
17		KDS03		Michael		Rosario		KTN
18		KMME1		John		St-Pierre		DMME
19		EX001		Peter		Frank		EX
20		KIS01		Mathias		Ouellet		DIN

▲▼

lecturer_report guarantee_report

 Refresh: 0 ▼

	↑	SCHOOL_YEAR	↑	SUBJECT_ID	↑	NAME
1		2009		BS11		Unix
2		2005		II03		Database systems - administration
3		2009		IPA2		Internet of things 3
4		2009		IT12		Database and knowledge discovery
5		2005		BS15		Intranet applications
6		2009		IPA1		Internet of things 2
7		2002		BI10		Java
8		2001		BI30		Language C++

Fig. 15.46: Guarantee report

15.6 Graph reports

In the previous section, we have been dealing with reports in the table form. However, *SQL developer* can also provide services to form graphical output using various chart types, like a bar, pie, line, area, bubble, stock, ...

The following sections contain several examples as the most effective way for modeling and describing principles. It also expresses the possibilities offered by the *SQL Developer Reports* tool. Therefore, create a new report, including records defining the number of times a person has been a student. For simplicity, we will now work only with *Inner Join*. When creating a new report, select the **Chart** option from the combo box **Style**.

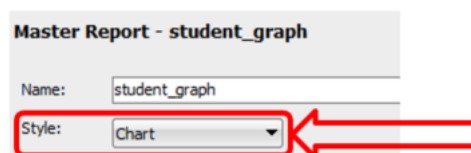


Fig. 15.47: Graph report definition

We will use, **Bar** chart type. Therefore, the *Select* statement should define three attributes, which will be reflected in the graph. The first attribute defines the expression in the x-axis

(if multiple attribute values should be written in the x-axis, they must be formed to the one string using *concatenations*), the second attribute denotes the graph legend (it should be a constant character string). The last attribute of the defined *Select* statement delimits the value in the y-axis. Thus, the *Select* statement will look like this:

```
select personal_id || '-' || name || '-' || surname,
       'number of times, person has been student yet',
       count(student_id)
from personal_data left join student using(personal_id)
group by personal_id, name, surname;
```

First of all, let's see the results in the table form:

	PERSONAL_ID '-' NAME '-' SURNAME	NUMBEROFTIMES,PERSONHASBEENSTUDENTYET	COUNT(*)
1	830703/7486-Charlie-Lewis	number of times, person has been student yet	1
2	840307/7485-Mathias-Thiss	number of times, person has been student yet	1
3	841106/3456-Michael-Pearce	number of times, person has been student yet	1
4	860907/1259-John-Young	number of times, person has been student yet	1
5	820101/8452-Thomas-Simson	number of times, person has been student yet	1
6	871124/3578-Lucas-Austin	number of times, person has been student yet	1
7	871203/5472-Tom-Moore	number of times, person has been student yet	1

Fig. 15.48: Table report output

If you choose the **Chart** option of the **Style** in the definition, the graph will be created. In the left part (tree structure), the **Property** option can be listed with several attributes and parameters influencing the design of the graph. It consists of these five parts: **Data**, **Titles**, **Plot Area**, **X-Axis**, and **Y-Axis**.

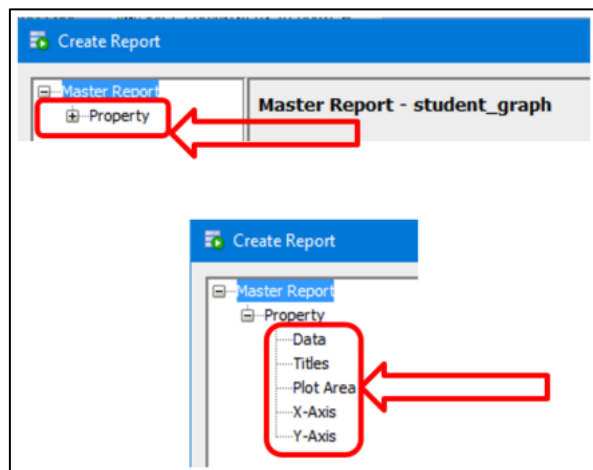


Fig. 15.49: Report properties

In the **Titles** sub-branch, it is possible to format the information to be shown together with the graph properties themselves – title, subtitle, graph, ... In this part, we can also define the font style:

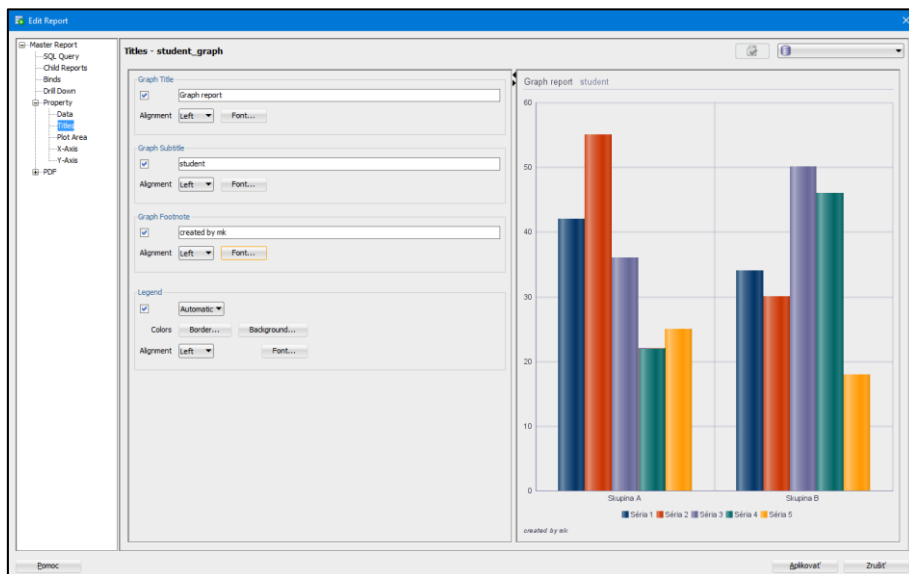


Fig. 15.50: Titles property sub-branch

Plot Area delimits the graphic style of the graph – e.g., colors and borders.

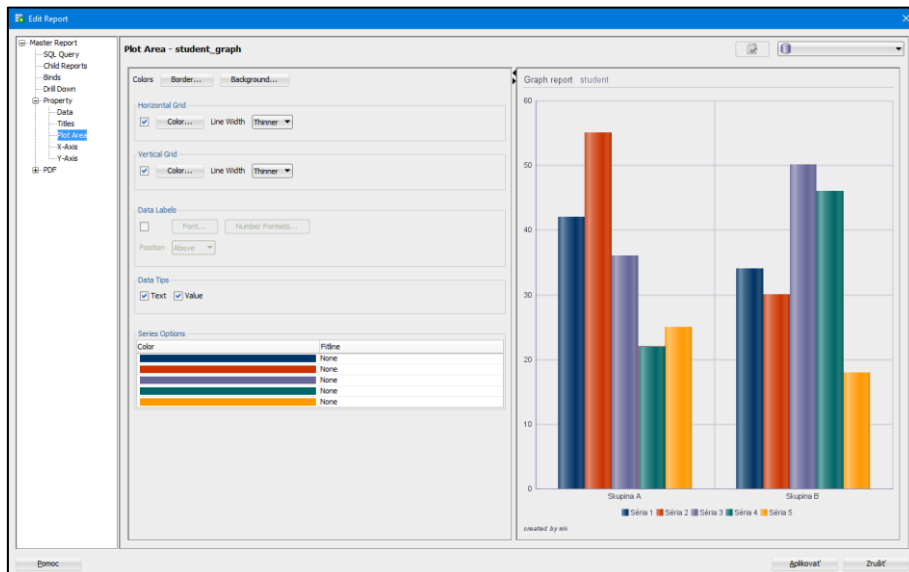


Fig. 15.51: Plot area property sub-branch

X-Axis, **Y-Axis** allow the user to set the scale of axes and design of the line characteristics.

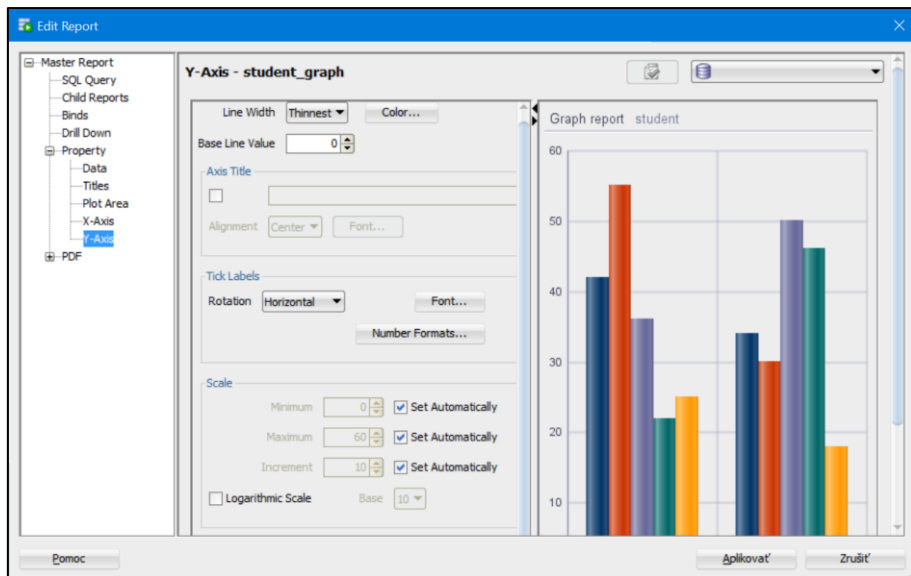


Fig. 15.52: Y-axis property sub-branch

The following figure shows the output of the defined report.

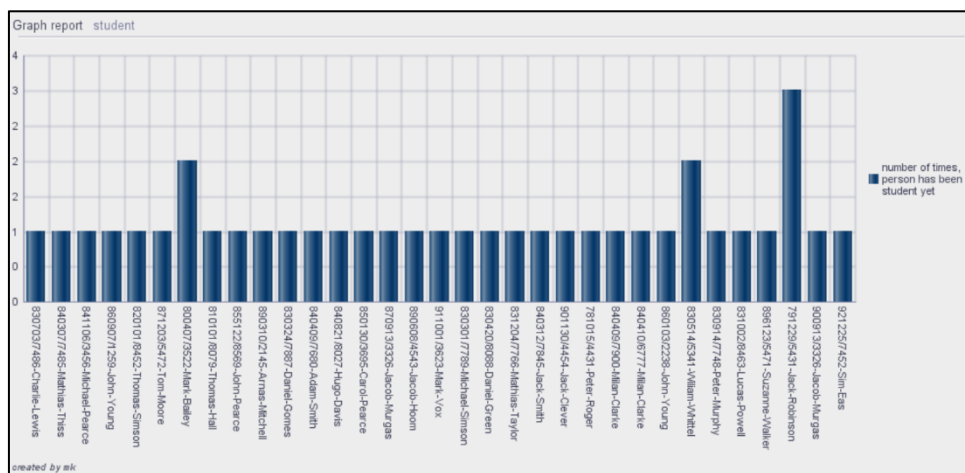


Fig. 15.53: Graph report

As mentioned, such defined reports are dynamic, which can be reflected by the following schema.

Delete the data about the student with *student_id* = 501469 from the database. In standard conditions, it must also be deleted from the *study_subjects* table. However, that person has not registered any subjects yet. Consequently, such a person will also be naturally *deleted* from the report.

It should be emphasized that *Outer Joins* are not reflected in this report, and thus, there will be no data about the people without student information, although such people exist:

```
select personal_id
from personal_data
where personal_id NOT IN (select personal_id from student);
```

	PERSONAL_ID
1	601224/6537
2	880329/1233
3	871124/3578
4	601224/6526
5	841201/1248
6	740210/6536
7	740210/6525

Fig. 15.54: Select statement result set

15.7 Pie graph type reports

Another chart type often used is *Pie*. In this case, output values projected by the *Select* statements are normalized to the 100% range.

Let's create the report, which will contain proportion characteristics of the students in the particular study fields (*field_id*) and specializations (*specialization_id*). Accordingly, the defined *Select* statement should have three attributes. However, compared to the *bar* graph type, the order is a bit different. The first attribute specifies the name of the graph (header description), the second attribute defines the legend, and the last attribute delimits the number to be modeled inside the graph – *count*, which will be normalized and expressed in percentages.

```
select 'Pie proportion graph of the number of students in
particular fields and specializations',
field_name ||', '|| spec_name,
count(*)
from student JOIN st_field using(field_id, specialization_id)
group by field_name, spec_name, field_id, specialization_id;
```

In the *Property* option of the *Master Report* tree branch, the proposed property allows you to define and change graph structure. In this case, choose *Pie* in the *Chart Type* combo box.

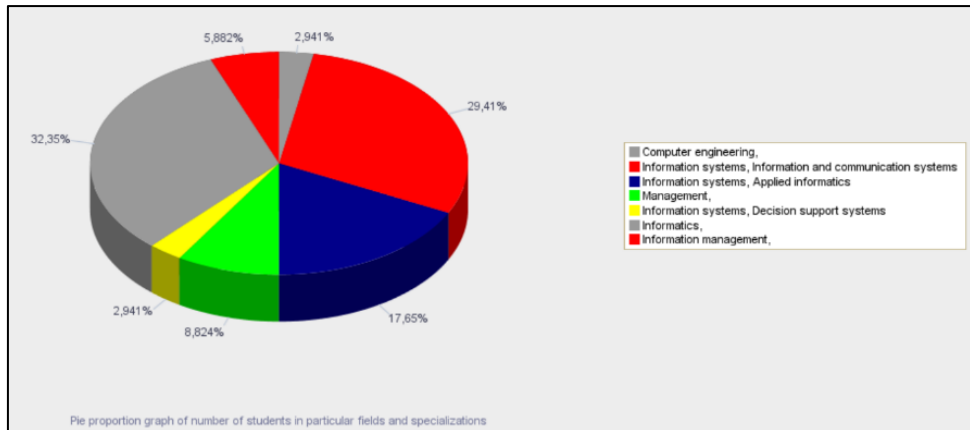


Fig. 15.55: Graph report

In the previous example, the **3D effect** has been applied, which can be checked in **Property** definition:

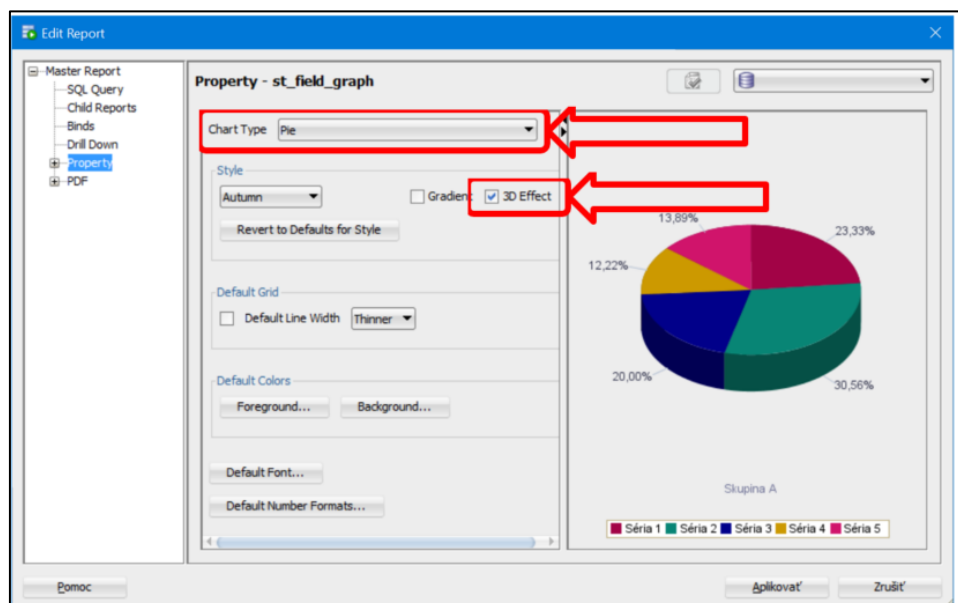


Fig. 15.56: Setting 3D effect

Be strictly aware when defining **Select** statement forming report. During the definition, there is no automatic syntax and semantics check. Consequently, the report will be invalid, and data output will be shown (e.g., the chart will be empty).

Let's have the simple example based on the previous example, but the **From** keyword is missing in the definition.

```
select 'Pie proportion graph of number of students in
particular fields and specializations',
field_name || ', ' || spec_name,
count(*)
from student JOIN st_field using(field_id, specialization_id)
group by field_name, spec_name, field_id, specialization_id;
```

The result will be an empty graph because no data have been found for evaluation.

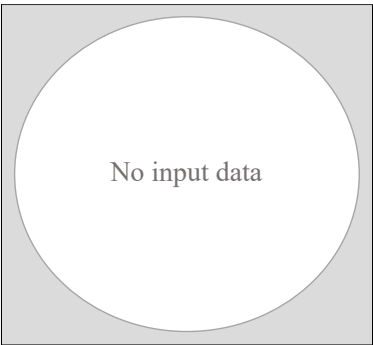


Fig. 15.57: Result of incorrectly defined Graph report

15.8 Line type reports

Line graph type is mainly used for changes and progress monitoring over time. Therefore, for the needs of observations, we will define a new table with random values.

The *sensor_table* table will consist of two attributes – *value* and *time* (time of occurrence). For the simplicity, *value* attribute will be an integer:

```
create table sensor_table(value integer, time date);
desc sensor_table
```

Name	Null?	Type
-----	-----	-----
VALUE		NUMBER(38)
TIME		DATE

Data values will be generated, provided by the anonymous block. We will generate 100 rows. Inside the block, the *dbms_random* package is referenced, which has several methods to be used:

Tab. 15.3: *dbms_random* methods

Method	Description
Random	Returns a random integer greater or equal to <i>-power(2,31)</i> and less than <i>power(2,31)</i> .
String	Generates random string based on the parameters: <ul style="list-style-type: none">opt – specifies what the returning string looks like:<ul style="list-style-type: none">'u', 'U' - returning string in uppercase alpha characters.'l', 'L' - returning string in lowercase alpha characters.'a', 'A' - returning string in mixed case alpha characters.'x', 'X' - returning string in uppercase alpha-numeric characters.'p', 'P' - returning string in any printable characters.Otherwise, the returning string is in uppercase alpha characters.len – length of the string.

Method	Description
Value	<p>Generates random string based on the optional parameters:</p> <ul style="list-style-type: none"> • low • high <p>The function gets a random number, greater than or equal to 0 and less than 1, with 38 digits precision.</p> <p>Alternatively, you can get a random Oracle number <i>x</i>, where <i>x</i> is greater than or equal to low and less than high.</p>

Other initialization methods of the *dbms_random* package:

Tab. 15.4: *dbms_random* methods

Method	Description
INITIALIZE Procedure	Initializes the package with a seed value.
SEED Procedures	Resets the seed.
TERMINATE Procedure	Terminates package.

We will generate random values of the uniform distribution from the interval <1;20), therefore **Value** method will be used. Time granularity will be the day. The appropriate code looks like this:

```
begin
  for i in 1..100 loop
    insert into sensor_table
      values(trunc(dbms_random.value(1, 20)), sysdate - i);
  end loop;
end;
/
```

Notice, whereas the **Value** function of the *dbms_random* package produces real values, they must be truncated. Otherwise, they will be rounded to the whole part so that the table would also include values of 20.

To produce the line graph for such data, the following *Select* statement is defined for the report. Like always, it is determined by the three attributes in the result set. The first attribute stores values in the x-axis (attribute **time**), the second attribute is constant and refers to the legend (**'Time values evolution'**), the third attribute expresses values to be shown in the y-axis (attribute **value**). In this case, values should be ordered to have an informative value.

```
select time, 'Time values evolution', value
  from sensor_table
 order by time;
```

The results will look like this:

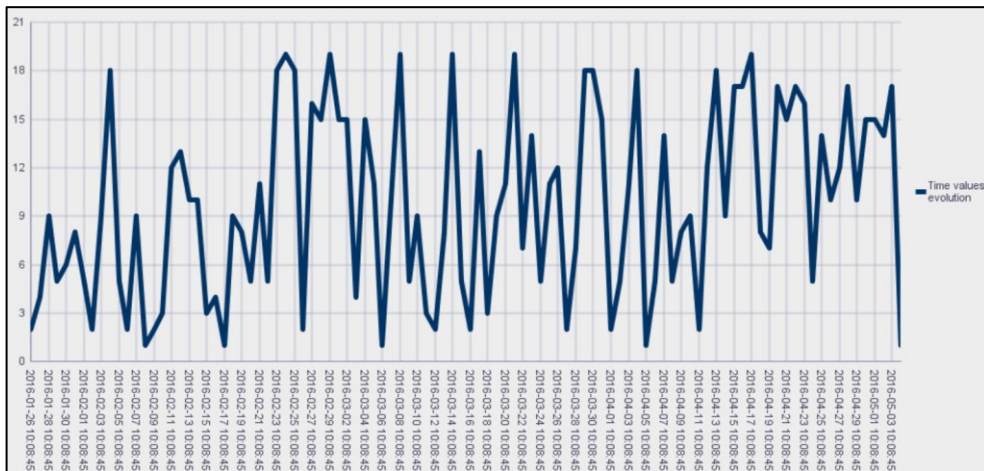


Fig. 15.58: Line report

We can remove the time element of the processing by converting the *time* attribute to the string.

Then, the report will look like this:

```
select to_char(time, 'DD.MM.YYYY'), 'Time values evolution', value
from sensor_table
order by time;
```

Also, design parameters like colors, line strength, and so on can be set in the *Property* branch.

Let's extend the previously defined table by another sensor data results.

```
alter table sensor_table add value2 integer;
```

In this case, generate values for the attribute *value2* using interval <1,5).

```
update sensor_table set value2 = trunc(dbms_random.value(1, 5));
commit;
```

Now, we can define multiple graphs inside one line chart (*other chart types can also be used*).

Follow the instructions:

1. create a new report, name it (*multiple_line_chart*) with the *chart* type.

Fig. 15.59: Report definition

2. *Select* statement for the definition must contain all data, which will be required for the reports (sensorial data (attribute *value*, *value2*) delimited by the *time* attribute). It should also contain constants for legends). The order of attributes is not fundamental. We will show how to map them to the result set (graph).

```
select to_char(time, 'DD.MM.YYYY'), value, value2, 'report - value1',
      'report - value2'
from sensor_table;
```

3. In the *Property* section, choose the *Line* of the *Chart Type* combo box.

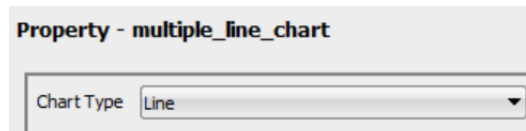


Fig. 15.60: Setting property Chart Type

4. Now, it's time to map attributes for the graph. This functionality can be set *Data* subsection of the *Property* branch in *Master Report*:

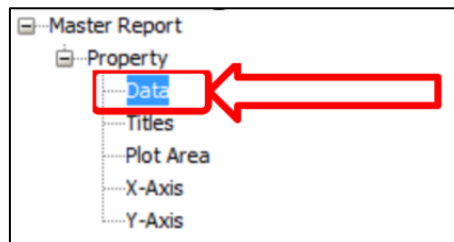


Fig. 15.61: Data subsection

5. For this purpose, an active connection to the database must be provided to execute mapping, so choose the appropriate connection in the right part of the window:

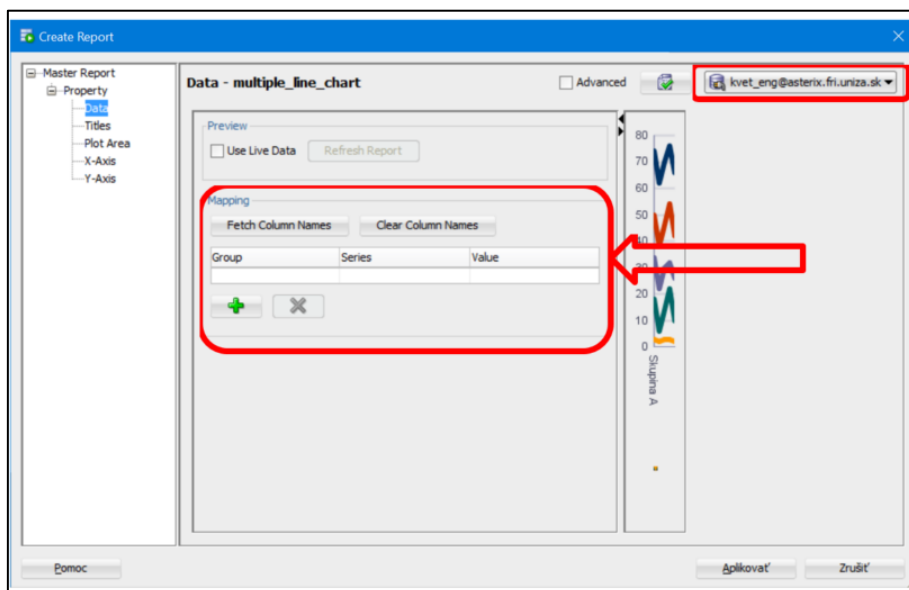


Fig. 15.62: Mapping

6. Now, the mapping is enabled. However, you must fetch column names available from the associated *Select* statement (click on the **Fetch Column Names** button):

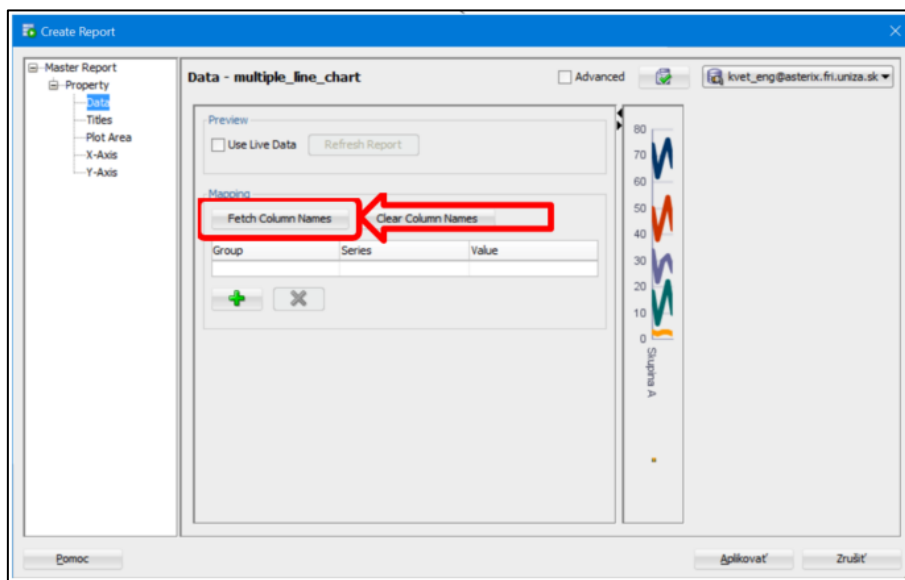


Fig. 15.63: Mapping

7. Fill the *Mapping* section of the data grid. The first column defines the attribute to be processed in the x-axis. The second column represents the legend, so use constants. The last (third) column of the data grid delimits the attribute association for the y-axis.
8. We will create two graphs inside one chart. In our case – attribute **time** with removed time spectrum (only *day*, *month*, and *year* are processed) is used for the x-axis, another axis (y-axis) will be defined by sensorial data – attributes **value**, **value2**. Thus, the data grid for mapping will look like this:

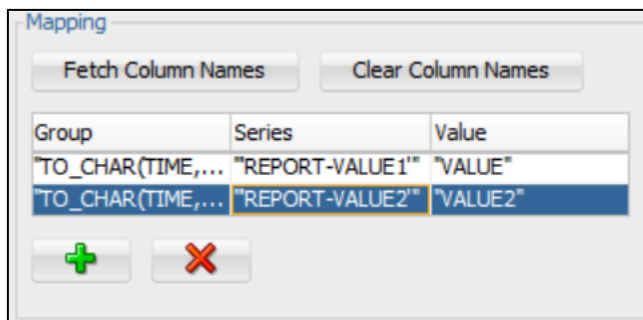


Fig. 15.64: Mapping

9. Apply changes and create a report. By default, the *value1* of the report will have a red color, *value2* will be shown in blue color; however, the settings can be changed in the **Plot area** of the **Property** branch.

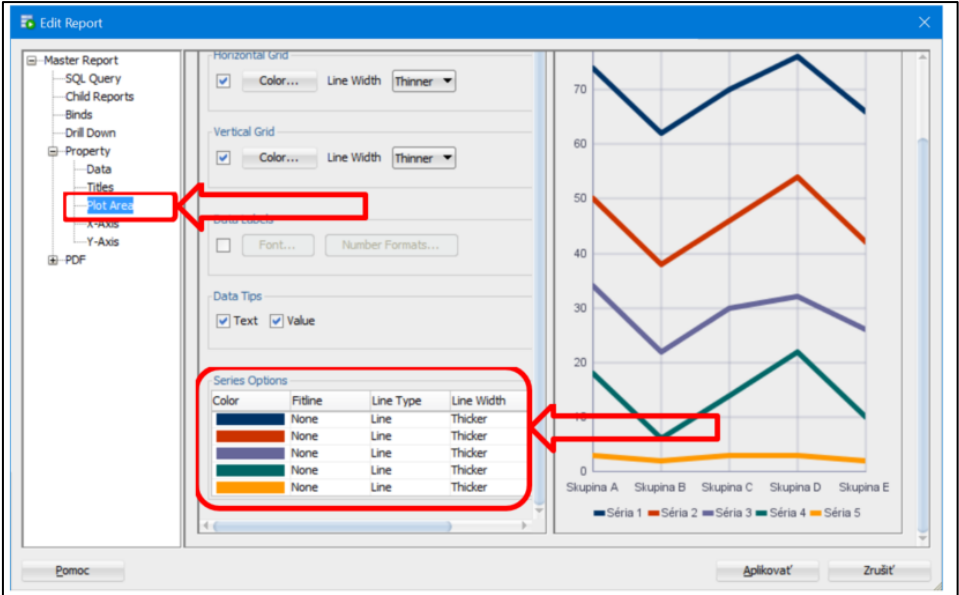


Fig. 15.65: Plot area property

10. Finish.

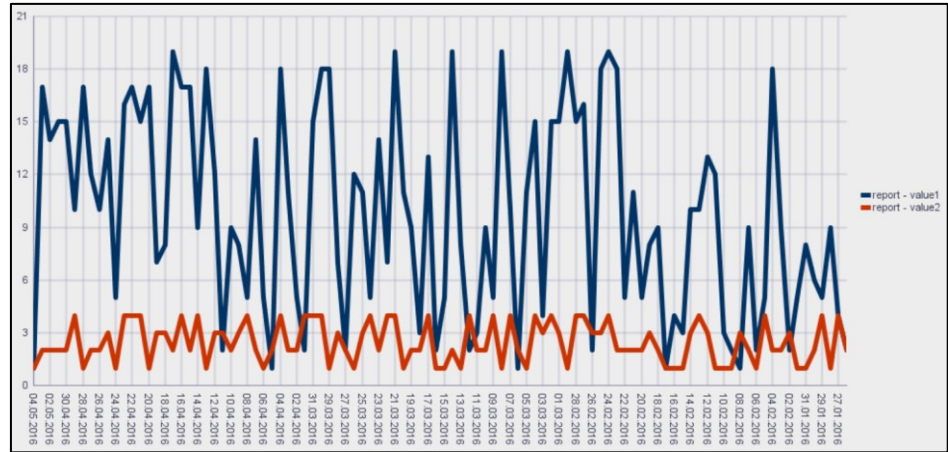


Fig. 15.66: Multiple graphs

11. Optionally, you can associate a separate y-axis scale with the changing *Chart Type* to **Line Dual Y**. These settings can be found in the *Property* branch.

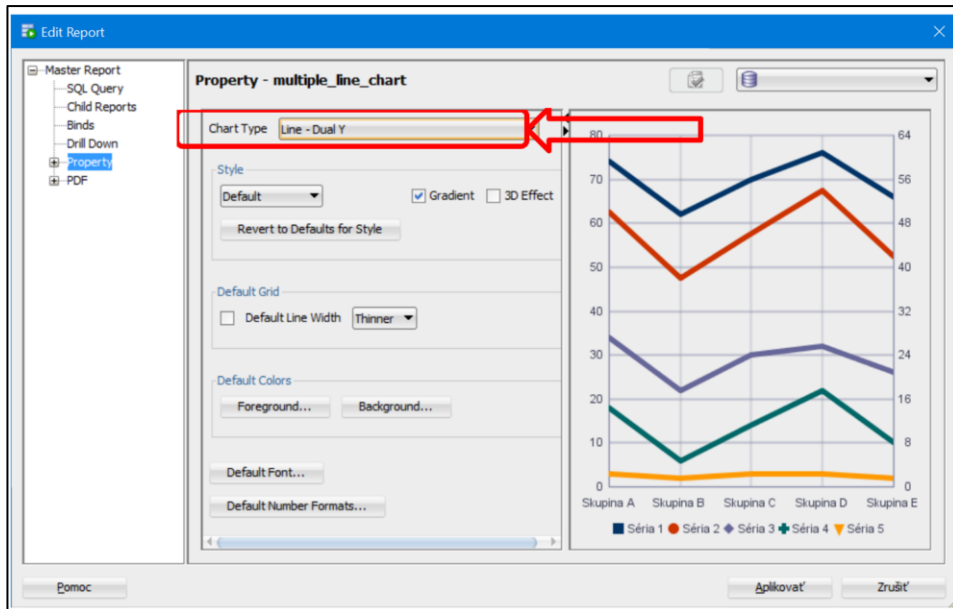


Fig. 15.67: Chart type

12. The result (y-axis scale for the *value* will be 0-21 (by default, it is set automatically based on provided data), and the y-axis scale for the *value2* will be 0-5).

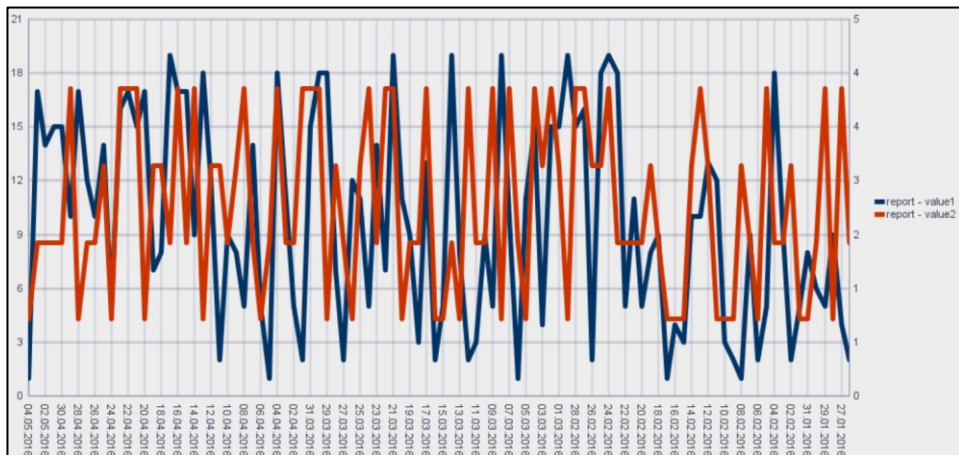


Fig. 15.68: Multiple graphs

15.9 Three-dimensional (3D) graph types

We can define *three-dimensional (3D) graph* reports. Let's have a simple example. We want to get the number of students for each class and study field. So, the *Select* statement will be like this:

```
select field_id, class, count(*), field_name
  from student join st_field using(field_id, specialization_id)
 group by field_id, class, field_name;
```

We want to get the text form (*field_name*), so join the *student* table with the *st_field* table.

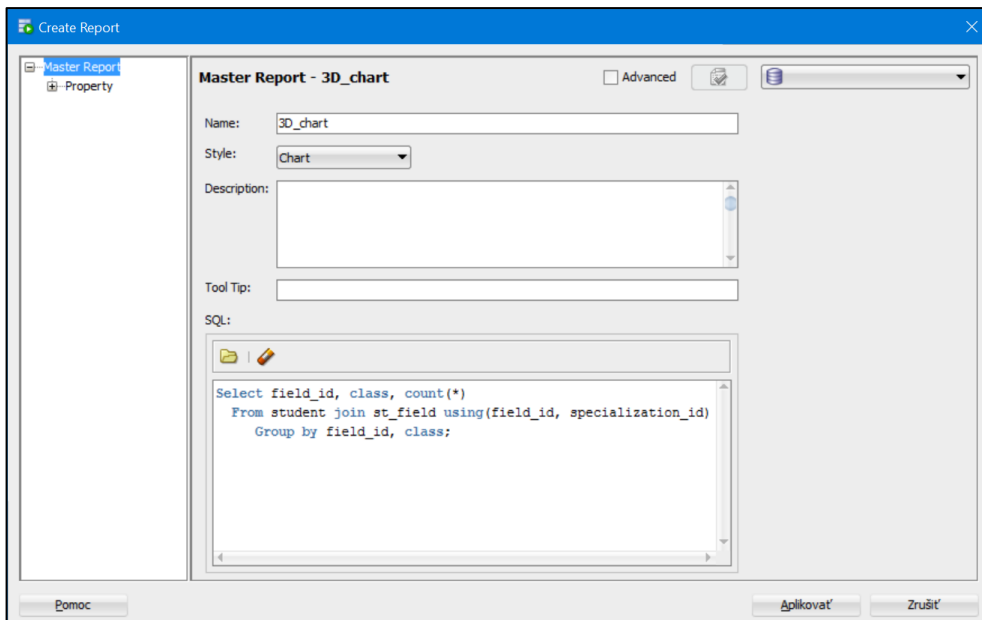


Fig. 15.69: Report definition

The definition consists of these steps:

1. Choose the **3D-bar** for the **Chart Type**.

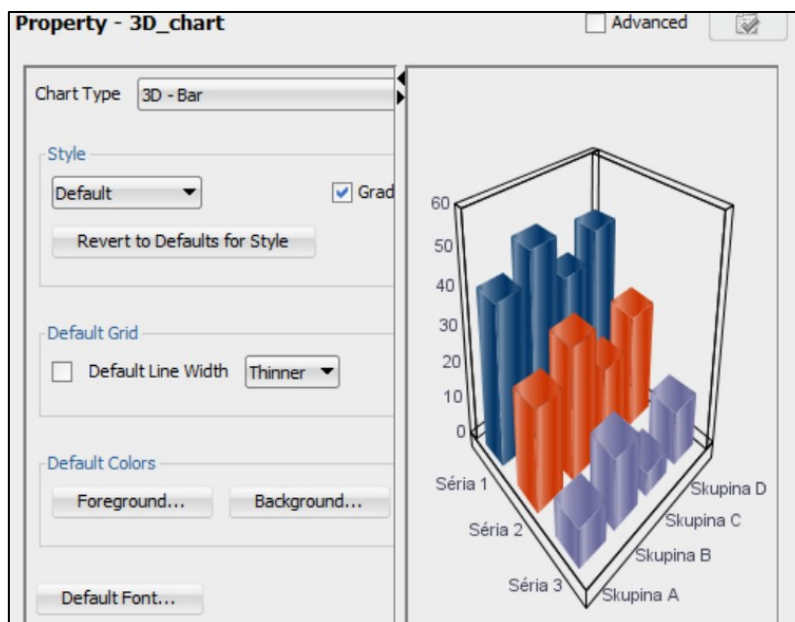


Fig. 15.70: 3D-bar graph definition

2. Map the attributes to the graph visualization definition:

Mapping

Fetch Column Names Clear Column Names

Group	Series	Value
"CLASS"	"FIELD_NAME"	"COUNT(*)"

+ ×

Fig. 15.71: Mapping

3. Result:

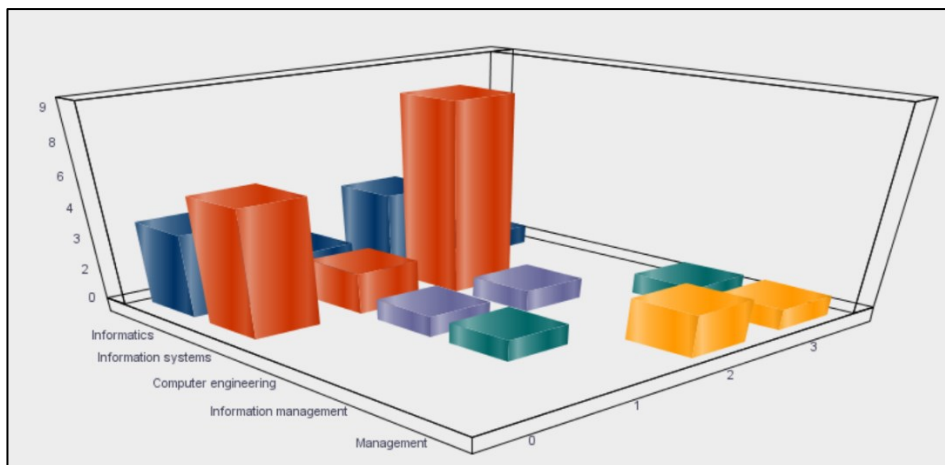


Fig. 15.72: 3D-bar Graph report

15.10 Binding multiple reports of various types

In the previous part of this section, we have experienced the principles of multiple reports binding. It was based on two or more table reports (one is master, the rest of them are children). In principle, it is possible to build child reports not only by using table form. We can use any style type. Now, we will show the chart reflecting the actual selection in the master table.

The master report of the example will be based on *personal_data* and *student* information – *name*, *surname*, and *student_id*. Child report will be based on *student_id* and will reflect the number of registered subjects for each *school_year*.

Master report Select statement:

```
select name, surname, student_id
  from personal_data join student using(personal_id);
```

Master report *Style* should be set to *Table*:



Fig. 15.73: Setting style

Child report Select statement (it must be bonded):

```
select school_year, 'subjects registration for student: ' || student_id,  
       count(*)  
from study_subjects  
where student_id = :STUDENT_ID  
group by school_year, student_id;
```

Child report *Style* should be set to *Chart*. Such settings can be found using this tree path:
Master Report => Child Reports => *name_of_child*:

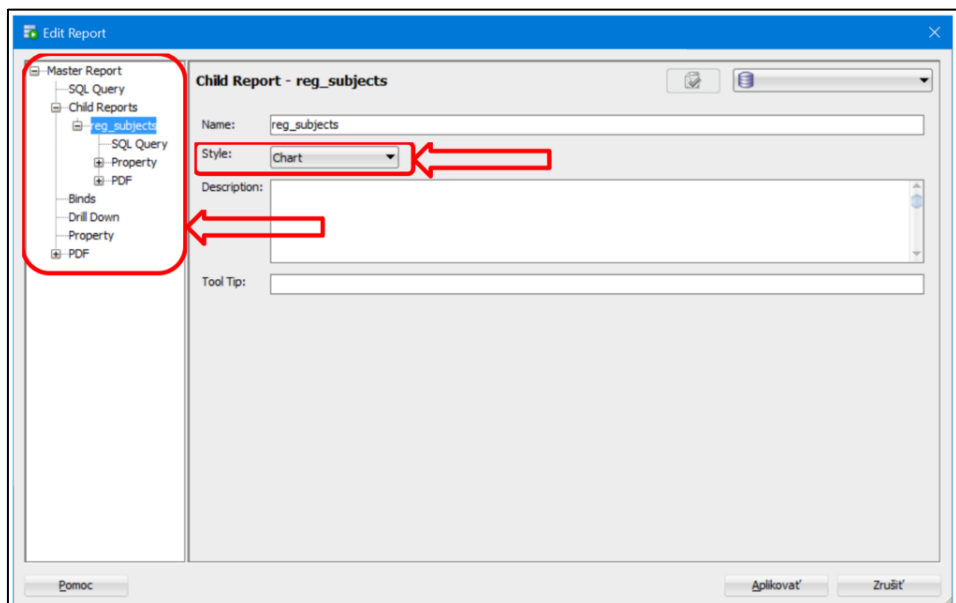


Fig. 15.74: Setting style

Chart type can be then changed in the *Property* section of the *Child Reports* node:

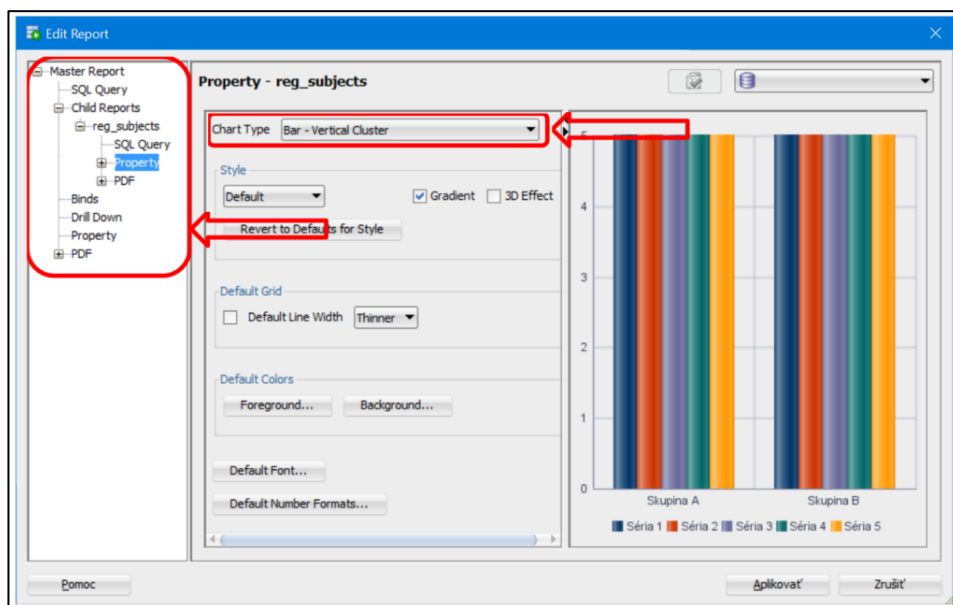


Fig. 15.75: Setting properties of the child report

The result of the binding is performed by clicking on the master report cell or row. The particular graph will be redrawn automatically.

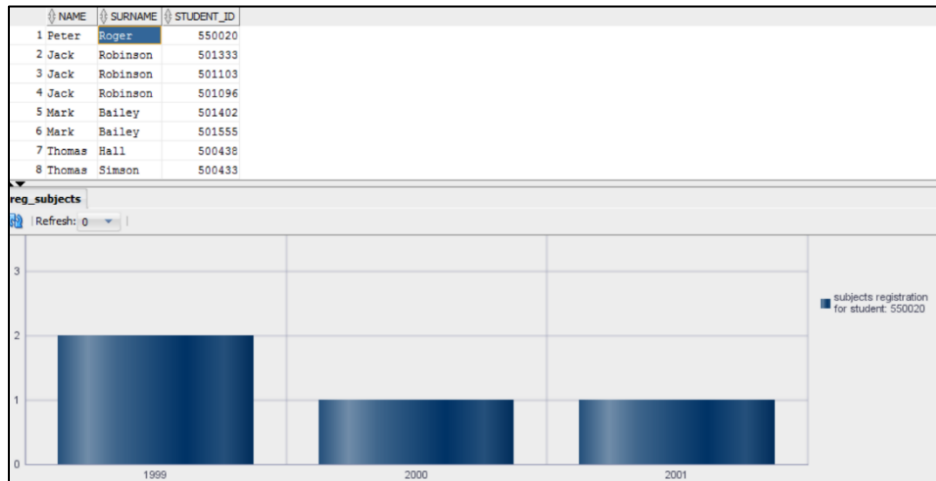


Fig. 15.76: Report output (binding)

15.11 Exports

Defined reports can be exported to multiple formats, which provide great techniques for presentations, evaluations, and post-processing. Moreover, the *Reports* module allows you to create various export formats for subsequent use in other database systems and applications.

For simplicity, exactness, and cleanness, we will deal with reports based on table data (*personal_data* and *student*) joined using *Left Outer Join*:

```
select personal_id, name, surname, student_id, class, status
from personal_data LEFT JOIN student using(personal_id);
```

We can store the results of the report physically in the file. So, create such a report, execute it and on the generated table report, right-click and select the **Export...** option.

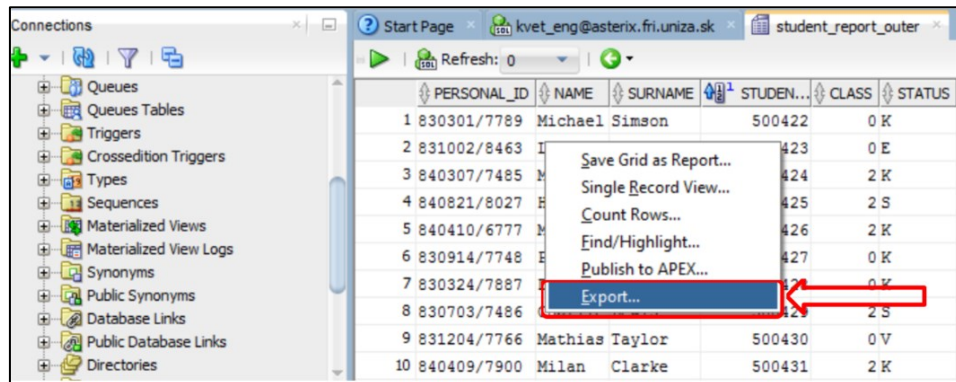


Fig. 15.77: Exporting the report

A new window will be created, allowing you to define the format in which the report should be exported. There are multiple types, which can be selected (combo box named *Format*). Therefore, we will describe only the most significant of them:

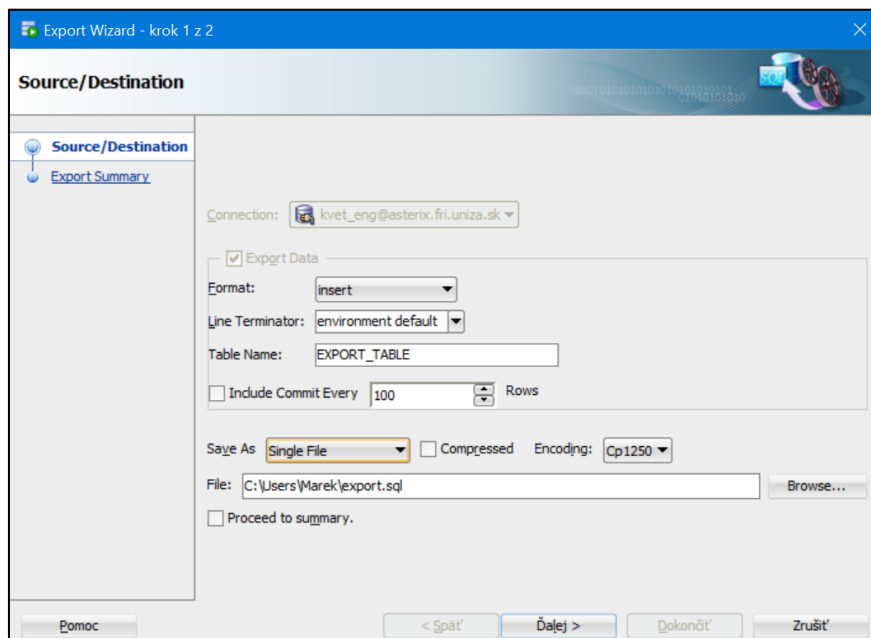


Fig. 15.78: Export wizard

15.11.1 CSV format

CSV (*Comma Separated Values*) file format has been proposed for data exchanges between various applications and systems. Such a file consists of a non-limited number of records (rows) delimited by the new line symbol. Each attribute (column) of the record is usually bounded by the comma (,), semicolon (;), or tab. Usually, each record has the same number of columns. Column values can be optionally enclosed by the quotation marks (" "). The main advantage of this format is based on allowing multiple system data transformations preparing data as input to another system. Notice that the *CSV* format is not strictly defined. There is no strict specification for it.

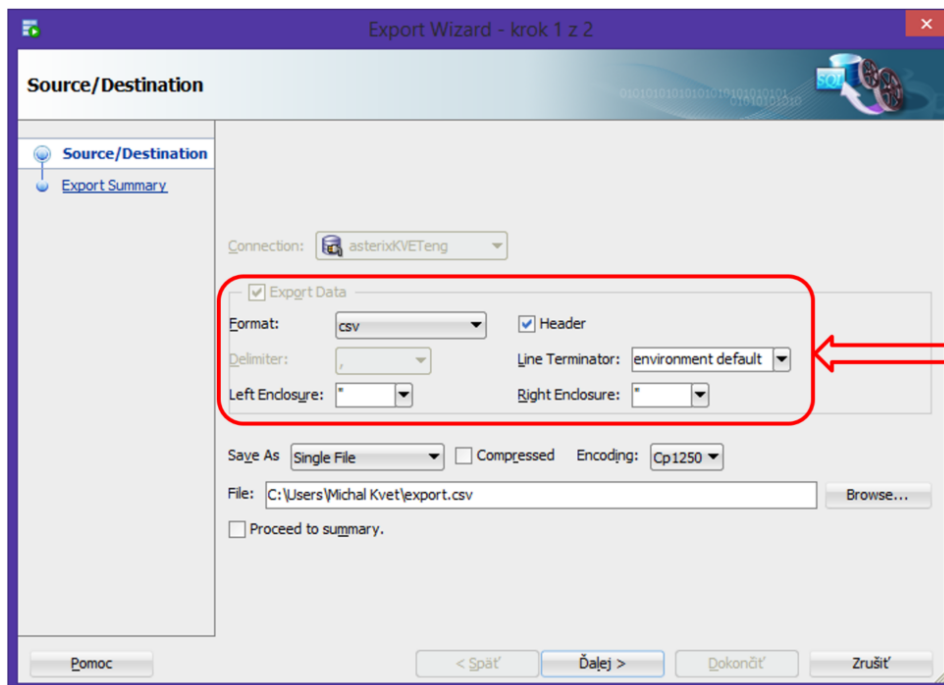


Fig. 15.79: CSV export

Optionally, you can select whether the header will be included or not (**Header** checkbox).

After selecting *CSV* in the **Format** combo box, also define the file path for the export and click on the **Next** button, which provides an export summary:

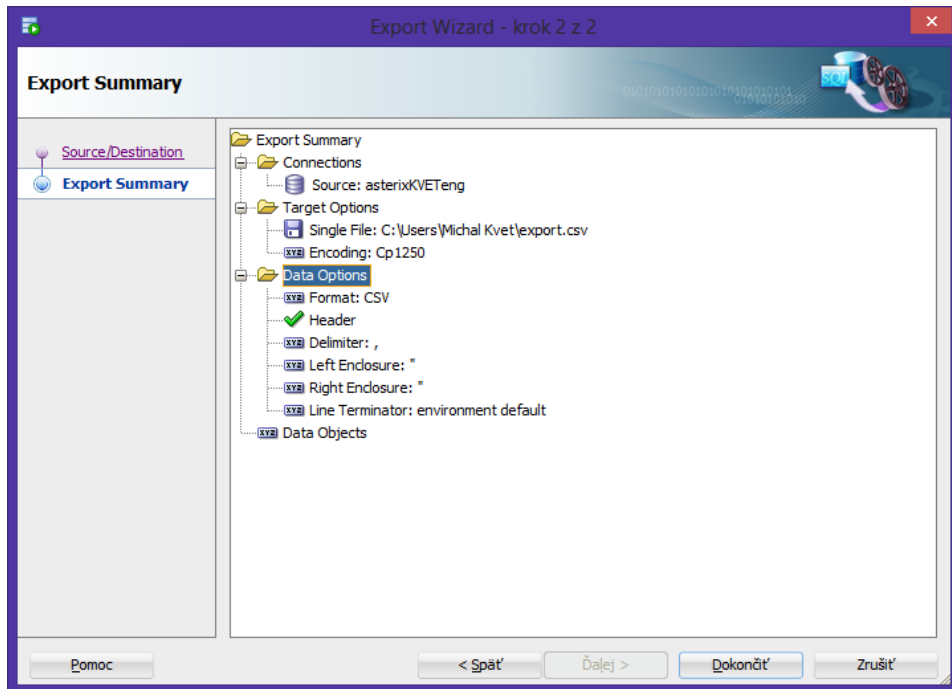


Fig. 15.80: Export wizard

Click the **Finish** button, and the file is automatically created in the destination folder.

```
"PERSONAL_ID","NAME","SURNAME","STUDENT_ID","CLASS","STATUS"
"601224/6526","Michael","Flower",,,
"601224/6537",,,,""
"740210/6525","Carol","Matlasko",,,
"740210/6536","Michael","Flower",,,
"781015/4431","Peter","Roger",550020,3,"S"
"791229/5431","Jack","Robinson",501333,1,"S"
```

In this case, *NULL* values are represented by empty strings.

15.11.2 Delimited format

Generalization of the *CSV* format provides a **Delimited** file format type. Possible types for the value of the property **Delimiter** are *comma (,)*, *pipe (|)*, *semicolon (;)*, *tab, whitespace, space*, etc. Values themselves can be enclosed using the following symbols: *quotation marks (")*, *apostrophes (')*, *parentheses (())*, *<, >*, *square parentheses ([])*, or no special symbol can be used (**none** option). A **line terminator** can be defined based on the environment to be used, like **<LF>** for Unix or by **<CR> <LF>** for Windows. For this example, select **Pipe (|)** for property **Delimited** and **none** for **Left** and **Right Enclosure**.

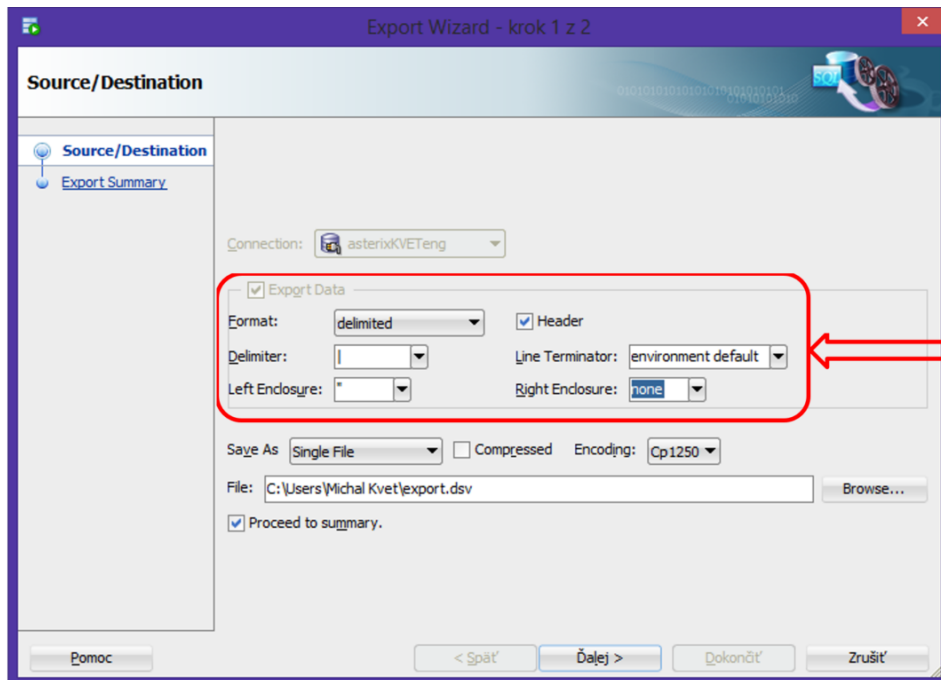


Fig. 15.81: Data export

Again, click the *Next* button and *Finish* button.
This is the input file for *SQL Loader*.

```
601224/6526|Michael|Flower|||
601224/6537|||||
740210/6525|Carol|Matiasko|||
740210/6536|Michael|Flower|||
781015/4431|Peter|Roger|550020|3|S
791229/5431|Jack|Robinson|501333|1|S
791229/5431|Jack|Robinson|501103|0|K
791229/5431|Jack|Robinson|501096|0|V
```

The defined format is well known for you, isn't it? Where has that format been used?

15.11.3 Text format

A special case of the *Delimited* format is *Text*. If the *Text* option of the *Format* combo box is selected, the used *Delimited* option is automatically selected to *Tab*. The user can define *Left* and *Right Enclosures* and *Line Terminator*. Values are aligned using tabs,

so each *starting* position of the value for the particular attribute is the same. The file format is **.tsv* and can be opened in any text editor (e.g., *Notepad* or *WordPad*):

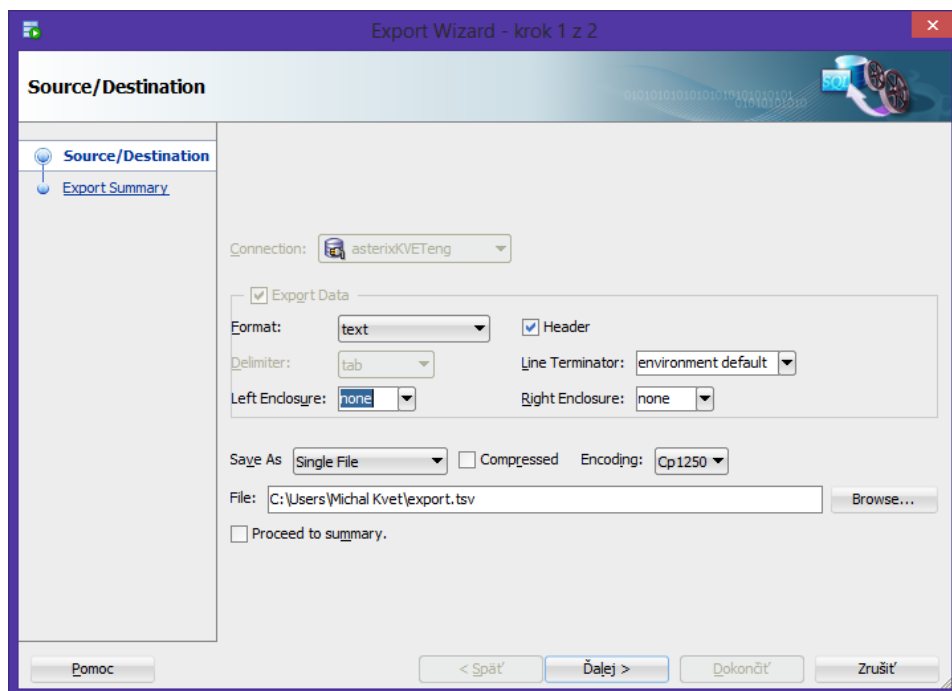


Fig. 15.82: Export wizard

Example of the data file output:

PERSONAL_ID	NAME	SURNAME	STUDENT_ID	CLASS	STATUS
791229/5431	Jack	Robinson	501333	1	S
791229/5431	Jack	Robinson	501103	0	K
791229/5431	Jack	Robinson	501096	0	V
800407/3522	Mark	Bailey	501402	2	S
800407/3522	Mark	Bailey	501402	1	S

Fig. 15.83: Output

15.11.4 Excel format

Reports connected to the *Export* module can also provide output format in *Excel*. Each column value is in a separate *Excel* cell. These values can be changed using a compatible application. Moreover, they can be automatically mapped to the database using *external table* functionality.

The following example will map the result to the **.xlsx* output format, which is characterized by *Excel version 2003* and newer releases. **.xls* file format is used for older versions than *Excel 2003*.

Export to *Excel* is also provided by the Wizard, *Excel 2003+* or *Excel 95-2003* should be used (**Format** combo box).

Two generated sheets will delimit the provided output file. The first will contain data themselves, and the second will include a statement, which generated such a report. The user

can name these sheets (*Data Worksheet Name* and *Query Worksheet Name* input box), or default names can be used (name *Export Worksheet* for *Data Worksheet Name* and name *SQL* for *Query Worksheet Name*).

When you deselect the option *True* in the *Query Worksheet Name* check box, only one sheet will be provided (with the report data).

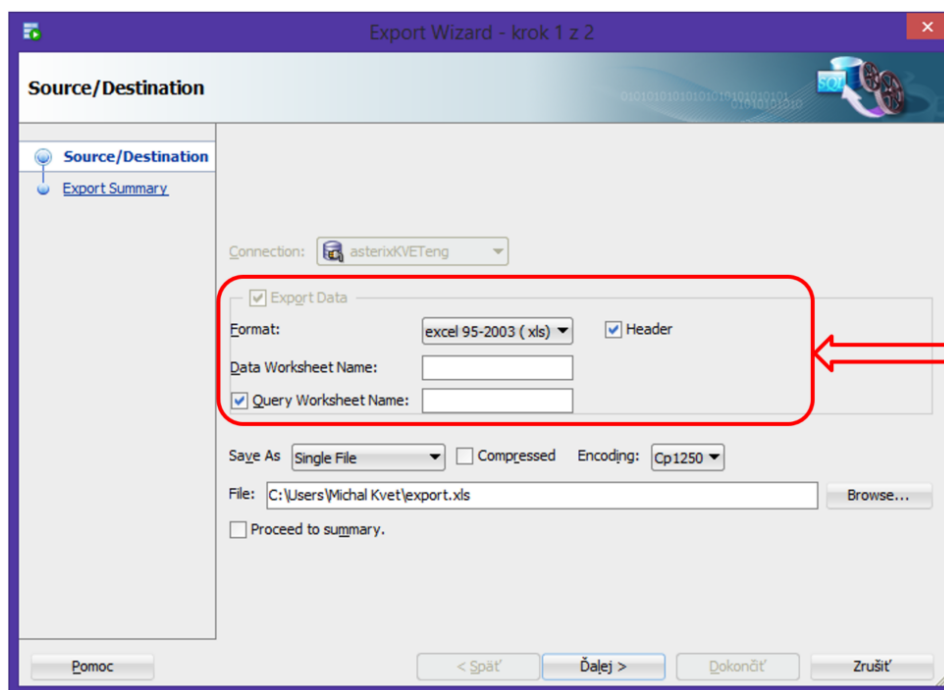


Fig. 15.84: Export wizard

If *NULL* values are defined for any attribute, an empty string will be included in the *Excel* file.

	A	B	C	D	E	F
1	PERSONAL_ID	NAME	SURNAME	STUDENT	CLASS	STATUS
2	601224/6526	Michael	Flower			
3	601224/6537					
4	740210/6525	Carol	Matiasako			
5	740210/6536	Michael	Flower			
6	781015/4431	Peter	Roger	550020	3	S
7	791229/5431	Jack	Robinson	501333	1	S
8	791229/5431	Jack	Robinson	501103	0	K
9	791229/5431	Jack	Robinson	501096	0	V
10	800407/3522	Mark	Bailey	501402	2	S

Fig. 15.85: Excel export

The second sheet with SQL statement forming export is following:

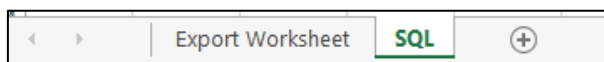


Fig. 15.86: SQL worksheet

	A	B	C	D	E	F	G	H	I	J	K	L
1												
2	select personal_id, name, surname, student_id, class, status from personal_data LEFT JOIN student using(personal_id)											
3												

Fig. 15.87: SQL worksheet

15.11.5 XML format

XML (*eXtensible Markup Language*) is the vital format used in many information systems for data transferring, sharing, or platform changing. It simplifies data availability and portability. It has been proposed by the consortium *W3C* and is based on previous markup approaches.

An **XML** document contains specific instructions called *tags*, *elements*, and *entities*. The resulting document is self-describing. Therefore, it is possible to use it to define data as well as their meaning (semantics).

Tags are not predefined. You must create your own ones, which will describe the relevant data.

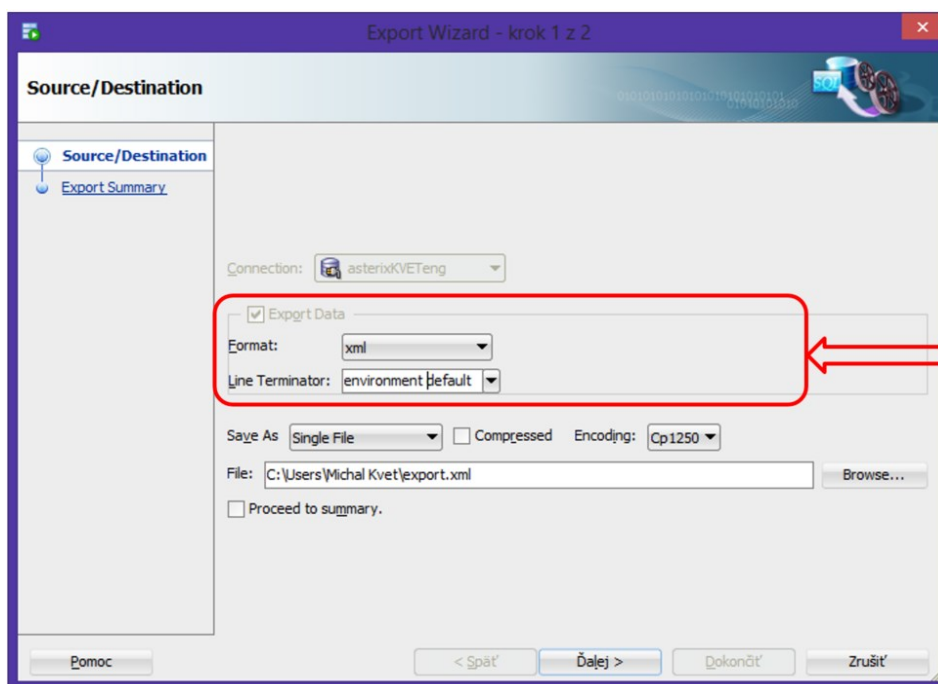


Fig. 15.88: XML wizard

```
<?xml version='1.0' encoding='Cp1250' ?>
<RESULTS>
  <ROW>
    <COLUMN NAME="PERSONAL_ID"><![CDATA[601224/6526]]></COLUMN>
    <COLUMN NAME="NAME"><![CDATA[Michael]]></COLUMN>
    <COLUMN NAME="SURNAME"><![CDATA[Flower]]></COLUMN>
    <COLUMN NAME="STUDENT_ID"><![CDATA[]]></COLUMN>
    <COLUMN NAME="CLASS"><![CDATA[]]></COLUMN>
    <COLUMN NAME="STATUS"><![CDATA[]]></COLUMN>
  </ROW>
```

```

<ROW>
  <COLUMN NAME="PERSONAL_ID"><![CDATA[601224/6537]]></COLUMN>
  <COLUMN NAME="NAME"><![CDATA[]]></COLUMN>
  <COLUMN NAME="SURNAME"><![CDATA[]]></COLUMN>
  <COLUMN NAME="STUDENT_ID"><![CDATA[]]></COLUMN>
  <COLUMN NAME="CLASS"><![CDATA[]]></COLUMN>
  <COLUMN NAME="STATUS"><![CDATA[]]></COLUMN>
</ROW>
</RESULTS>

```

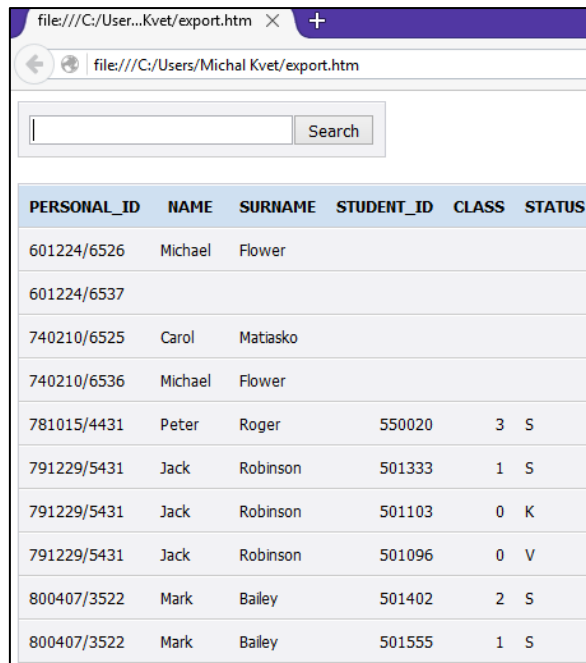
15.11.6 HTML format

Reports can be exported and stored in the **HTML** format. Thanks to that, websites can reference such page and provide report values in any form (table, charts, ...). **HMTL** reports can be generated by two methods, which also influence the characteristics. The first type is similar than the other exporting techniques described earlier. It can be provided in multiple formats like tables and charts, so right click on the report and choose **Export...** option. **Export Format** should be **HTML**. By this approach, it is possible to search in the result set using the search input box. However, be aware, generated **HTML** code is static, so it does not reflect any change in the database, which generated such **HTML**. Thus, there is no connection to the database, the **HTML** site itself embeds all the data:

```

<tr>
  <th>PERSONAL_ID</th>
  <th>NAME</th>
  <th>SURNAME</th>
  <th>STUDENT_ID</th>
  <th>CLASS</th>
  <th>STATUS</th>
</tr>
<tbody id="data">
<tr>
  <td>601224/6526</td>
  <td>Michael</td>
  <td>Flower</td>
  <td align="right">&nbsp;</td>
  <td align="right">&nbsp;</td>
  <td>&nbsp;</td>
</tr>

```

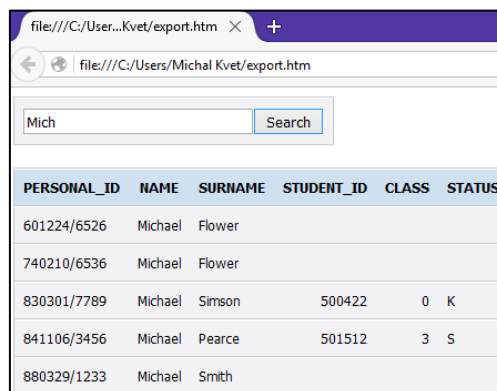



The screenshot shows a web browser window with the address bar displaying 'file:///C:/Users/Michal Kvet/export.htm'. Below the address bar is a search box with the text 'Search' and a magnifying glass icon. The main content area displays a table with the following data:

PERSONAL_ID	NAME	SURNAME	STUDENT_ID	CLASS	STATUS
601224/6526	Michael	Flower			
601224/6537					
740210/6525	Carol	Matiasco			
740210/6536	Michael	Flower			
781015/4431	Peter	Roger	550020	3	S
791229/5431	Jack	Robinson	501333	1	S
791229/5431	Jack	Robinson	501103	0	K
791229/5431	Jack	Robinson	501096	0	V
800407/3522	Mark	Bailey	501402	2	S
800407/3522	Mark	Bailey	501555	1	S

Fig. 15.89: HTML report

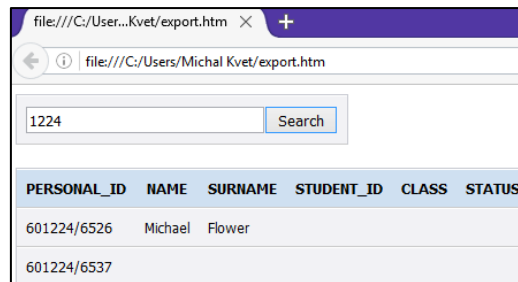
Searching using wildcards is available, and the *Search* box is added automatically. Thus, if you write 'Mich' in the search box, it will be automatically replaced by the '%Mich%' – first example. A similar principle also works for numerical values, e.g. ('1224' is replaced by the '%1224%' – second example).



The screenshot shows the same web browser window as Fig. 15.89, but the search box now contains the text 'Mich'. The table below shows the results of the search, filtered to show only students with the name 'Michael':

PERSONAL_ID	NAME	SURNAME	STUDENT_ID	CLASS	STATUS
601224/6526	Michael	Flower			
740210/6536	Michael	Flower			
830301/7789	Michael	Simson	500422	0	K
841106/3456	Michael	Pearce	501512	3	S
880329/1233	Michael	Smith			

Fig. 15.90: HTML report



PERSONAL_ID	NAME	SURNAME	STUDENT_ID	CLASS	STATUS
601224/6526	Michael	Flower			
601224/6537					

Fig. 15.91: *HTML report*

The second way, how to generate *HTML* export of the report can be processed by right-clicking on the report name in the **Reports** window by selecting the **HTML...** option.

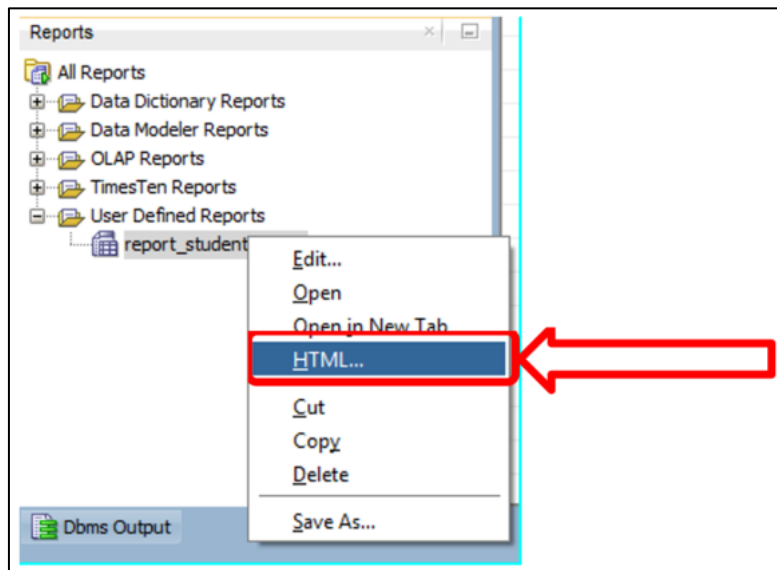


Fig. 15.92: *HTML report*

You can select the destination folder and the name of the file (**Destination HTML File**). Like all exports, defined file outputs are *STATIC*, so any change in the database IS NOT automatically reflected in the output. For the *PDF* files, it is clear. However, the same principle is also used for *HTML* generated files. Therefore, you can select whether the time stamp of the generation should (or should not) be part of the report header (checkbox **Include Time Stamp in Report Header**).

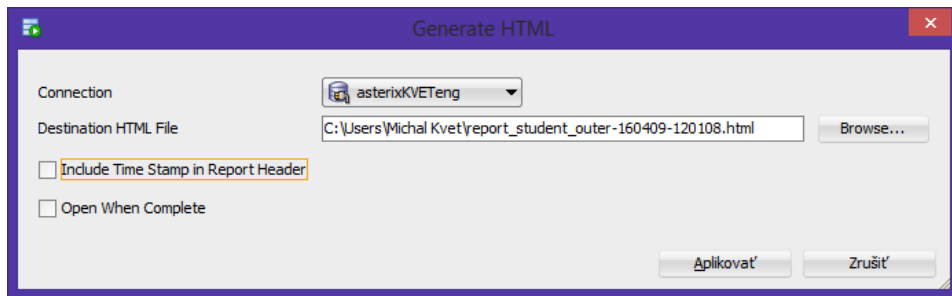


Fig. 15.93: *HTML report*

The output of this module is an *HTML* site, which can be published on the web. It will look like this:

PERSONAL_ID	NAME	SURNAME	STUDENT_ID	CLASS	STATUS
601224/6526	Michael	Flower	null	null	null
601224/6537	null	null	null	null	null
740210/6525	Carol	Matiasco	null	null	null
740210/6536	Michael	Flower	null	null	null
781015/4431	Peter	Roger	550020	3	S
791229/5431	Jack	Robinson	501333	1	S
791229/5431	Jack	Robinson	501103	0	K

Fig. 15.94: *HTML report*

Using this solution, you can select (set focus on) individual rows. Thanks to that, you can generate exports based on correlations (bindings). Thus, not only one report will be managed (master table), but records can also be associated with the child exports. So, add child report, which will contain all registered subjects of the particular (selected) student using the following *Select* statement:

```
select subject_id, name, school_year
  from study_subjects join subject using(subject_id)
 where student_id = :STUDENT_ID;
```

When you generate export using the second way, after selecting some row, also subject information of such student will be listed in the second report (child):

```
select school_year, 'subject registration count', count(*)
  from study_subjects
 where student_id = :STUDENT_ID
 group by school_year;
```

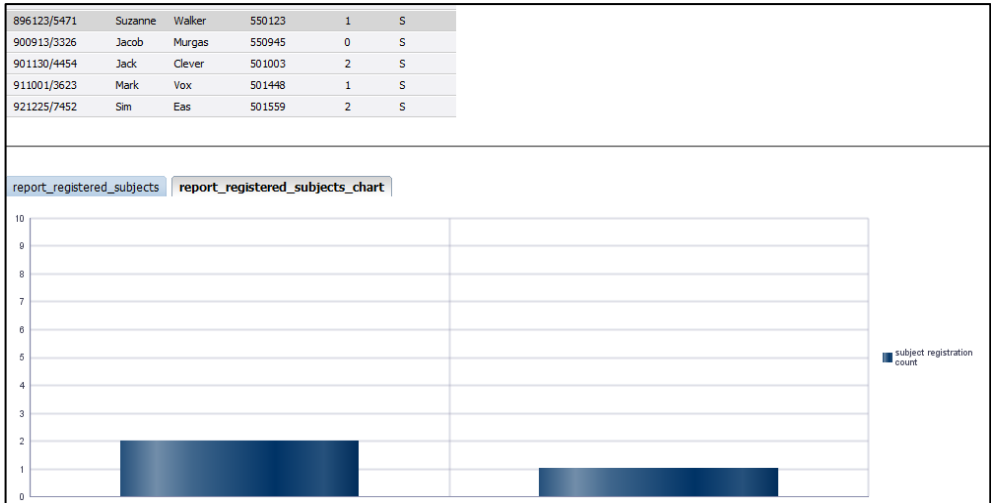


Fig. 15.95: HTML report

15.11.7 Exporting to PDF

The last category we will deal with is an export tool for generating PDF files. This functionality is based on the *Export...* option:

	PERSONAL_ID	NAME	SURNAME	STUDENT_ID	CLASS	STATUS
1	601224/6526	Michael	Flower	(null)	(null)	(null)
2	601224/6537	(null)	(null)	(null)	(null)	(null)
3	740210/6525	Carol	Matiasco	(null)	(null)	(null)
4	740210/6536	Michael	Flower	(null)	(null)	(null)
5	781015/4431	Peter	Roger	(null)	(null)	(null)
6	791229/5431	Jack	Robinson	(null)	(null)	(null)
7	791229/5431	Jack	Robinson	(null)	(null)	(null)
8	791229/5431	Jack	Robinson	(null)	(null)	(null)
9	800407/3522	Mark	Bailey	(null)	(null)	(null)
10	800407/3522	Mark	Bailey	(null)	(null)	(null)
11	810101/8079	Thomas	Hall	(null)	(null)	(null)

Fig. 15.96: Exporting to PDF

In the *Export window*, choose the *pdf* in the *Format* combo box. If selected, *Title*, *Subtitle*, and *Keywords* can be optionally defined.

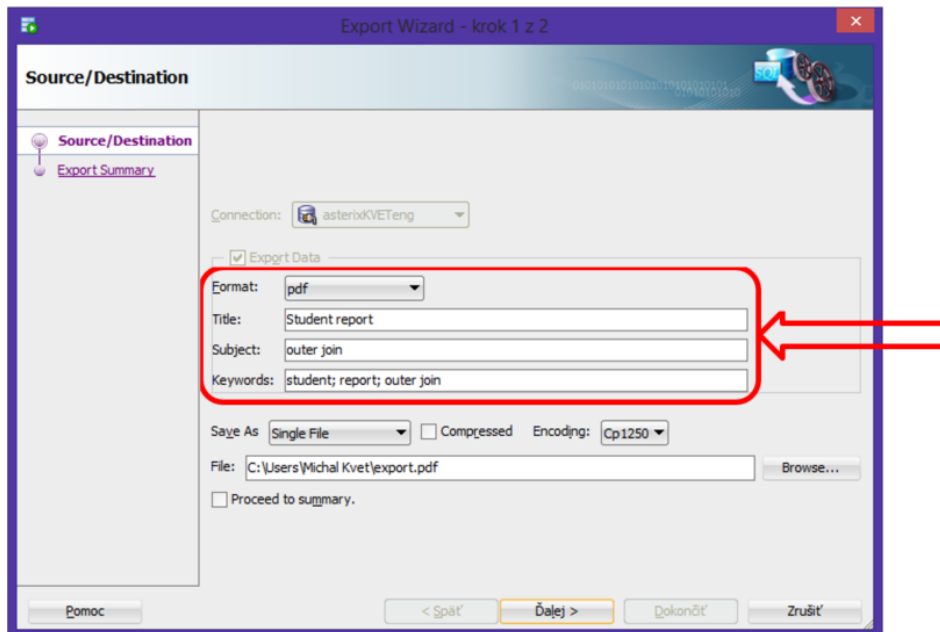


Fig. 15.97: Export to PDF wizard

You can optionally define also fonts, headers, footers, etc., to create *PDF* exact to your wishes. These settings can be changed after right-clicking on the report name and selecting the **Edit...** option.

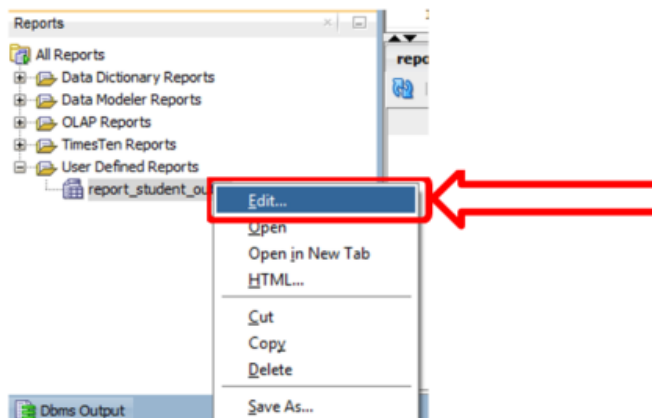


Fig. 15.98: Export to PDF wizard

Just the **PDF** branch part deals with such attributes influencing the design.

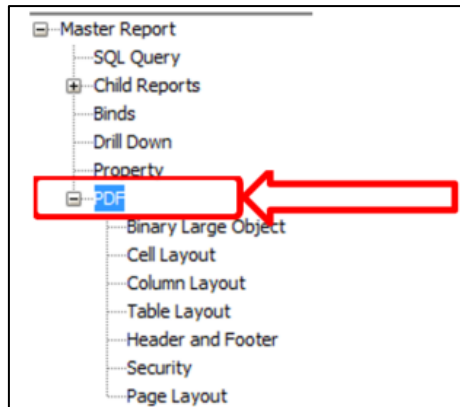


Fig. 15.99: Export to PDF wizard

Possible design changes can be defined and changed on the following nodes:

- Cell Layout,
- Column Layout,
- Table Layout,
- Header,
- Footer,
- Security,
- Page Layout.

Solution:

Report Date: 5/9/2016 4:54:21 PM					
Student Report					
PID	NAME	SURNAME	STUDENT ID	CLASS	STATUS
601224/6526	Michael	Flower	(null)	(null)	(null)
601224/6537	(null)	(null)	(null)	(null)	(null)
740210/6525	Carol	Matiasco	(null)	(null)	(null)
740210/6536	Michael	Flower	(null)	(null)	(null)
781015/4431	Peter	Roger	550020	3	S
791229/5431	Jack	Robinson	501333	1	S
791229/5431	Jack	Robinson	501103	0	K
791229/5431	Jack	Robinson	501096	0	V
800407/3522	Mark	Bailey	501402	2	S
800407/3522	Mark	Bailey	501555	1	S
810101/8079	Thomas	Hall	500438	2	K
820101/8452	Thomas	Simson	500433	0	E
830301/7789	Michael	Simson	500422	0	K
830324/7887	Daniel	Gomez	500428	0	K
830420/8088	Daniel	Green	500432	2	K
830514/5341	Viliam	Whittel	501567	0	E
830514/5341	Viliam	Whittel	501319	2	S
830703/7486	Charlie	Lewis	500429	2	S
830914/7748	Peter	Murphy	500427	0	K
831002/8463	Lucas	Powel	500423	0	E
831204/7766	Mathias	Taylor	500430	0	V

generated by: mk

page no: 1

Fig. 15.100: Export to PDF wizard

15.12 Script format (Insert)

SQL developer Report tool allows you to create (generate) the script for inserting data into the database – particular *Insert* statements are generated. There should be a defined name of the table (*Table Name* input box) and *Terminator*. Transaction *commit* can also be optionally generated after a specified number of data to be inserted.

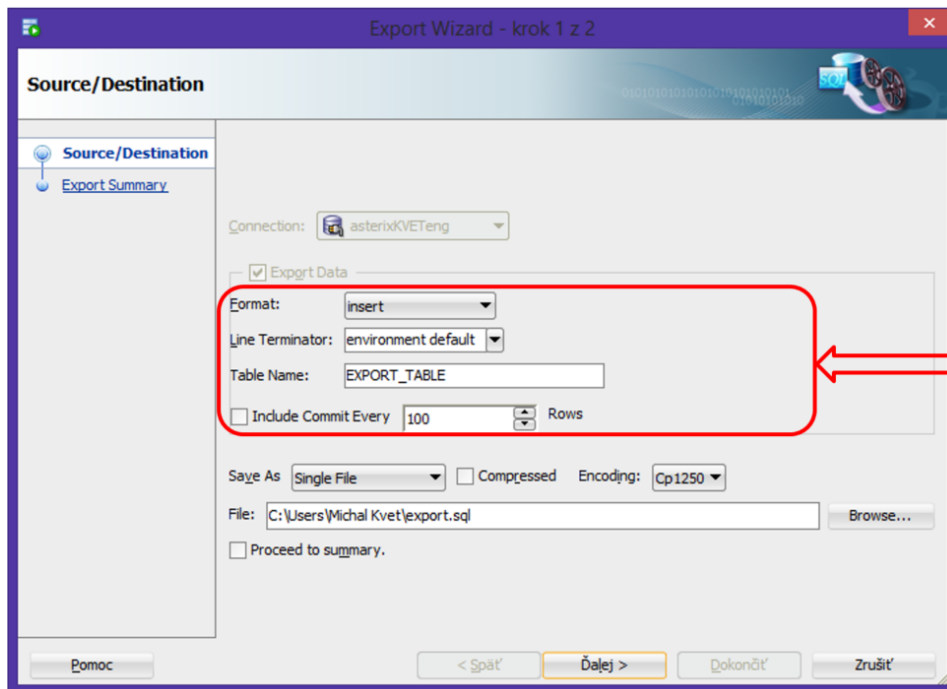


Fig. 15.101: Creating a script

The result (in our case, preview for 3 records) of the processing will provide the following data (example for inserting into *personal_data* table):

```
Insert into EXPORT_TABLE
(PERSONAL_ID, NAME, SURNAME, STREET, TOWN, ZIP, NATIONALITY)
values ('841106/3456', 'Michael', 'Pearce', 'Kamenna 27',
       'Banska Bystrica', '97401', 'SK');

Insert into EXPORT_TABLE
(PERSONAL_ID, NAME, SURNAME, STREET, TOWN, ZIP, NATIONALITY)
values ('840312/7845', 'Jack', 'Smith', 'Zelena 9',
       'Nove Mesto nad Vahom', '91501', 'SK');

Insert into EXPORT_TABLE
(PERSONAL_ID, NAME, SURNAME, STREET, TOWN, ZIP, NATIONALITY)
values ('860907/1259', 'John', 'Young', 'Slnečne namestie',
       'Komarno', '94501', 'SK');
```

As described in this lab, the *Report* tool provides sufficient power for user-friendly formatted output, responsible for data changes. This lab does not contain practice, however, be familiar with such technology and offered options.

Summary

You have reached the end of the textbook. Its goal is to teach you the principles of relational database systems in the *Oracle Cloud* environment.

In chapter 1, we have been dealing with the *Oracle Cloud Infrastructure* - the registration and database provisioning process. Oracle provides you with an *Always Free option* suitable for studying and testing. By implementing real solutions, you can migrate the account to the paid option extending the opportunities and individual resources. Available database types are *autonomous*, so most administration activities are managed automatically without your intervention. Connection details have been discussed, covering the *SQL Developer* tool.

The basics and principles of the data retrieval process are divided into multiple chapters. Chapter 2 emphasizes the *Select*, *From*, *Where*, and *Order By* clauses. In addition, it covers the *Inner Join* and *set operations*. You have been navigated to the problem of a *Cartesian product*, caused mostly by improper joining in the case of using composite primary keys.

In chapter 3, data changing operations (*Insert*, *Update*, *Delete* statements) are discussed, summarizing all *data manipulation language (DML)* operations. All executed statements are part of the *transactions*. To make the change durable, it is necessary to approve the transaction by reaching *Commit*. The transaction itself ensures the *consistency* of the data. Thus, all *integrity constraints* must be passed during the approval process. Otherwise, the transaction is refused (*Rollback*).

Chapter 4 covers the *data modeling* principles in a theoretical and practical manner. Data models are created in the *Toad Data Modeler* environment. The selection was made based on its versatility across multiple database systems. It can generate models and scripts not only for DBS Oracle. Moreover, individual models and types are interchangeable. This chapter has driven you through the table definitions and relationship types, focusing on the *primary* and *foreign* keys. Note that *SQL Developer* has embedded data modeler for Oracle databases.

In chapter 5, you have been proposed *data definition language (DDL)*, covering *Create*, *Alter*, and *Drop* commands, which implicitly reach transaction approval (implicit *Commit*). You have been focused on *user* management, *table* and *relationship* definitions, and *indexes* as key elements ensuring the system's performance.

Robustness of the *data loading* process in terms of *SQL Loader*, *client*, and *server import* and *export* operations was covered in chapter 6. You learned the principles of data object sharing and *privilege management* in chapter 7.

The second part of the *data retrieval* process was discussed in chapter 8, focusing on the *Outer Join* operators and *aggregate functions* by creating the groups in *Group By* clause. You have been focused on the *relational algebra* operations manipulating multiple sets.

Chapter 9 gave you an overview of the *procedural extension of SQL (PL/SQL)*. Code blocks can be *anonymous*, executed only once, or the *named* notation can be used, allowing you to reference the methods later. The focus was done on the *syntax*, code primitives up to the *procedure*, and *function* definitions, optionally covered by the *packages*. The data retrieval process inside the block must produce values stored in the *variables*. As stated, *Select Into* type should reference one row precisely, while the *cursor* provides a general solution using the assignment loop. Code blocks were also discussed in chapter 10, dealing with the *triggers* associated with the individual operations or events.

Chapter 11 provides you with the reference knowledge of *relational integrity*, supervised by the *transactions*.

Stored Select statements using *View* references were offered in chapter 12.

Date and Time management (chapter 13) is a complex topic related to the individual elements, *time-zone* synchronization, regions, *NLS* parameters, etc. You also obtained the information about the existing functionality summary, focusing on the principles and limitations.

A complex data object overview can be got by the *system tables*. Their principles, structures, and type division were discussed in chapter 14. After learning it, you are responsible for obtaining any structural information from the database.

In the last chapter, we returned to the topic of *data retrieval*, shaped in the *report* format. We discussed various styles of reporting and output formats to provide you with robust power. We have been dealing with the exports, as well.

In conclusion, we would like to draw your attention to further study in database technology. *Oracle APEX (Application Express)* is a *low-code development platform* enabling you to create *robust, secure, and scalable* applications deployed in the *cloud*. *APEX applications* are *data-driven* and can be developed fast using pre-prepared components. All necessary features and tools are part of the cloud, so you need to log on there and enjoy the modules. More about the *APEX technology* can be found in:

- <https://beeapex.eu/>



- <https://apex.oracle.com/en/>



That's the end. Thank you very much for your interest in database technology. If would like to contact the authors, please use the email addresses in the following format:

<name> . <surname> @ uniza . sk

References

- [1] ALAPATI, S. – KIM, CH.: *Oracle Database 11g: New Features for DBAs and Developers (Expert's Voice in Oracle)*, Apress, 2014.
- [2] ATZENI, P.- CERI, S.- PARABOSCHI, S.- TORLONE, R.: *Database Systems – concepts, languages & architectures*, McGraw-Hill, England, 1999.
- [3] BEAULIEU, A.: *Learning SQL: Generate, Manipulate, and Retrieve Data*, O'Reilly Media, 2020.
- [4] BLOKDYK, G.: *Oracle Cloud Infrastructure A Complete Guide*, 5STARCOoks, 2019.
- [5] BRYLA, B.: *Oracle Database 12c The Complete Reference*, Oracle Press, 2013, ISBN - 978-0071801751.
- [6] CANNAN, S. J.: *SQL – The Standard Handbook*, McGraw-Hill, 1992.
- [7] CUMMING, A. – RUSSELL, G.: *SQL Hacks*, O'Reilly, 2007, ISBN-13: 978-0-596-52799-0.
- [8] DATE, C.J.: *Database in Depth*, O'Reilly, 2005, ISBN 0-596-10012-4.
- [9] DATE, C.J. – DARWEN H. – LORENTZOS N.: *Temporal Data & the Relational Model*, Morgan Kaufmann, 2002, ISBN – 9780080518718.
- [10] DATE, C.J. – DARWEN H. – LORENTZOS N.: *Time and Relational Theory – Temporal Databases in the Relational Model and SQL*, Morgan Kaufmann, 2014, ISBN – 9780128006313.
- [11] DUTKA A., HANSON H.: *Fundamentals of Data Normalization*, Addison Wesley, London, 1994.
- [12] FAROULT, S. – ROBSON, P.: *The Art of SQL*, O'Reilly, 2006, ISBN 0-596-00894-4.
- [13] FERNANDEZ, I.: *Beginning Oracle Database 12c Administration: From Novice to Professional*, Apress, 2015.
- [14] FEUERSTEIN, S.: *Oracle PL/SQL Best Practices: Write the Best PL/SQL Code of Your Life*, O'Reilly Media, 2007, ISBN - 978-0596514105.
- [15] FEUERSTEIN, S. – PŘIBYL, B.: *Oracle PL/SQL programming*, O'Reilly Media 5th, 2009, ISBN - 978-1449324452.
- [16] FINKELSTEIN S., SCHKOLNICK M., TIBERIO P.: *Physical Database Design for Relational Databases*, ACM Transactions on Database Systems Vol. 13, No. 1, 1988.
- [17] GAN, I.: *T-SQL Fundamentals*, Microsoft Press, 2016, ISBN - 978-1509302000.
- [18] GELLER, A. – SPENDOLINI, B.: *Oracle Application Express: Build Powerful Data-Centric Web Apps with APEX (Oracle Press)*, McGraw-Hill Education, 2017.
- [19] GENNICK J. – MISHRA S.: *Oracle SQL*Loader: The Definitive Guide*, O'Reilly Media, 2001, ISBN - 978-1565929487.
- [20] GORMAN, T. – JORGENSEN I. – CAFFREY M. – HAAN L.: *Beginning Oracle SQL: For Oracle Database 12c*, Apress, 2014, ISBN - 978-1430265566.
- [21] GREENWALD, R. – STACKOWIAK R. – STERN J.: *Oracle Essentials: Oracle Database 12c*, O'Reilly Media, 2013, ISBN - 978-1449343033.

- [22] GURRY M.: *Oracle SQL Tuning Pocket References*, O'Reilly, 2002, ISBN 0-596-0068-8.
- [23] HALPIN, T.: *Information Modeling and Relational Databases – From Conceptual Analysis to Logical Design*, Morgan Kaufmann, 2001, ISBN – 9781558606722.
- [24] HANSEN, K.: *Practical Oracle SQL: Mastering the Full Power of Oracle Database*, Apress, 2020.
- [25] HARRINGTON, J.: *Relational Database Design and Implementation*, Morgan Kaufmann, 2016, ISBN – 9780128043998.
- [26] HELLER, J.: *Pro Oracle SQL Development: Best Practices for Writing Advanced Queries*, Apress, 2019.
- [27] HERMANDEZ, M.: *Database Design for Mere Mortals: A Hands-On Guide to Relational Database Design*, Addison-Wesley Professional, 2013, ISBN - 978-0321884497.
- [28] CHEN, L.: *Query Processing and Optimization in Information-integration Systems*, Stanford University, 2001.
- [29] CHEN, P. P.: *The Entity-Relationship Model: Toward a Unified View of Data*, ACM Transactions on Database Systems, vol. 1, no. 1, pp. 9-36, 1976.
- [30] CHURCHER, C.: *Beginning SQL Queries: From Novice to Professional*, Apress, 2016, ISBN - 978-1484219546.
- [31] JAIN, A. – MAHAJAN, N.: *The Cloud DBA-Oracle: Managing Oracle Database in the Cloud*, Apress, 2017.
- [32] JAKÓBCZYK, M.: *Practical Oracle Cloud Infrastructure: Infrastructure as a Service, Autonomous Database, Managed Kubernetes, and Serverless*, Apress, 2020.
- [33] JOHNSTON, T.: *Bitemporal data*, Morgan Kaufmann, 2014, ISBN-9780124080676.
- [34] JOHNSTON, T. – WEIS, R.: *Managing Time in Relational Databases – How to Design, Update and Query Temporal Data*, Morgan Kaufmann, 2010, ISBN – 9780123750419.
- [35] JURIC, K.: *Oracle CX Cloud Suite: Deliver a seamless and personalized customer experience with the Oracle CX Suite*, Packt Publishing, 2019.
- [36] KUMAR, Y. – BASHA, N. et al.: *Oracle High Availability, Disaster Recovery, and Cloud Services: Explore RAC, Data Guard, and Cloud Technology*, Apress, 2019.
- [37] KUHN, D. – KYTE, T.: *Oracle Database Transactions and Locking Revealed: Building High Performance Through Concurrency*, Apress, 2020.
- [38] KUHN, D. – KYTE, T.: *Expert Oracle Database Architecture: Techniques and Solutions for High Performance and Productivity*, Apress, 2021.
- [39] KVET, M. – MATIAŠKO, K. – VESTENICKÝ, V. – ŠALGOVÁ, V.: *Rýchly vývoj dátových modelov a aplikácií v prostredí Oracle APEX*, EDIS UNIZA, 2020. ISBN: 978-80-554-1678-6.
- [40] KVET, M. – MATIAŠKO, K.: *Temporálne databázy*, EDIS UNIZA, 2020. ISBN: 978-80-554-1662-5.
- [41] KVET, M.: *Database index balancing strategy*, 29th conference of open innovations association FRUCT: Tampere, Finland 12-14 May 2021. ISBN: 978-952-69244-5-8.

- [42] KVET, M. – KRŠÁK, E. – MATIAŠKO, K.: *Locating and accessing large datasets using Flower Index Approach*, Concurrency and computation-practice and experience (Vol. 32, Issue 13). ISSN: 2305-7254.
- [43] KVET, M. – MATIAŠKO, K.: *Managing, locating and evaluating undefined values in relational databases*, Information technology and systems, Springer Nature 2021. ISBN: 978-3-030-68284-2.
- [44] KVET, M. – MATIAŠKO, K.: *Trigger performance characteristics in temporal environment*, SIMS 2016: second international conference on systems informatics, modelling and simulation: Riga, Latvia 1-3 June 2016. – Piscataway: IEEE, 2016. – ISBN-978-1-5090-2693-7.
- [45] KVET, M. – MATIAŠKO, K.: *Temporal transaction integrity constraints management*, Cluster Computing, Springer, 2017, ISSN-13867857.
- [46] LACKO, L.: *Oracle. Správa, programování a použití databázového systému.*, Computer Press, a.s., Brno, ISBN 978-80-251-1490-2, 2007.
- [47] LEWIS, J.: *Cost-Based Oracle Fundamentals*, Apress, US, 2005, ISBN: 1-59059-636-6.
- [48] LONEY, K. – THERIAULT, M.: *Oracle. Kompletní průvodce tvorbou, správou a údržbou databází*, Computer Press, Praha 2002, ISBN 80-7226-635-7.
- [49] MALCHER, M. – KUHN, D.: *Pro Oracle Database 18c Administration: Manage and Safeguard Your Organization's Data*, Apress, 2019.
- [50] MATIAŠKO, K. – KVET, M. – KVET, M.: *Databázové systémy – 1. diel*, EDIS UNIZA, 2018, ISBN: 978-80-554-14881.
- [51] MATIAŠKO, K. – KVET, M. – KVET, M.: *Databázové systémy – 2. diel*, EDIS UNIZA, 2018, ISBN: 978-80-554-1489-8.
- [52] MATIAŠKO, K. – VAJSOVÁ, M. – KVET, M.: *Pokročilé databázové systémy – 1. Diel – Umenie programovania a administrácie*, EDIS UNIZA, 2017, ISBN: 978-80-554-1311-2.
- [53] MATIAŠKO, K. – VAJSOVÁ, M. – KVET, M.: *Pokročilé databázové systémy – 1. Diel – Architektúra, programovanie s objektmi a XML*, EDIS UNIZA, 2017, ISBN: 978-80-554-1312-9.
- [54] MATIAŠKO, K. – KVET, M. – KVET, M.: *Practices for database systems*, EDIS UNIZA, 2017, ISBN: 978-80-554-1396-9.
- [55] MCLAUGHLIN, M.: *Oracle Database 12c PL/SQL Advanced Programming Techniques*, McGraw-Hill Education, 2014, ISBN - 978-0071835145.
- [56] MCLAUGHLIN, M.: *Oracle Database 11g PL/SQL Programming*, Oracle Press, 2008, ISBN: 978-0071494458.
- [57] MELTON, J. – SIMON A.: *Understanding Relational Language Components*, Morgan Kaufmann, 2001, ISBN – 9781558604568.
- [58] MISHRA, S. – BEAULLIEU, A.: *Mastering Oracle SQL*, 2nd Edition, O'Reilly Media, 2004, ISBN - 978-0596006327.
- [59] MOLINA, H. – ULLMAN, J. – WIDOM, J.: *Database Systems: The Complete Book*, Pearson, 2008, ISBN - 978-0131873254.
- [60] MOLINARO, A.: *SQL Cookbook: Query Solutions and Techniques for Database Developers*, O'Reilly Media, 2005, ISBN - 978-0596009762.
- [61] MOKKEN, R.: *Implementing Oracle Integration Cloud Service: Understand everything you need to know about Oracle's Integration Cloud Service and how to utilize it optimally for your business*, Packt Publishing, 2017.

- [62] MORTON, K. – OSBORNE K. – SANDS R. – SHAMSUDEEN R. – STILL J.: *Pro Oracle SQL (Expert's Voice in Oracle)*, Apress, 2013, ISBN - 978-1430262206.
- [63] MUSTAFA, O. – LOCKARD, R.: *Oracle Database Application Security: With Oracle Internet Directory, Oracle Access Manager, and Oracle Identity Manager*, Apress, 2019.
- [64] NIEMEC R.: *Oracle Database 12c Release 2 Performance Tuning Tips & Techniques*, Oracle Press, 2017, ISBN - 978-1259589683.
- [65] O'HEARN, S.: *OCE Oracle Database SQL Certified Expert Exam Guide*, Oracle Press, McGraw-Hill Education, 2009, ISBN - 978-0071614214.
- [66] OZSU, M. T. – VALDURIEZ, P.: *Principles of Distributed Database Systems*, Prentice Hall, Englewood Cliffs, N. J., 1999.
- [67] PNG, A. – DEMANCHE, L.: *Getting Started with Oracle Cloud Free Tier: Create Modern Web Applications Using Always Free Resources*, Apress, 2020.
- [68] POKORNÝ, J.: *Databázové systémy*, CVUT, 2013.
- [69] PRICE, J.: *Oracle Database 11g SQL*, Oracle Press, 2007.
- [70] RAMAKRISHNAN, R. – GEHRKE, J.: *Database Management Systems*, Mc Grow Hill, New York, 2000.
- [71] RAMKLASS, R.: *Oracle Cloud Infrastructure Architect Associate All-in-One Exam Guide (Exam 1Z0-1072)*, McGraw-Hill Education, 2020.
- [72] ROSENBLUM, M. – DELMOLINO, D. – CUNNINGHAM, L. – SHAMSUDEEN, R. – MCDONALD, C. – CAFFREY, M. – HARPER, S. – HOLM, T. – SANDS, R. – BERESNIEWICZ, J. – CRISCO, R. – BCHI, M. – BILLINGTON, A. – PETIT, S. – NANDA, A.: *Expert PL/SQL Practices: for Oracle Developers and DBAs*, Apress, 2011.
- [73] ROSENBLUM, M.: *Oracle PL/SQL Performance Tuning Tips & Techniques*, McGraw-Hill Education, 2014, ISBN - 978-0071824828.
- [74] SCIORE, E.: *Understanding Oracle APEX 20 Application Development: Think Like an Application Express Developer*, Apress, 2020.
- [75] SIMSION, G. – WITT G.: *Data Modeling Essentials*, Morgan Kaufmann, 2004, ISBN - 9780126445510.
- [76] TEOREY T. – LIGHTSTONE, S. – NADEAU, T. – JAGADISH, H.V.: *Database Modeling and Design – Logical Design*, Morgan Kaufmann, 2014, ISBN - 9780123820211.
- [77] VALDURIEZ, P. – GARDARIAN, G.: *Analysis and Comparison of Relational Database Systems*, Addison Wesley, 1989.
- [78] VESTERLI, S.: *Oracle Visual Builder Cloud Service Revealed: Rapid Application Development for Web and Mobile*, Apress, 2019.
- [79] VIESCAS, J. – STEELE, D. – CLOTHIER B.: *Effective SQL: 61 Specific Ways to Write Better SQL (Effective Software Development Series)*, Addison-Wesley Professional, 2016, ISBN - 978-0134578897.
- [80] *Oracle documentation* (docs.oracle.com)

Oracle database	Autonomous database	APEX
		

Abbreviations

<i>Abbreviation</i>	<i>Meaning</i>
1NF	The first normal form
2NF	The second normal form
2PC	Two-Phase Commit
2PL	Two-Phase Locking
2VL	Two-Valued Logic
3NF	The third normal form
3VL	Three-Valued Logic
4NF	The fourth normal form
5NF	The fifth normal form
ABORT	Abnormal Termination
ACID	Atomicity. Consistency, Isolation, Durability
ACM	Association of Computer Machinery
ADT	Abstract Data Type
AK	Alternate Key
ANSI	American National Standards Institute
ANSI /SPARC	ANSI/Systems Planning and Requirements Committee
ASM	Automatic Storage Management
BCNF	Boyce Codd Normal Form
BCS	British Computer Society
BLOB	Binary Large Object
BNF	Backus/Nur form
CACM	Communications of the Association of Computer Machinery
CAD/CAM	Computer-Aided Design/Computer-Aided Manufacturing
CAD/CAM	Computer-Aided Design
CAM	Computer Aided Manufacturing
CASE	Computer Aided Software Engineering
CBO	Cost Base Optimization
CDO	Class Definition Object
CIM	Computer Integrated Manufacturing
CLOB	Character Large Object
CODASYL	Conference on Data Systems Languages
CPU	Central Processor Unit
CS	Cursor Stability
DB2	Database system of the IBM company
DBA	DataBase Administrator
DBCA	Database Configuration Assistant
DBMS	Database Management System
DBS	Database System
DBTG	Database Task Group
DC	Data Communication
DDB	Distributed Database
DDBS	Distributed Database System
DDL	Data Definition Language
DES	Data Encryption Standard

Abbreviation	Meaning
DKNF	Domain-Key Normal Form
DM	Data Model
DML	Data Manipulation Language
DOC	Document Object Model
DSL	Data Sublanguage
DTD	Document Type Definition
EJB	Enterprise JavaBeans
EM	Enterprise Manager
ER	Entity Relation
ERA	Entity Relation Attribute
FD	Functional Dependency
FIFO	First In First Out approach
FK	Foreign Key
HTML	HyperText Markup Language
I/O	Input /Output
IC	Integrity constraints
ID	Identification
IDMS	Integrated Database Management System
IMS	Information Management System
Informix	Database system type
Ingres	Database system type
I/O	Input/Output
IO	Integrity constraint
IS	Information system
IS	Intent Share
ISAM	Index Sequence Method
ISO	International Organization for Standardization
IT	Information technologies
IX	Intent exclusive
JD	Join Dependence
JDBC	Java Database Connectivity
JSP	Java Server Pages
LAN	Local Area Network
LOB	Large Object
MVD	MultiValued Dependency
MySQL	Database system type
NULL	Undefined value
O2	Database system type
ObjectStore	Database system type
ODL	Object Definition Language
ODMG	Object Database Management Group
OID	Object ID
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
OMF	Oracle Managed Files
OML	Object Manipulation Language
OO	Object Oriented

Abbreviation	Meaning
OODB	Object-Oriented DataBase
OOPL	OO Programming Language
OUI	Oracle Universal Installer
OQL	Object Query Language
ORACLE	Database system type
ORDBS	Object Relation Database System
OSI	Open System Interconnection
OSQL	Object SQL
PGA	Process Global Area
PJ/NF	Project/Join normal form
PK	Primary Key
PL/SQL	Procedural Language/Structured Query Language
Postgres	Database system type
Progress	Database system type
QBE	Query By Example
QMF	Query Management Facility
QUEL	Query Language
RA	Relational algebra
RAC	Real Application Cluster
RAID	Redundant Array of Inexpensive Disk
RBO	Rule-Based Optimization
RDB	Relational Database
RDBS	Relational Database System
RF	Reduction Factor
RI	Relational integrity
RID	Record ID
RMAN	Recovery Manager
ROWID	Pointer to the physical row
RPC	Remote Procedure Call
RR	Repeatable Read
SGML	Standard Markup Language
SIGMOD	Special Interest Group on Management of Data
SGA	System Global Area
SID	Oracle System Identifier
SIX	Shared Intent Exclusive
SQL	Structured Query Language
SRBD	DBMS
TCP/IP	Transmission Control protocol/Internet Protocol
UNK	UNKNOWN
VLDB	Very Large Data Base
VSAM	Virtual Storage Access Method
WAL	Write Ahead Log
WAN	Wide Area Network
WFF	Well-Formed Formula
WWW	World Wide Web
WYSIWYG	What You See Is What You Get
XML	Extended Markup Language

<i>Abbreviation</i>	<i>Meaning</i>
XPATH	Language for defining parts of an XML document
XQUERY	XML language for querying XML data
XSLT	EXtensible Stylesheet Language
XSD	XML Schema Definition

Index

A	
Abs	65
Access Method	172
Full Table Scan	172
Index Scan	172
Access Rights	292
Add_months	69
Afiedt.buf	48
Aggregate Function	225
Allen Relationship	374
Alias	
Column	58
Table	95
Alter	162
Alternative Key	333
Anonymous Block	265
Ascii	62
Assignment	254
Associative Entity	128
Attribute	122
Group	122
Multi-value	122
Non-atomic	122
Authid current_user	293
Avg	225
C	
Cartesian Product	88
Cascade	334
Case	76 259
Cd	146
Ceil	65
Coalesce	77
ColumnName macro	139
Column Alias	58
Comment	51

Commit	54
Compilation error	52
Concat	62
Conceptual modeling	117
Condition management	81
Constraint naming	160
Count	225
Cp	147
Create	156
Create Table	156
Cursor	286
Open	287
Fetch	287
Close	287
D	
Data Dictionary View	384
All	384
Dba	384
User	384
Dict_Columns	386
User_Arguments	393
User_Constraints	389
User_Objects	392
User_Sequences	396
User_Tab_Cols	387
User_Tables	47
User_Triggers	392
Data import	193
Data Model	114
Data Type	48 152
Date	48 353
Double	48
Float	48
Char	48
Integer	48

Interval	353
Interval Day To Second	379
Interval Year To Month	378
Lob	48
Long	48
Number	48
Timestamp	48
Varchar2	353
Dbms_Output Package	48
Disable	270
Enable	270
Get_Line	271
Get_Lines	271
New_Line	271
Put	271
Put_Line	272
Dbms_Random Package	428
Dbms_Stats Package	402
DCL	217
DDL	156
Decode	77
Default	160
	314
Delete	104
Desc	58
Determinant	126
Difference	244
Directory Oracle	43
	46
	185
	202
Directory OS	146
Disconnect	54
Distinct	94
DML	99
Drop	165
Dual Table	61
Dynamic Performance View	383

E	
Editor Joe	148
Entity	133
Entity-Relational (E-R) Conceptual Model	118
Exception	53
	275
	342
Execute	53
	269
Exists	90
Exit	54
	263
Exp	184
Expdp	207
Extract	69
F	
File management	147
Flashback	166
Floor	66
For	264
Foreign Key	105
	158
Function	52
	61
	266
G	
Grant	217
Object Privilege	219
System Privilege	217
Group By	227
H	
Having	233
Help	50
Host	50
CH	
Check Constraint	159
Chmod	147
I	
Identifying Key	119
If	255
Imp	184

Impdp	193 195
In	90
Index	167
B+ Tree	168
Bitmap	171
Function-Based	170
Index Organized Table	172
Reverse	169
Initcap	63
Initialization Block	302
Insert	99
Insert – Select	101
Insert – Values	100
Intersection	245
J	
Joe	148
Join	235 85
Anti	239
Full	238
Inner	235 85
Left	237
Natural	240
Right	238
Semi	239
L	
Last_Day	70
Length	64
Like	81
Linear notation	119
Loop	263
Lower	63
Ls	146
M	
Max	225
MaxValueTime	377
Min	225
Mkdir	147

Mod	67
Months_Between	70
Mv	147
N	
Next_Day	71
Nls_Parameters	358
Nls_Date_Format	361
Nls_Date_Language	360
Nls_Language	359 367
Nls_Territory	360
Null	80 254
Null Management	258
Nullif	78
Nullified	334
Nvl	78
Nvl2	78
O	
Order By	83
Overloading	301
P	
Package	296
Body	298
Specification	297
Password	154
Period	374
Personal_id structure	61 361
Primary Key	106 157 322 332
Privilege Database	217
Privilege OS	147
Privilege Directory	147
Procedure	52 266
Projection	58
Pwd	146

R	
Recycle Bin	166
Relational Algebra	240
Relational Integrity	331
Column	338
Domain	339
Entity	332
Referential	333
User	338
Relationship	141
Recursive	131 246
Cardinality – 1:1	124
Cardinality – 1:N	125
Cardinality – M:N	125
Identifying	123
Non-Identifying	123
Report	399
Binding	414
Csv Format	440
Delimited Format	441
Excel Format	443
Export	438
Graph	422
Hidden Column	413
HTML format	446
Child	414
Mapping	431
Master	414
PDF Export	450
Script Format	453
Table	404
Text Format	442
XML Format	445
Restricted	334
Revoke	220
Rmdir	147
Role	223
Rollback	54

Round	66
Rowid	167
S	
Select	57
Select Into	285
Selection	58
Sequence	323 396
Currval	324
Nextval	324
Set Operator	90
Set Serveroutput On	54 256
Show User	52
Spool	49
SQL Developer	38
SQL Loader	175
Sqldr	177
Sqlplus	41
SQL script generating	142
Start	51
Substr	64
Sum	225
Superkey	333
Syntax	55
Sys_Context	79
Sysdate	68
System Analysis	113
System Design	114
Systimestamp	68
T	
Table Management	133 155
Table Renaming	164
Technical Design	114
To_Date	75 67
To_Char	73
To_Number	75
To_Timestamp	75

Toad Modeler	121 131
Transaction	107
Trigger	307
DDL	327
Event	329
New	308
Old	308
Row	308
Statement	309
Trim	64
Trunc	66
U	
Union	241

Update	102
Upper	63
User	79
User Account	151 154
User Defined Domain	137
User Management	152
V	
Variable	254
View	341
Check Option	347
Read Only	349
W	
While	264

APPENDICES

APPENDIX A – MODEL STUDENT

Data model *Student* consists of nine tables (*Personal_data*, *Student*, *Study_subjects*, *St_field*, *Subject*, *Teacher*, *St_program*, *Subject_year*, and *Contact*). It deals with students (personal, contact, and student data) and their registrations to particular subjects supervised by teachers.

Table PERSONAL_DATA

This table contains information about the details of the person. Such table is connected to the table *Student* and *Contact*.

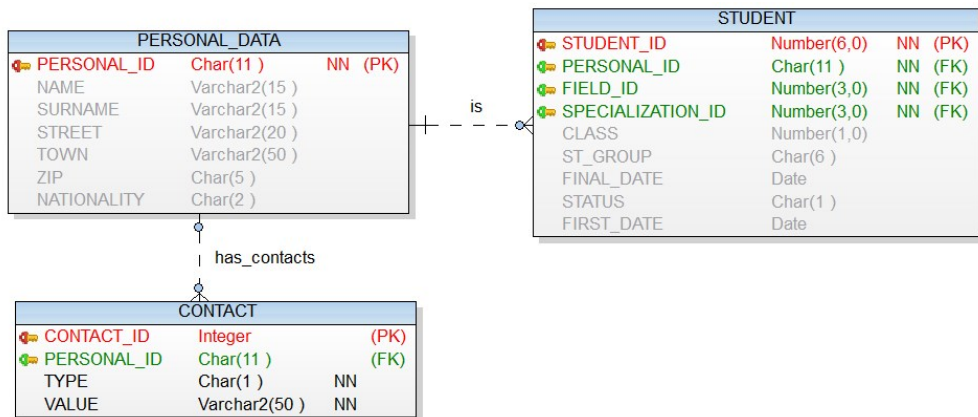


Fig. A.1: Personal_data submodel

Attributes

- ❖ **personal_id** – unique identification of the person.
 - attribute *personal_id* is the *primary key* of the table.
 - the data type is the string with exactly 11 characters. It follows this structure:
 - YYMMDD/XXXX where:
 - YY is two digits for the year of birth of the person,
 - MM is two digits for the month of birth of the person,
 - DD is two digits for the day of birth of the person (for women, 50 is added to the appropriate value),
 - “/” – separator,
 - XXXX are four digits for defining the order number of the person.
 - Notice that in a standard environment, the *personal_id* value can be divided by 11 without the remainder.
 - Example:
 - 890811/0134 is the identification for the person born on 11th August 1989 with order number 0134. It reflects the man.
 - 895811/0137 is the identification for the person born on 11th August 1989 with order number 0137. It reflects the woman.

- ❖ **name** – first name of the person.
 - value is the string with variable length limited to 15 characters.
 - Example:
 - Karol
- ❖ **surname** – family name of the person.
 - value is the string with variable length limited to 15 characters.
 - Example:
 - Matiaško
- ❖ **street** – street and house number of the person address.
 - value is the string with variable length limited to 20 characters.
 - Example:
 - Moyzesova 20
- ❖ **town** – the town of the address of the person.
 - value is the string with a variable limited to 50 characters.
 - Example:
 - Prievidza
- ❖ **zip** – ZIP code of the address of the person.
 - value is the string with a fixed length of 5 numerical characters. Although it expresses numerical value, it is stored as a string due to possible initial zeros.
 - Example:
 - 97251
 - 01001
- ❖ **nationality** – nationality abbreviation of the person.
 - value is the string with a fixed length of 2 characters.
 - Example:
 - SK ... it expresses “Slovakia”

Primary key

The primary key is attribute *personal_id*.

Foreign key

The table has no foreign keys.

SQL script for table creation

```
Create table personal_data
(
    personal_id    Char(11)          NOT NULL,
    name           Varchar2(15),
    surname        Varchar2(15),
    street         Varchar2(20),
    town           Varchar2(50),
    zip            Char(5),
    nationality     Char(2),
    primary key (personal_id)
);
```

Script for the relationship definition

None.

Table data example

Tab. A.1: Personal_data

<i>personal_id</i>	<i>name</i>	<i>surname</i>	<i>street</i>	<i>town</i>	<i>zip</i>	<i>nationality</i>
841106/3456	Michael	Pearce	Kamenna 27	Banska Bystrica	97401	SK
840312/7845	Jack	Smith	Zelena 9	Nove Mesto nad Vahom	91501	SK
871124/3578	Lucas	Austin	Dolna 12	Cadca	02201	SK
871203/5472	Tom	Moore	Prievoznicka	Ruzomberok	03401	SK
890310/2145	Arnas	Mitchell	Kosicka cesta	Michalovce	07101	SK
911001/3623	Mark	Vox	Tatranska 22	Poprad	05801	SK
901130/4454	Jack	Clever	Janka Borodaca 12	Prievidza	97101	SK
921225/7452	Sim	Eas	Kolarovce 12	Kolarovce	01401	SK
900913/3326	Jacob	Murgas	Namestie SNP 15	Banska Bystrica	97401	SK
870913/3326	Jacob	Murgas	Fatranska 13	Zilina	01008	SK
890608/4543	Jacob	Hoom	Orlove 16	Orlove	01701	SK
860103/2238	John	Young	Bratislavská cesta 2	Zilina	01001	SK
896123/5471	Suzanne	Walker	Pivovarska 14/536	Plzen	30100	SK
855122/8569	John	Pearce	Priecna ulica 35	Bytca	01401	SK
830914/7748	Peter	Murphy	147	Vysne Ruzbachy	06501	SK
840410/6777	Milan	Clarke	Mostna 19/1	Handlova	97251	SK

Table STUDENT

This table contains details of the student. This table is connected to the tables *Personal_data*, *Study_subjects*, and *St_field*.

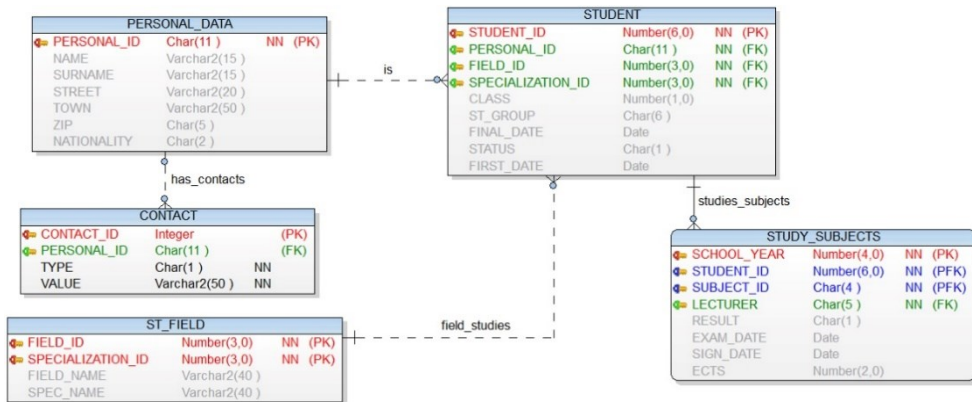


Fig. A.2: Student submodel

Attributes

- ❖ **student_id** – identification number of the student.
 - attribute *student_id* is the primary key of the table.
 - value is the integer with the maximal number composed of 6 digits.
 - Example:
 - 11111
- ❖ **personal_id** – identification key of the person.
 - this attribute is the foreign key to the *Personal_data* table.
 - the data type is the string with exactly 11 characters. It follows this structure:
 - YYMMDD/XXXX where:
 - YY is two digits for the year of born of the person,
 - MM is two digits for the month of born of the person,
 - DD is two digits for the day of born of the person (for women, 50 is added to the appropriate value),
 - “/” – separator,
 - XXXX are four digits for defining order number of the person.
 - Notice, that in a standard environment, the *personal_id* value can be divided by 11 without the remainder.
 - Example:
 - 890811/0134 is the identification for the person born on 11th August 1989 with order number 0134. It reflects the man.
 - 895811/0137 is the identification for the person born on 11th August 1989 with order number 0137. It reflects the woman.
- ❖ **field_id** – identification number of the study field.
 - the attribute is the part of the foreign key of the table, references *St_field* table.
 - value is the integer with the maximal number composed of 3 digits.
 - Example:
 - 111

- ❖ **specialization_id** – identification number of the study specialization of the study field.
 - the attribute is the part of the foreign key of the table, references *st_field* table.
 - value is the integer with the maximal number composed of 3 digits.
 - Example:
 - 111
- ❖ **class** – particular class of the student.
 - value is the integer with the maximal number composed of 1 digit.
 - Example:
 - 1
- ❖ **st_group** – value is the identifier for the study group.
 - value is the string with fixed length - 6 characters and following structure:
 - ABCDEF where:
 - A is one alphabet character for the faculty,
 - B is one alphabet character for the location - place of the campus,
 - C is one alphabet character for the field of the study,
 - D is one numerical character for the specialization of the study,
 - E is one numerical character for the class of the study,
 - F is one alphanumerical character for the order number of the group.
 - Example:
 - 5ZI02A ... is the identifier for the study group of the faculty with number **5** (Faculty of Management Science and Informatics) with a location in town **Z** (Zilina) in study field **I** (Informatics) in the **2**nd class and with the order preference number **A** (tenth group).
- ❖ **final_date** – value is the last day of the study of the student.
 - value has the *Date* data type.
 - Example:
 - 25.6.2015
 - NULL
- ❖ **status** – value representing the status of the study of the student.
 - value has the *Date* data type.
 - Example:
 - *S* = student (actual),
 - *E* = ended successfully,
 - *A* = aborted,
 - *I* = interrupted,
 - *X* = fired due to disciplinary commission decision.
- ❖ **first_date** – value the date of the beginning of the study.
 - value is the *Date* data type.
 - Example:
 - 1.9.2017

Primary key

The primary key is attribute *student_id*.

Foreign key

Attribute *personal_id* is the foreign key to the *Personal_data* table.

Composite attributes (*field_id*, *specialization_id*) form the foreign key to the *St_field* table.

SQL script for table creation

```
Create table student
(
    student_id          Number(6, 0)          NOT NULL,
    personal_id         Char(11)              NOT NULL,
    field_id            Number(3, 0)          NOT NULL,
    specialization_id   Number(3, 0)          NOT NULL,
    class               Number(1, 0),
    st_group            Char(6),
    final_date          Date,
    status              Char(1),
    first_date          Date,
    primary key (student_id)
);
```

Script for relationship definition

```
Alter table student
add foreign key (personal_id)
references personal_data (personal_id);

Alter table student
add foreign key (field_id, specialization_id)
references st_field (field_id, specialization_id);
```

Table data example

Tab. A.2: Student

<i>student_id</i>	<i>personal_id</i>	<i>field_id</i>	<i>speciali- zation_id</i>	<i>class</i>	<i>st_group</i>	<i>final_date</i>	<i>status</i>	<i>first_date</i>
550020	781015/4431	102	0	3	5ZM031	5.8.1999	S	(null)
501096	791229/5431	100	0	0	5ZI000	13.6.2000	V	20.12.2001
501103	791229/5431	100	0	0	5ZI000	13.7.2002	K	23.6.2006
501333	791229/5431	200	1	1	5ZSD11	5.9.2006	S	(null)
501555	800407/3522	101	0	1	5ZP012	10.10.2000	S	(null)
501402	800407/3522	102	0	2	5ZM023	15.7.2000	S	(null)
500428	830324/7887	200	3	0	5ZSN00	6.9.2005	K	15.6.2007
501567	830514/5341	100	0	0	5ZI000	19.7.2005	E	31.8.2006
501319	830514/5341	100	0	2	5ZIA21	5.9.2006	S	(null)
500429	830703/7486	200	3	2	5ZSN23	6.9.2005	S	31.8.2007
500427	830914/7748	200	3	0	5ZSN00	6.9.2005	K	15.6.2007

Table STUDY_SUBJECTS

This table contains details of courses studied by the student. This table is connected to the tables *Student*, *Teacher*, and *Subject*.

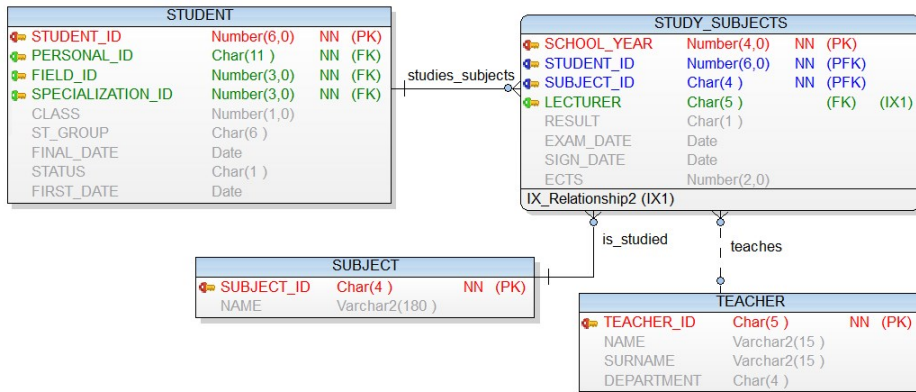


Fig. A.3: Study_subjects submodel

Attributes

- ❖ **school_year** – number of the actual school year.
 - attribute *school_year* is the part of the *primary key* of the table.
 - value is the integer with the maximal number composed of 4 digits.
 - Example:
 - 2016 – this number represents the school year 2016/2017
- ❖ **student_id** – identification number of the student.
 - attribute *student_id* is the part of the *primary key* of the table.
 - this attribute is the *foreign key* to the table *Student*.
 - value is the integer number with the maximal number composed of 6 digits.
 - Example:
 - 11111
- ❖ **subject_id** – identification number of the subject.
 - attribute *subject_id* is the part of the *primary key* of the table.
 - this attribute is the *foreign key* to the table *Subject*.
 - value is the string (see table *Subject*).
 - Example:
 - 5BI006
- ❖ **lecturer** – identification number of the teacher.
 - this attribute is the *foreign key* to the table *Teacher*.
 - value is the number (see table *Teacher*).
 - Example:
 - 33088

- ❖ **result** – value expressing the result of the exam.
 - value is the string with a fixed length – 1 character.
 - It can hold the following values:
 - *A – excellent results,*
 - *B – results above average,*
 - *C – results on average,*
 - *D – acceptable result,*
 - *E – results fulfilling the minimum requirements,*
 - *F – failed – further work required.*
- ❖ **exam_date** – value expresses the date of the exam.
 - Example:
 - 15.6.2017
- ❖ **sign_date** – value represents the date of the evaluation test.
 - Example:
 - 30.4.2017
- ❖ **ects** – value is the number of credits for the course.
 - Example:
 - 6

Primary key

The primary key is composite, formed by attributes *student_id*, *subject_id*, and *school_year*.

Foreign key

Attribute *student_id* is the *foreign key* to the *Student* table.

Attribute *subject_id* is the *foreign key* to the *Subject* table.

Attribute *lecturer* is the *foreign key* to the *Teacher* table.

SQL script for table creation

```
Create table study_subjects
(
    school_year    Number(4, 0)    NOT NULL,
    student_id     Number(6, 0)    NOT NULL,
    subject_id     Varchar2(30)    NOT NULL,
    lecturer       Char(5)         NOT NULL,
    result         Varchar2(1),
    exam_date      Date,
    sign_date      Date,
    ects           Number(2, 0),
    primary key (student_id, subject_id, school_year)
);
```

Script for relationship definition

```

Alter table study_subjects
  add foreign key (student_id)
    references student (student_id);

Alter table study_subjects
  add foreign key (subject_id)
    references subject (subject_id);

Alter table study_subjects
  add foreign key (lecturer)
    references teacher (teacher_id);

```

Table data example

Tab. A.3: Study_subjects

<i>school_year</i>	<i>student_id</i>	<i>subject_id</i>	<i>lecturer</i>	<i>result</i>	<i>exam_date</i>	<i>sign_date</i>	<i>ects</i>
2005	500424	II08	KI001	(null)	(null)	(null)	5
2005	500424	IN09	KI001	E	20.12.2005	8.2.2006	5
2005	500424	IP02	KI001	(null)	3.2.2006	(null)	6
2005	500424	IP03	KI001	(null)	22.6.2006	(null)	6
2006	500424	II15	KI001	F	(null)	(null)	5
2006	500424	II08	KI001	F	(null)	(null)	5
2006	500424	IP07	KI001	E	20.12.2006	17.1.2007	5
2007	500424	II08	KI001	D	5.5.2008	27.5.2008	5
2007	500424	IP05	KI001	(null)	23.5.2008	(null)	0
2007	500424	IPN3	KI001	(null)	22.5.2008	(null)	6

Table ST_FIELD

This table contains details of the study fields and study specializations offered by the faculty. This table is connected to the tables *Student* and *St_program*.

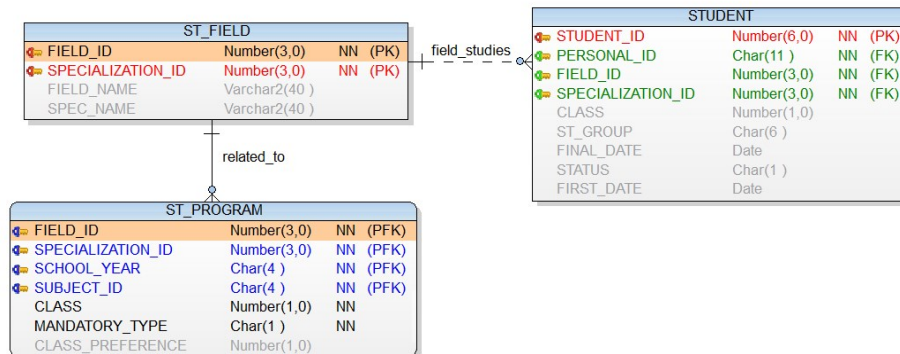


Fig. A.4: *St_field* submodel

Attributes

- ❖ **subject_id** – identification number of the subject.
 - The attribute is the *primary key* of the table.
 - value is the string composed of 4 characters.
 - Example:
 - BI06
- ❖ **name** – the name of the subject.
 - string value contained a maximally of 180 characters.
 - Example:
 - Database systems

Primary key

The primary key is the attribute *subject_id*.

Foreign key

This table has no foreign keys.

SQL script for table creation

```
Create table subject
(
    subject_id      Varchar2(6)      NOT NULL,
    name            Varchar2(180),
    primary key (subject_id)
);
```

Script for the relationship definition

None.

Table data example*Tab. A.4: St_field*

<i>field_id</i>	<i>specialization_id</i>	<i>field_name</i>	<i>spec_name</i>
100	0	Informatics	(null)
101	0	Computer engineering	(null)
102	0	Management	(null)
200	0	Information systems	(null)
200	1	Information systems	Decision support systems
200	2	Information systems	Applied informatics
200	3	Information systems	Information and communication systems
201	0	Information management	(null)
202	0	Computer engineering	(null)

Table SUBJECT

This table contains details of the subjects offered by the faculty. This table is connected to the tables *Study_subjects* and *Subject_year*.

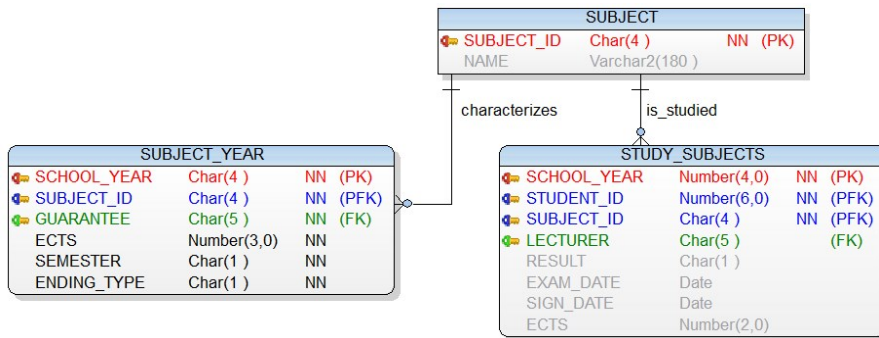


Fig. A.5: Subject submodel

Attributes

- ❖ **field_id** – this attribute is used as an identifier of the study field.
 - integer value composed maximally to 3 digits.
 - this attribute is part of the primary key of the table.
 - Example:
 - 1
- ❖ **specialization_id** – this attribute is used for the identification of the study specialization.
 - integer value composed maximally to 3 digits.
 - this attribute is part of the primary key of the table.
 - Example:
 - 2
- ❖ **field_name** – this attribute expresses the name of the study field.
 - string value composed maximally of 40 characters.
 - Example:
 - Information systems
- ❖ **spec_name** – this attribute represents the name of the study specialization.
 - string value contained a maximally of 40 characters.
 - Example:
 - Data processing

Primary key

The primary key is composite, formed by attributes *field_id* and *specialization_id*.

Foreign key

This table has no foreign keys.

SQL script for table creation

```
Create table st_field
(
    field_id                Number(3, 0)    NOT NULL,
    specialization_id       Number(3, 0)    NOT NULL,
    field_name              Varchar2(40),
    spec_name               Varchar2(40),
    primary key(field_id, specialization_id)
);
```

Script for the relationship definition

None.

Table data example

Tab. A.5: Subject

<i>subject_id</i>	<i>name</i>
BA20	Modern approximate methods
BI26	Object programming in Windows
BE16	Business management
BI22	Open source techniques
IE04	Taxes and budget
IH07	Signal processing 2
BH08	Automatic control theory 1
II12	Databases and knowledge discovery
BI06	Database systems - the best subject :)

Table TEACHER

This table contains details of the teachers. This table is connected to the tables *Study_subjects* and *Subject_year*.

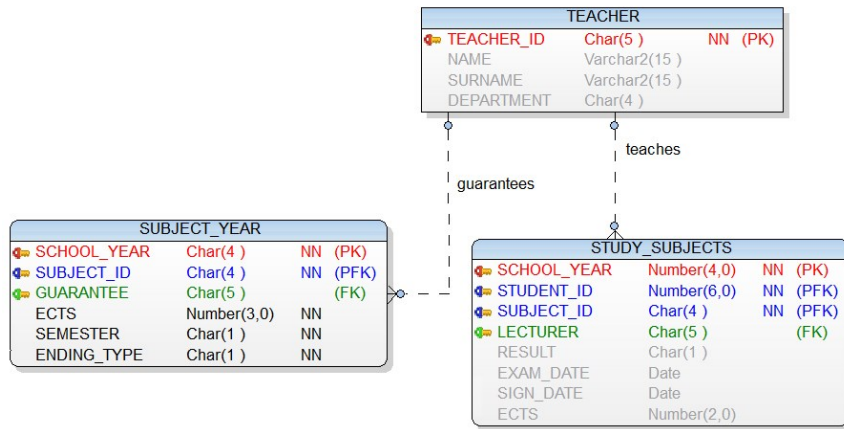


Fig. A.6: Teacher submodel

Attributes

- ❖ **teacher_id** – identification number of the teacher.
 - the attribute is the *primary key* of the table.
 - value is the string composed exactly of 5 characters.
 - Example:
 - GAR01
- ❖ **name** – first name of the teacher.
 - value is the string composed maximally of 15 characters.
 - Example:
 - Michal
- ❖ **surname** – the family name of the teacher.
 - value is the string composed maximally of 15 characters.
 - Example:
 - Kvet
- ❖ **department** – identification of the department of the teacher.
 - value is the string composed of exactly 4 characters.
 - Example:
 - KINF

Primary key

The primary key is the attribute *teacher_id*.

Foreign key

This table has no foreign keys.

SQL script for table creation

```
Create table teacher
(
    teacher_id          Char(5)          NOT NULL,
    name                Varchar2(15),
    surname              Varchar2(15),
    department          Char(4),
    primary key(teacher_id)
);
```

Script for the relationship definition

None.

Table data example

Tab. A.6: Teacher

<i>teacher_id</i>	<i>name</i>	<i>surname</i>	<i>department</i>
GAR01	Mark	Madrigal	Gar
KI001	Wiliam	Santos	DI
KMM02	Michael	Cloutier	DMM
KMM03	Carol	Poulin	DMM
KI002	Charlie	Polanco	DI
KI003	Rachel	Vargas	DI
KI005	Mathias	Fortin	DI
KTK02	Jacob	Demers	DTK
KDS04	Bill	Rosario	KTN

Table SUBJECT_YEAR

This table contains details of the courses offered by the faculty in the defined school year. This table is connected to the tables *Subject*, *Teacher*, and *St_program*.

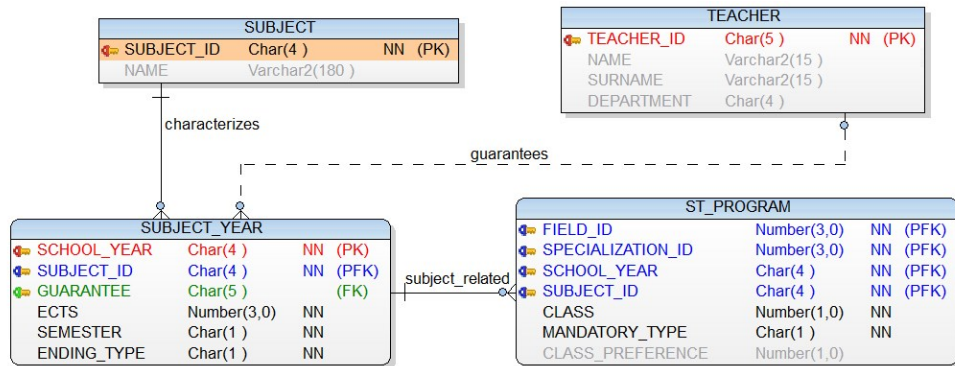


Fig. A.7: Subject_year submodel

Attributes

- ❖ **school_year** – number of the actual school year.
 - attribute *school_year* is part of the *primary key* of the table.
 - value is the integer with the maximal number composed of 4 digits.
 - Example:
 - 2017 – this number represents the school year 2017/2018
- ❖ **subject_id** – identification number of the subject.
 - attribute *subject_id* is part of the *primary key* of the table.
 - this attribute is the *foreign key* to the table *Subject*.
 - value is the string (see table *Subject*).
 - Example:
 - BI06
- ❖ **guarantee** – identification number of the teacher, expresses teacher responsible for the subject.
 - this attribute is the *foreign key* to the table *Teacher*.
 - value is the number (see table *Teacher*).
 - Example:
 - 33088
- ❖ **ects** – attribute defining the number of credits for the course.
 - value is the number of credits for the course.
 - Example:
 - 6
- ❖ **semester** – attribute representing the semester of the education of the particular subject.
 - value is one character, where value “S” is for the summer semester and “W” for the winter semester.
 - Example:
 - S

- ❖ **ending_type** – attribute is the symbol for the form of the exam.
 - value is one character:
 - **B** = exam + accreditation to exam,
 - **E** = exam,
 - **S** = semester only (no exam).
 - Example:
 - E

Primary key

The primary key is composite, formed by attributes *subject_id* and *school_year*.

Foreign key

Attribute *subject_id* is the *foreign key* to the table *Subject*,
 Attribute *guarantee* is the *foreign key* to the table *Teacher*.

SQL script for table creation

```
Create table subject_year
(
  school_year    Char(4)          NOT NULL,
  subject_id     Varchar2(30)     NOT NULL,
  guarantee      Char(5)          NOT NULL,
  ects           Number(3, 0)     NOT NULL,
  semester       Char(1)          NOT NULL,
  ending_type    Char(1)          NOT NULL,
  primary key(school_year, subject_id)
);
```

Script for relationship definition

```
Alter table subject_year
  add foreign key (subject_id)
    references subject (subject_id);

Alter table subject_year
  add foreign key (guarantee)
    references teacher (teacher_id);
```

Table data example

Tab. A.7: Subject_year

<i>school_year</i>	<i>subject_id</i>	<i>guarantee</i>	<i>ects</i>	<i>semester</i>	<i>ending_type</i>
2009	IA06	KDS01	5	S	B
2009	IM09	KMT01	6	W	B
2009	IZ01	GAR01	30	S	E
2009	IM15	KMT02	4	W	B
2009	IPD1	KDS03	6	W	S
2008	BI23	EX001	5	W	B
2005	BI01	KI001	6	W	B
2007	BI06	KI001	6	S	B
2006	BI11	EX002	1	W	S

Table ST_PROGRAM

This table contains details of the content of study programs offered by the faculty for the study in the defined school year. This table is connected to the tables *St_field* and *Subject_year*.

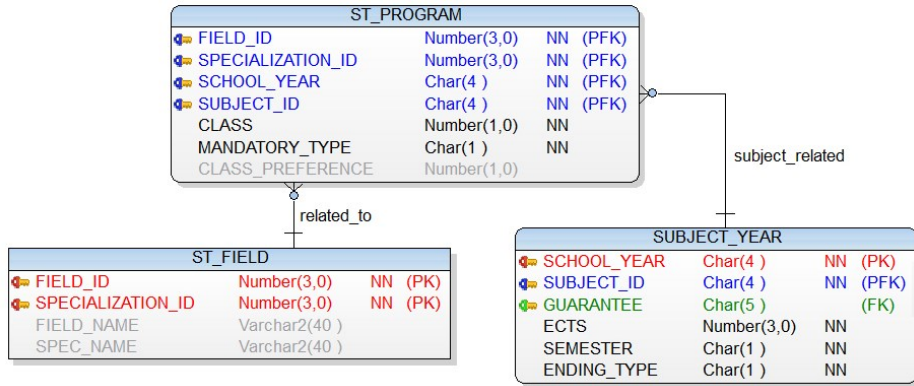


Fig. A.8: *St_program* submodel

Attributes

- ❖ **field_id** – this attribute is used for the identification of the study field.
 - the number is composed maximally of 3 digits.
 - this attribute is part of the *primary key* of the table.
 - this attribute is also part of the *foreign key* referencing table *St_field*.
 - Example:
 - 1
- ❖ **specialization_id** – this attribute is used for the identification of the study specialization.
 - the number is composed maximally of 3 digits.
 - this attribute is part of the *primary key* of the table.
 - this attribute is also part of the *foreign key* referencing table *St_field*.
 - Example:
 - 2
- ❖ **school_year** – number of the actual school year.
 - attribute *school_year* is the part of the *primary key* of the table.
 - value is the integer composed of 4 digits.
 - Example:
 - 2017 ... this number represents the school year 2017/2018
- ❖ **subject_id** – identification number of the subject.
 - attribute *subject_id* is the part of the *primary key* of the table.
 - this attribute is also the *foreign key* referencing table *Subject*.
 - value is the string composed maximally of 30 characters (see table *Subject*).
 - Example:
 - BI06
- ❖ **class** – attribute represents the year of the study during education.
 - value is expressed by one digit.
 - Example:
 - 1 ... the subject is respective to the first year of the study.

- ❖ **mandatory_type** – attribute value characterizes the kind of the course.
 - value is one character:
 - **M** = *mandatory*,
 - **O** = *optional*,
 - **X** = *mandatory / optional*,
 - Example:
 - M
- ❖ **class_preference** – attribute represents recommended year of the study during education.
 - value is one digit.
 - Example:
 - 1 ... the subject is recommended to be studied in the first year of the study.

Primary key

The primary key is composite, formed by attributes *field_id*, *specialization_id*, *subject_id*, and *school_year*.

Foreign key

Composite attribute group (*school_year*, *subject_id*) is the *foreign key* to the table *Subject_year*.

The composite attribute group (*field_id*, *specialization_id*) is the *foreign key* to the table *St_field*.

SQL script for table creation

```
Create table st_program
(
    field_id          Number(3, 0)    NOT NULL,
    specialization_id Number(3, 0)    NOT NULL,
    school_year       Char(4)         NOT NULL,
    subject_id        Varchar2(30)    NOT NULL,
    class             Number(1, 0)    NOT NULL,
    mandatory_type     Char(1)         NOT NULL,
    class_preference   Number(1, 0),
    primary key(field_id, specialization_id, school_year, subject_id)
);
```

Script for relationship definition

```
Alter table st_program
add foreign key (school_year, subject_id)
references subject_year (school_year, subject_id);

Alter table st_program
add foreign key (field_id, specialization_id)
references st_field (field_id, specialization_id);
```

Table data example

Tab. A.8: St_program

<i>field_id</i>	<i>speciali zation_id</i>	<i>school_year</i>	<i>subject_id</i>	<i>class</i>	<i>mandatory _type</i>	<i>class _preference</i>
200	2	2009	IPA3	2	M	2
200	3	2009	IPN3	2	M	2
202	0	2009	IPM3	2	M	2
101	0	2009	III14	3	O	3
102	0	2009	III14	3	O	3
100	0	2009	IS05	3	O	3
101	0	2009	IS05	3	O	3
102	0	2009	IS05	3	O	3
200	0	2009	IM20	0	O	3
200	0	2009	IM12	0	O	3

Table CONTACT

This table contains details about the contacts of the persons. This table is connected only to the *Personal_data* table. Notice that in this table, a foreign key can hold a *NULL* value.

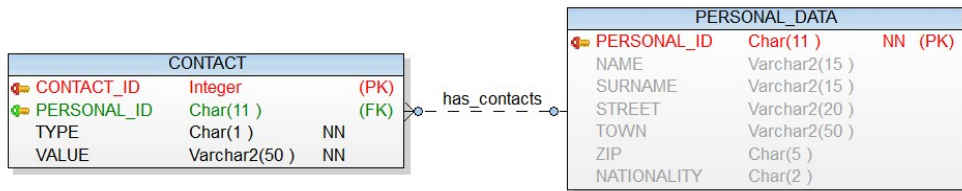


Fig. A.9: Contact submodel

Attributes

- ❖ **contact_id** – identification key of the contact.
 - attribute *contact_id* is the primary key of the table.
 - value is the integer.
 - Example:
 - 111
- ❖ **personal_id** – unique identifier of the person.
 - attribute *personal_id* is the *foreign key* of the table *Personal_data*.
 - the data type is the string with exactly 11 characters. It follows this structure:
 - YYMMDD/XXXX where:
 - YY is two digits for the year of birth of the person,
 - MM is two digits for the month of birth of the person,
 - DD is two digits for the day of birth of the person (for women, 50 is added to the appropriate value),
 - “/” – separator,
 - XXXX are four digits for defining the order number of the person.
 - Notice that in a standard environment, the *personal_id* value can be divided by 11 without the remainder.
 - Example:
 - 890811/0134 is the identification for the person born on 11th August 1989 with order number 0134. It reflects the man.
 - 895811/0137 is the identification for the person born on 11th August 1989 with order number 0137. It reflects the woman.
- ❖ **type** – this attribute defines kind of the contact.
 - value is the only one character:
 - **M** = *mobile*
 - **E** = *email*
 - Example:
 - E
- ❖ **value** – this attribute includes real contact value.
 - value is the string with a variable length of the characters limited to 50 characters.
 - Example:
 - name.surname@email.com;
 - 0912 345 678;

Primary key

The primary key is attribute *contact_id*.

Foreign key

Attribute *personal_id* is the *foreign key* to the table *Personal_data*. Notice that a foreign key can hold a *NULL* value.

SQL script for table creation

```
Create table contact
(
    contact_id      Integer          NOT NULL,
    personal_id     Char(11),
    type            Char(1)          NOT NULL,
    value           Varchar2(50)    NOT NULL,
    primary key (contact_id)
);
```

Script for relationship definition

```
Alter table contact
add foreign key (personal_id)
references personal_data (personal_id);
```

Table data example

Tab. A.9: Contact

<i>contact_id</i>	<i>personal_id</i>	<i>type</i>	<i>value</i>
1	841106/3456	E	Michael.Pearce@dbs.web
2	840312/7845	E	Jack.Smith@dbs.web
3	860907/1259	E	John.Young@dbs.web
4	850130/3695	E	Carol.Pearce@dbs.web
5	841201/1248	E	Carol.Pearce@dbs.web
6	830514/5341	E	Wiliam.Whittel@dbs.web
7	781015/4431	E	Peter.Roger@dbs.web
8	896123/5471	M	22368479
9	840409/7900	M	8404097900
10	810101/8079	M	0908123456

APPENDIX B – MODEL FLIGHT

Data model *Flight* consists of eleven tables (*L_person*, *L_flight_ticket*, *L_class*, *L_flight*, *L_plane*, *L_employee*, *L_airport*, *L_plane_type*, *L_country*, *L_town* and *L_air_company*). It deals with flights and relevant information to them (plane, plane types, air_company, employee) and people booking them (persons). For management simplicity, each table is prefixed by the “L_”. Therefore, it is easy to distinguish the model, which it belongs to.

Table L_PERSON

This table contains information about the details of the person. Such table is connected to the table *L_country*, *L_employee*, and *L_flight_ticket*.

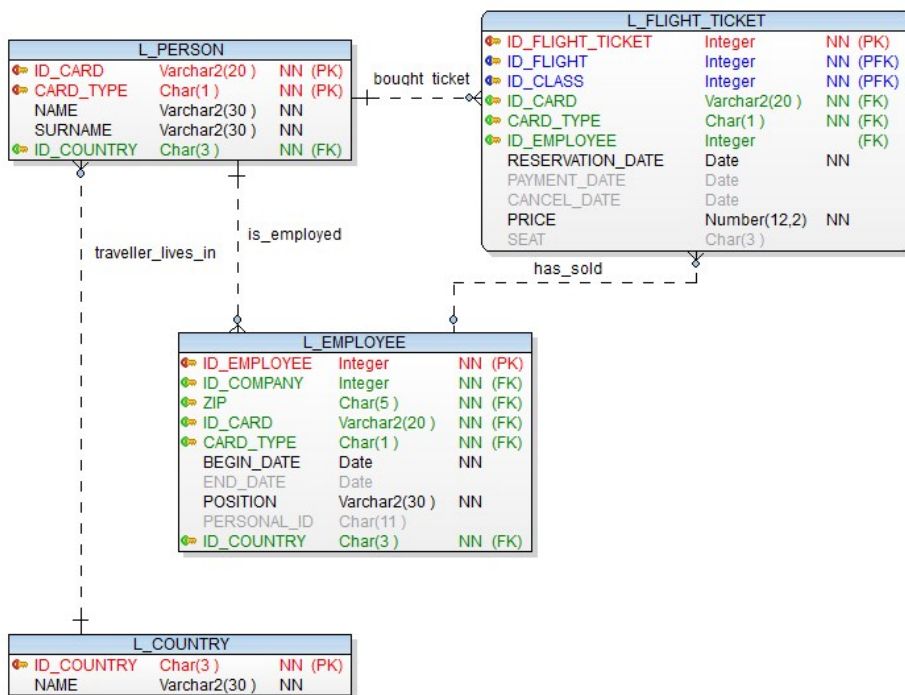


Fig. B.1: *L_person* submodel

Attributes

- ❖ **id_card** is the identification key of the card associated with the person.
 - attribute *id_card* is part of the *primary key* of the table.
 - value is the string with variable length limited to 20 characters.
 - Example:
 - AA 93286116

- ❖ **card_type** – value expresses the type of the card for the person.
 - attribute *card_type* is part of the primary key of the table.
 - value is one character belonging to the domain *dom_card_type*:
 - *I* = *ID card*
 - *P* = *passport*
 - Example:
 - P
- ❖ **name** – first name of the person.
 - value is the string with variable length limited to 30 characters.
 - Example:
 - Karol
- ❖ **surname** – family name of the person.
 - value is the string with variable length limited to 30 characters.
 - Example:
 - Matiaško
- ❖ **id_country** – attribute expresses country person comes from.
 - attribute *id_card* is the part of the *foreign key* referencing table *L_country*.
 - value is the string with a length of (*maximally*) 3 characters.
 - Example:
 - SK

Primary key

The primary key is composite, formed by attributes *id_card* and *card_type*.

Foreign key

Attribute *id_country* is the foreign key to the *L_country* table.

SQL script for table creation

```
Create table L_person
(
    id_card      Varchar2(20)  NOT NULL,
    card_type    Char(1)       NOT NULL
                Check (card_type IN ('I', 'P')),
    name         Varchar2(30)  NOT NULL,
    surname      Varchar2(30)  NOT NULL,
    id_country    Char(3)       NOT NULL,
    primary key (id_card, card_type)
);
```

Script for relationship definition

```
Alter table L_person
add foreign key (id_country)
references L_country (id_country);
```

Table L_FLIGHT TICKET

This table contains information about the details of a bought ticket for the flight of the particular person. Such table is connected to the table *L_person*, *L_class*, and *L_employee*.

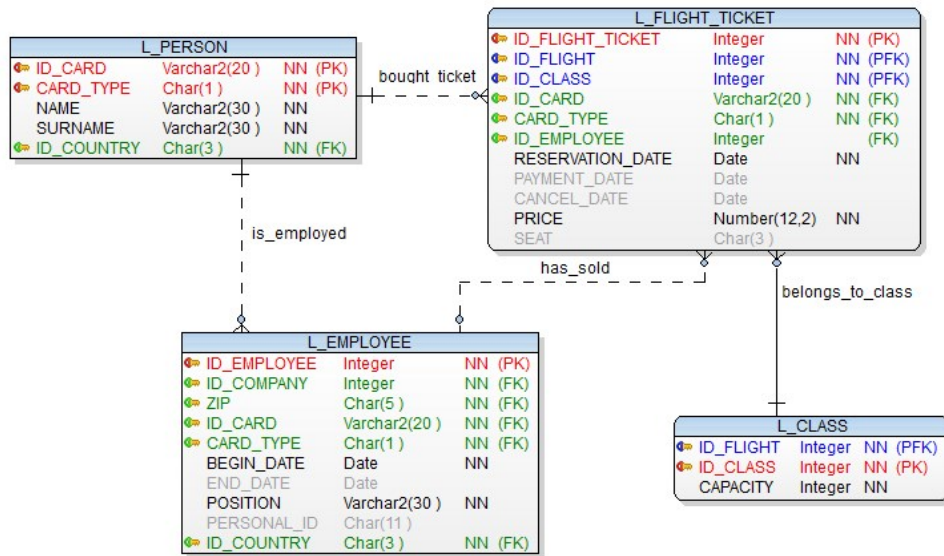


Fig. B.2: *L_flight_ticket* submodel

Attributes

- ❖ **id_flight_ticket** is the identification key of the ticket associated with the flight.
 - attribute *id_flight_ticket* is part of the *primary key* of the table.
 - value is an integer.
 - Example:
 - 1245
- ❖ **id_class** is the identification key of the class associated with the ticket.
 - attribute *id_class* is part of the *primary key* of the table.
 - attribute *id_class* is part of the *foreign key* referencing table *L_class*.
 - value belongs to the domain *dom_class*, which contains positive integers.
 - Example:
 - 1
- ❖ **id_flight** is the identification of the flight associated with the ticket.
 - attribute *id_flight* is part of the *primary key* of the table.
 - attribute *id_flight* is part of the *foreign key* referencing table *L_class*, respectively *L_flight*.
 - value is an integer.
 - Example:
 - 806068

- ❖ **id_card** is the identification key of the card associated with the passenger.
 - attribute *id_card* is part of the *foreign key* referencing table *L_person*.
 - value is the string with variable length limited to 20 characters.
 - Example:
 - AH 79800501
- ❖ **card_type** – value expresses the type of the card for the person.
 - attribute *card_type* is part of the *foreign key* referencing table *L_person*.
 - value is one character belonging to the domain *dom_card_type*:
 - **I** = **ID card**,
 - **P** = **passport**.
 - Example:
 - P
- ❖ **id_employee** determines the employee who sold such a ticket.
 - attribute *id_employee* is the *foreign key* referencing table *L_employee*.
 - value is the integer.
 - Example:
 - 85
- ❖ **reservation_date** determines the reservation time of the ticket.
 - value is the *Date* data type.
 - Example:
 - 12.6.2017
- ❖ **payment_date** determines the payment time of the ticket.
 - value is the *Date* data type.
 - Example:
 - 18.6.2017
- ❖ **cancel_date** determines the cancel time of the ticket.
 - value is the *Date* data type.
 - Example:
 - 22.1.2017
- ❖ **price** determines the price of the ticket.
 - value is the decimal number.
 - Example:
 - 110.50
- ❖ **seat** determines a specific number of the seat particular to the flight ticket.
 - value is the string formed by precisely 3 characters.
 - Example:
 - 12A

Primary key

The primary key is composite, formed by attributes *id_flight_ticket*, *id_flight*, and *id_class*.

Foreign key

Composite attributes (*id_card*, *card_type*) form the foreign key to the *L_person* table.

Composite attributes (*id_flight*, *id_class*) form the foreign key to the *L_class* table.

SQL script for table creation

```
Create table L_flight_ticket
(
    id_flight_ticket      Integer           NOT NULL,
    id_flight             Integer           NOT NULL,
    id_class              Smallint          NOT NULL
                        Check (id_class in (1, 2, 3)),
    id_card               Varchar2(20)      NOT NULL,
    card_type             Char(1)           NOT NULL
                        Check (card_type IN ('I', 'P')),
    id_employee           Integer,
    reservation_date      Date              NOT NULL,
    payment_date          Date,
    cancel_date           Date,
    price                 Number(12, 2)      NOT NULL,
    seat                  Char(3),
    primary key (id_flight_ticket, id_flight, id_class)
);
```

Script for relationship definition

```
Alter table L_flight_ticket
add foreign key (id_employee)
references L_employee (id_employee);

Alter table L_flight_ticket
add foreign key (id_flight, id_class)
references L_class (id_flight, id_class);
```

Table L_CLASS

This table contains information about the details of the flight categories (economic, business, etc.). Such table is connected to the table *L_flight_ticket* and *L_flight*.

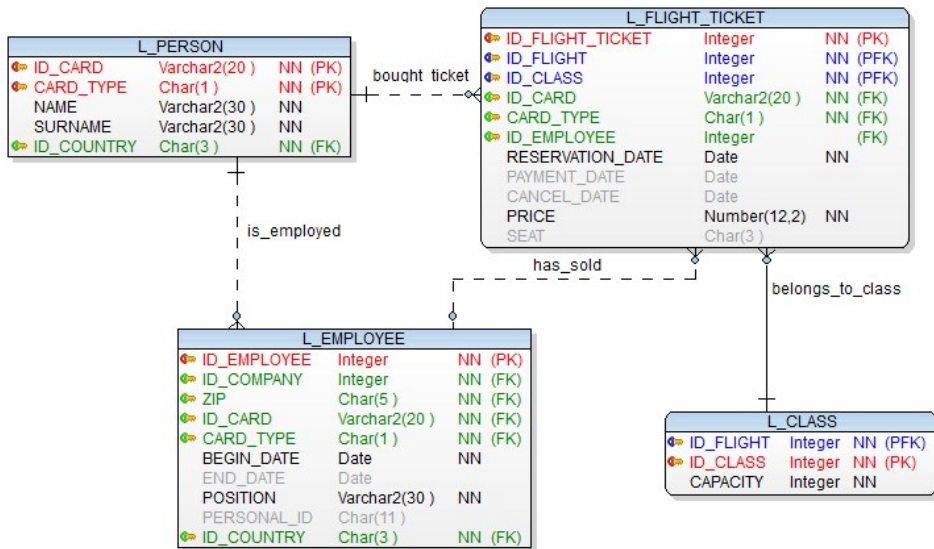


Fig. B.3: *L_class* submodel

Attributes

- ❖ **id_class** is the identification key of the class associated with the flight.
 - attribute *id_class* is part of the *primary key* of the table.
 - attribute *id_class* is part of the *foreign key* referencing table *L_flight_ticket*.
 - value belongs to the domain *dom_class*, which contains positive integers.
 - Example:
 - 1
- ❖ **id_flight** is the identification key associated with the flight.
 - attribute *id_flight* is part of the *primary key* of the table.
 - attribute *id_flight* is part of the *foreign key* referencing table *L_flight_ticket*.
 - value is an integer number
 - Example:
 - 19691
- ❖ **capacity** determines the total number of seats of the particular class in the flight.
 - value is the positive integer and belongs to the domain *dom_capacity*.
 - Example:
 - 160

Primary key

The primary key is composite, formed by attributes *id_flight* and *id_class*.

Foreign key

Attribute *id_flight* is the foreign key to the *L_flight* table.

SQL script for table creation

```
Create table L_class
(
    id_flight      Integer      NOT NULL,
    id_class       Smallint     NOT NULL
                    Check (id_class in (1, 2, 3)),
    capacity       Smallint     NOT NULL
                    Check (capacity > 0),
    primary key (id_flight, id_class)
);
```

Script for relationship definition

```
Alter table L_class
add foreign key (id_flight)
references L_flight (id_flight);
```

Table L_FLIGHT

This table contains information about the details flight. Such table is connected to the table *L_class*, *L_airport*, *L_air_company*, and *L_plane*.

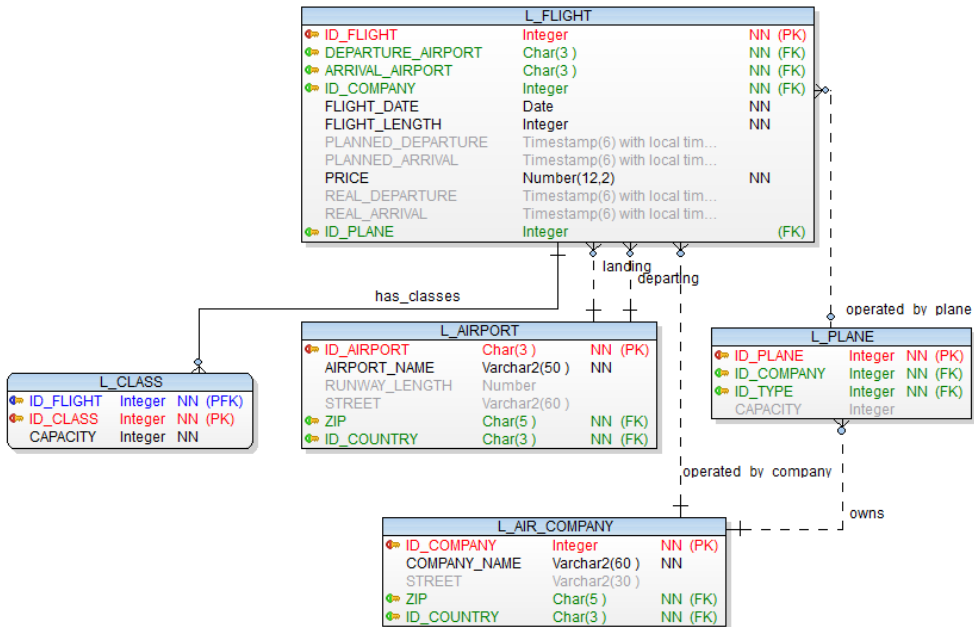


Fig. B.4: L_flight submodel

Attributes

- ❖ **id_flight** is the identification key of the flight.
 - attribute *id_flight* is the *primary key* of the table.
 - value is the integer number
 - Example:
 - 12345
- ❖ **departure_airport** defines the identifier of the departure airport.
 - attribute *departure_airport* is the *foreign key* referencing table *L_airport*.
 - value is the string composed of 3 characters.
 - Example:
 - BTS
- ❖ **arrival_airport** defines the identifier of the destination airport.
 - attribute *arrival_airport* is the *foreign key* referencing table *L_airport*.
 - value is the string composed of 3 characters.
 - Example:
 - PRG
- ❖ **id_company** defines the identification value for the air company.
 - attribute *id_company* is the *foreign key* referencing table *L_air_company*.
 - value is the integer.
 - Example:
 - 8

- ❖ **flight_date** defines the date of the flight.
 - value is the *Date* data type.
 - Example:
 - 17.8.2017
- ❖ **flight_length** defines the duration of the flight in minutes.
 - value is the integer
 - Example:
 - 120
- ❖ **planned_departure** defines the date and time of the flight departure (planned, expected).
 - value is the timestamp with the local time zone.
 - Example:
 - 17.8.2017 19:30
- ❖ **planned_arrival** defines the date and time of the flight arrival (planned, expected).
 - value is the timestamp with the local time zone.
 - Example:
 - 17.8.2017 21:30
- ❖ **price** is the value of the ticket.
 - value is a decimal number with two digits after the decimal point.
 - Example:
 - 100.50
- ❖ **real_departure** defines the actual date and time of the flight departure.
 - value is the timestamp with the local time zone.
 - Example:
 - 17.8.2017 19:30
- ❖ **real_arrival** specifies the actual date and time of the flight arrival.
 - value is the timestamp with the local time zone.
 - Example:
 - 17.8.2017 21:30
- ❖ **id_plane** determines the used plane for the flight.
 - attribute *id_plane* is the *foreign key* referencing table *L_plane*.
 - value is an integer
 - Example:
 - 17

Primary key

The primary key is attribute *id_flight*.

Foreign key

Attribute *id_plane* is the foreign key to the *L_plane* table.

Attribute *departure_airport* is the foreign key to the *L_airport* table.

Attribute *arrival_airport* is the foreign key to the *L_airport* table.

Attribute *id_company* is the foreign key to the *L_air_company* table.

SQL script for table creation

```
Create table L_flight
(
    id_flight            Integer            NOT NULL,
    departure_airport    Char(3)           NOT NULL,
    arrival_airport      Char(3)           NOT NULL,
    id_company           Integer            NOT NULL,
    flight_date          Date              NOT NULL,
    flight_length        Integer            NOT NULL,
    planned_departure    Timestamp(6) with local time zone,
    planned_arrival      Timestamp(6) with local time zone,
    price                Number(12,2)       NOT NULL,
    real_departure        Timestamp(6) with local time zone,
    real_arrival         Timestamp(6) with local time zone,
    id_plane             Integer,
    primary key (id_flight)
);
```

Script for relationship definition

```
Alter table L_flight
add foreign key (departure_airport)
references L_airport (id_airport);

Alter table L_flight
add foreign key (arrival_airport)
references L_airport (id_airport);

Alter table L_flight
add foreign key (id_plane)
references L_plane (id_plane);

Alter table L_flight
add foreign key (id_company)
references L_air_company (id_company);
```

Table L_PLANE

This table contains information about the details of the airplane owned by a particular air company. Such table is connected to the table *L_air_company*, *L_flight*, and *L_plane_type*.

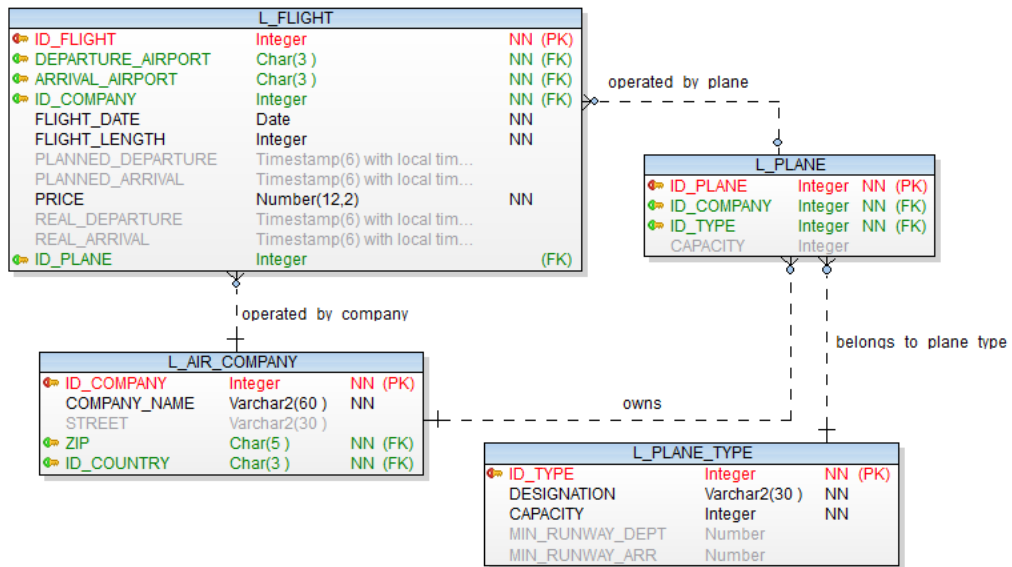


Fig. B.5: *L_plane* submodel

Attributes

- ❖ **id_plane** is the identification key of the airplane.
 - attribute *id_plane* is the *primary key* of the table.
 - value is the integer.
 - Example:
 - 125
- ❖ **id_company** is the identification of the owner air company of the plane.
 - attribute *id_company* is the *foreign key* referencing table *L_air_company*.
 - value is the integer.
 - Example:
 - 5
- ❖ **id_type** determines the kind of airplane.
 - attribute *id_type* is the *foreign key* referencing table *L_plane_type*.
 - value is the integer.
 - Example:
 - 5
- ❖ **capacity** determines the number of seats inside the airplane.
 - value is the positive integer number and belongs to the domain *dom_capacity*.
 - Example:
 - 160

Primary key

The primary key is attribute *id_plane*.

Foreign key

Attribute *id_company* is the foreign key to the *L_air_company* table.

Attribute *id_type* is the foreign key to the *L_plane_type* table.

SQL script for table creation

```
Create table L_plane (  
    id_plane      Integer      NOT NULL,  
    id_company    Integer      NOT NULL,  
    id_type       Integer      NOT NULL,  
    capacity      Smallint  
                Check (capacity > 0),  
    primary key (id_plane)  
);
```

Script for relationship definition

```
Alter table L_plane  
    add foreign key (id_company)  
        references L_air_company (id_company);  
  
Alter table L_plane  
    add foreign key (id_type)  
        references L_plane_type (id_type);
```

Table L_EMPLOYEE

This table contains information about the details of the employee of the air company. Such table is connected to the table *L_person*, *L_air_company*, *L_town*, and *L_flight_ticket*.

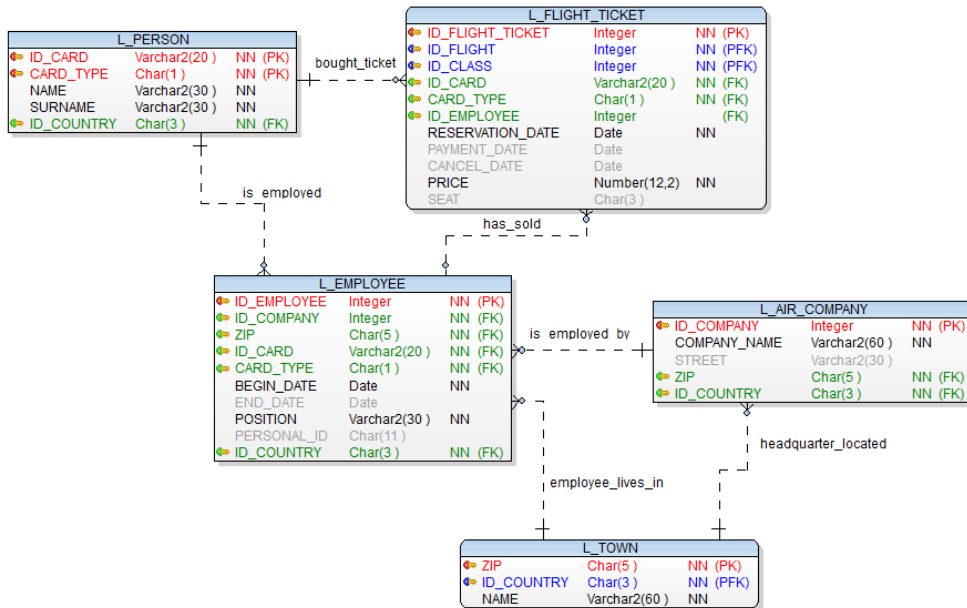


Fig. B.6: *L_employee* submodel

Attributes

- ❖ **id_employee** is the identification key of the person who the particular air company employs.
 - attribute *id_employee* is the *foreign key* referencing table *L_flight_ticket*.
 - value is the integer.
 - Example:
 - 9618
- ❖ **id_company** is the identification key of the air company.
 - attribute *id_company* is the *foreign key* referencing table *L_air_company*.
 - value is the integer.
 - Example:
 - 5
- ❖ **id_card** is the identification key of the card associated with the person.
 - attribute *id_card* is part of the *foreign key* referencing table *L_person*.
 - value is the string with variable length limited to 20 characters.
 - Example:
 - AH 79800501

- ❖ **card_type** – value expresses the type of the card for the person.
 - attribute *card_type* is part of the *foreign key* referencing table *L_person*.
 - value is one character belonging to the domain *dom_card_type*:
 - **I** = *ID card*
 - **P** = *passport*
 - Example:
 - P
- ❖ **begin_date** expresses the first date of the employer contract validity.
 - value is the *Date* data type.
 - Example:
 - 1.1.2000
- ❖ **end_date** expresses the last date of the employer contract validity.
 - value is the *Date* data type.
 - Example:
 - 31.1.2017
- ❖ **position** expresses the function of the person in the air company.
 - value is the string with variable length limited to 30 characters.
 - Example:
 - Steward
- ❖ **personal_id** is the identification of the person expressing his birth number.
 - the data type is the string with exactly 11 characters. It follows this structure:
 - YYMMDD/XXXX where:
 - YY is two digits for the year of birth of the person,
 - MM is two digits for the month of birth of the person,
 - DD is two digits for the day of birth of the person (for women, 50 is added to the appropriate value),
 - “/” – separator,
 - XXXX are four digits for defining the order number of the person.
 - Notice that in a standard environment, the *personal_id* value can be divided by 11 without the remainder.
 - Example:
 - 890811/0134 is the identification for the person born on 11th August 1989 with order number 0134. It reflects the man.
 - 895811/0137 is the identification for the person born on 11th August 1989 with order number 0137. It reflects the woman.
- ❖ **zip** is the identification key of the town (associated with the country) where such a person is employed.
 - attribute *zip* is part of the *foreign key* referencing table *L_town*.
 - value is the string with a fixed length of 5 numerical characters. Although it expresses numerical value, it is stored as a string due to possible initial zeros.
 - Example:
 - 97251
 - 01001

- ❖ **id_country** is the identification key associated with the country of the town.
 - attribute *id_country* is part of the foreign key references table *L_town*.
 - value is the string with a length of (*maximally*) 3 characters.
 - Example:
 - SK

Primary key

The primary key is attribute *id_employee*.

Foreign key

Attribute *id_company* is the foreign key to the *L_air_company* table.

Composite attributes (*id_card, card_type*) form the foreign key to the *L_person* table.

Composite attributes (*zip, id_country*) form the foreign key to the *L_town* table.

SQL script for table creation

```
Create table L_employee (
  id_employee      Integer          NOT NULL,
  id_company       Integer          NOT NULL,
  zip             Char(5)          NOT NULL,
  id_card         Varchar2(20)     NOT NULL,
  card_type       Char(1)         NOT NULL
                  Check (card_type IN ('I', 'P')),
  begin_date      Date            NOT NULL,
  end_date        Date,
  position        Varchar2(30)    NOT NULL,
  personal_id     Char(11),
  id_country      Char(3)         NOT NULL,
  primary key (id_employee)
);
```

Script for relationship definition

```
Alter table L_employee
  add foreign key (id_card, card_type)
  references L_person (id_card, card_type);

Alter table L_employee
  add foreign key (zip, id_country)
  references L_town (zip, id_country);

Alter table L_employee
  add foreign key (id_company)
  references L_air_company (id_company);
```

Table L_AIRPORT

This table contains information about the details of the airport. Such table is connected to the table *L_flight* and *L_town*.

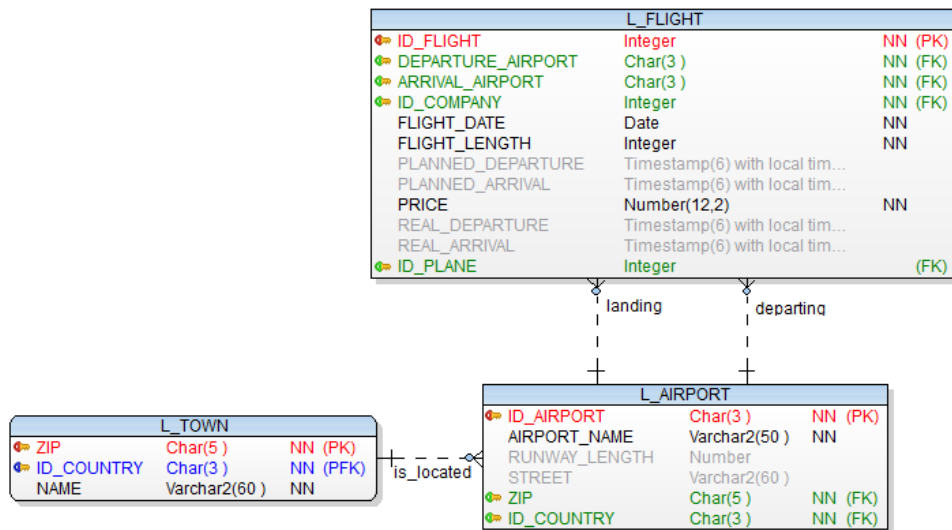


Fig. B.7: *L_airport* submodel

Attributes

- ❖ **id_airport** is the identification value for the airport.
 - attribute *id_airport* is the *primary key* for the airport.
 - value is the string with 3 characters in length.
 - Example:
 - LHR
- ❖ **airport_name** is the name of the airport.
 - value is the string with variable length limited to 50 characters.
 - Example:
 - Heathrow
- ❖ **runway_length** is the distance of the runway of the airport.
 - value is the numerical, and expressed value is in meters.
 - Example:
 - 2000
- ❖ **zip** is the identification key of the town associated with the country.
 - attribute *zip* is part of the *foreign key* referencing table *L_town*.
 - value is the string with a fixed length of 5 numerical characters. Although it expresses numerical value, it is stored as a string due to possible initial zeros.
 - Example:
 - 97251
 - 01001

- ❖ **id_country** is the identification key associated with the country of the town.
 - attribute *id_country* is part of the *foreign key* referencing table *L_town*.
 - value is the string with a length of (*maximally*) 3 characters.
 - Example:
 - SK
- ❖ **street** is the name of the street where the airport is located in the town.
 - value is the string with variable length limited to 60 characters.
 - Example:
 - Nelson Road

Primary key

The primary key is attribute *id_airport*.

Foreign key

Composite attributes (*zip, id_country*) form the foreign key to the *L_town* table.

SQL script for table creation

```
Create table L_employee
(
    id_employee      Integer      NOT NULL,
    id_company       Integer      NOT NULL,
    zip              Char(5)      NOT NULL,
    id_card          Varchar2(20) NOT NULL,
    card_type        Char(1)      NOT NULL
                    Check (card_type IN ('I', 'P')),
    begin_date       Date         NOT NULL,
    end_date         Date,
    position         Varchar2(30) NOT NULL,
    personal_id      Char(11),
    id_country       Char(3)      NOT NULL,
    primary key (id_employee)
);
```

Script for relationship definition

```
Alter table L_airport
add foreign key (zip, id_country)
references L_town (zip, id_country);
```

Table L_PLANE_TYPE

This table contains information about the technical details of the airport category. Such table is connected to the table *L_plane*.

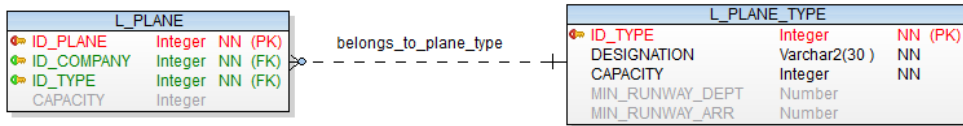


Fig. B.8: *L_plane_type* submodel

Attributes

- ❖ **id_type** is the identification key of the kind of airplane.
 - attribute *id_type* is the *primary key* of the table.
 - value is the integer
 - Example:
 - 1
- ❖ **designation** determines the type of airplane.
 - value is the string with variable length limited to 30 characters.
 - Example:
 - Boeing 360
- ❖ **capacity** determines the number of seats on the airplane.
 - value is the positive integer number belonging to the *dom_capacity* domain.
 - Example:
 - 160
- ❖ **min_runway_dep** expresses minimal distance of the runway of the airport for the departure of the flight.
 - value is the integer, expressed in meters.
 - Example:
 - 2000
- ❖ **min_runway_arr** expresses minimal distance of the runway of the airport for the arrival of the flight.
 - value is the integer, expressed in meters.
 - Example:
 - 2000

Primary key

The primary key is attribute *id_type*.

Foreign key

The table has no foreign keys.

SQL script for table creation

```
Create table L_plane_type
(
    id_type           Integer           NOT NULL,
    designation       Varchar2(30)     NOT NULL,
    capacity          Smallint         NOT NULL
                        Check (capacity > 0),
    min_runway_dept   Number,
    min_runway_arr    Number,
    primary key (id_type)
);
```

Script for the relationship definition

None.

Table L_COUNTRY

This table contains information about the country. Such table is connected to the table *L_person* and *L_town*.

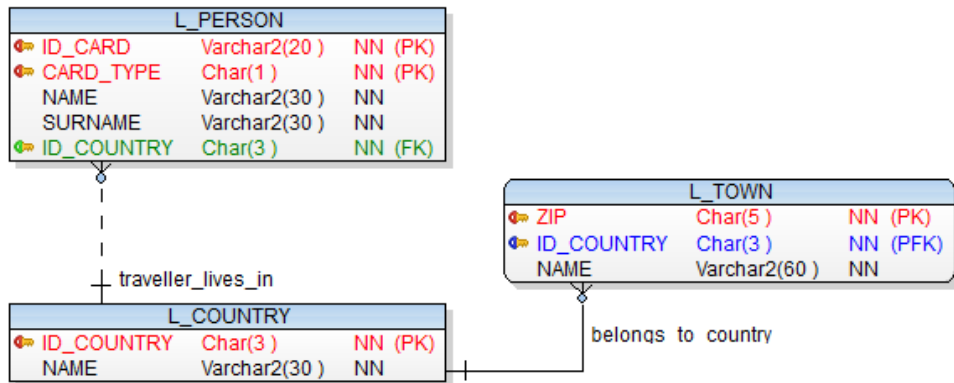


Fig. B.9: L_country submodel

Attributes

- ❖ **id_country** is the identification key of the table.
 - attribute *id_country* is the *primary* key of the table.
 - value is the string with a length of (*maximally*) 3 characters.
 - Example:
 - SK
- ❖ **name** – name of the country.
 - value is the string with variable length limited to 30 characters.
 - Example:
 - Slovakia

Primary key

The primary key is attribute *id_country*.

Foreign key

The table has no foreign keys.

SQL script for table creation

```
Create table L_country
(
    id_country      Char(3)          NOT NULL,
    name            Varchar2(30)     NOT NULL,
    primary key (id_country)
);
```

Script for the relationship definition

None.

Table L_TOWN

This table contains information about the town. Such table is connected to the table *L_country*, *L_air_company*, *L_employee*, and *L_airport*.

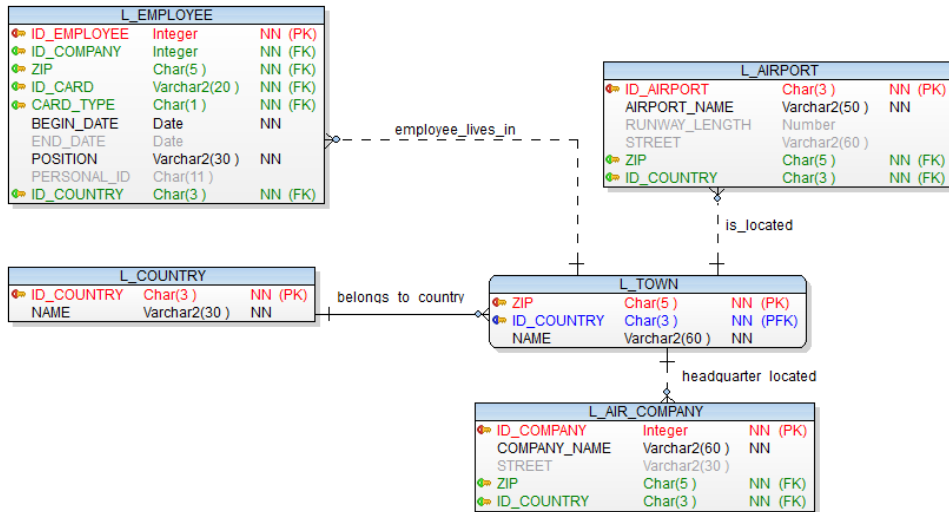


Fig. B.10: *L_town* submodel

Attributes

- ❖ **zip** is the identification key of the town associated with the country.
 - attribute *zip* is part of the *primary key* of the table.
 - value is the string with a fixed length of 5 numerical characters. Although it expresses numerical value, it is stored as a string due to possible initial zeros.
 - Example:
 - 97251
 - 01001
- ❖ **id_country** is the identification key associated with the country of the town.
 - attribute *id_country* is part of the *primary key* of the table.
 - attribute *id_country* is the *foreign key* referencing table *L_country*.
 - value is the string with a length of (*maximally*) 3 characters.
 - Example:
 - SK
- ❖ **name** – name of the town.
 - value is the string with variable length limited to 60 characters.
 - Example:
 - ZILINA

Primary key

The primary key is composite, formed by attributes *zip* and *id_country*.

Foreign key

Attribute *id_country* is the foreign key to the *L_country* table.

SQL script for table creation

```
Create table L_town
(
    zip            Char(5)            NOT NULL,
    id_country     Char(3)            NOT NULL,
    name           Varchar2(60)       NOT NULL,
    primary key (zip, id_country)
);
```

Script for relationship definition

```
Alter table L_town
add foreign key (id_country)
references L_country (id_country);
```

Table L_AIR_COMPANY

This table contains information about the details of the air company. Such table is connected to the table *L_flight*, *L_plane*, *L_town*, and *L_employee*.

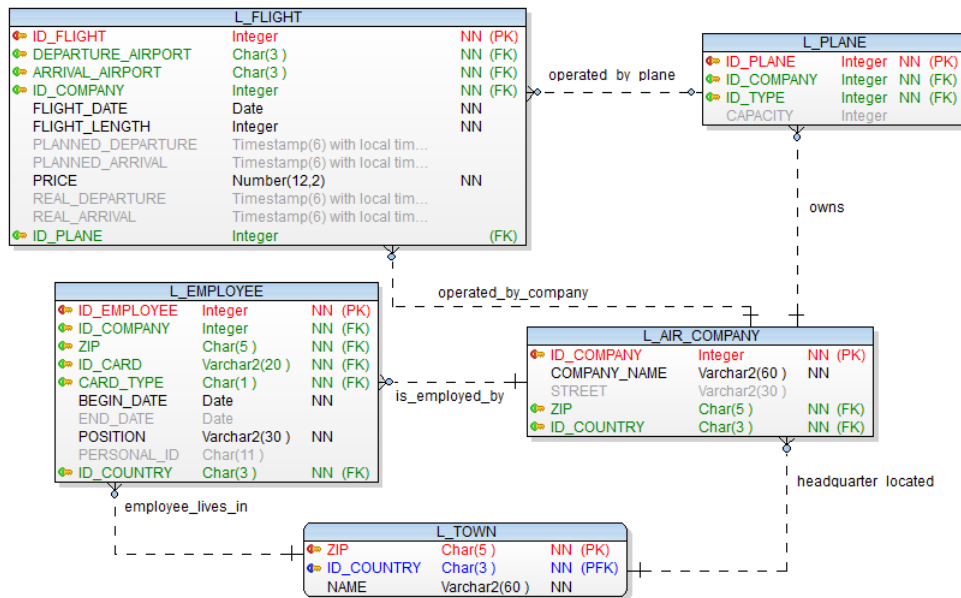


Fig. B.11: L_air_company submodel

Attributes

- ❖ **id_company** is the identification value for the air company.
 - attribute *id_company* is the *primary key* for the air company.
 - value is the integer number.
 - Example:
 - 5
- ❖ **company_name** is the name of the air company.
 - value is the string with variable length limited to 60 characters.
 - Example:
 - Czech Airlines
- ❖ **street** is the street's name where the air company headquarters is located in value is the string with variable length limited to 60 characters.
 - Example:
 - Ruzinska 21
- ❖ **zip** is the identification key of the town associated with the air company.
 - attribute *zip* is part of the *foreign key* referencing table *L_town*.
 - value is the string with a fixed length of 5 numerical characters. Although it expresses numerical value, it is stored as the string due to possible initial zeros.
 - Example:
 - 97251
 - 01001

- ❖ **id_country** is the identification key associated with the country of the town.
 - attribute *id_country* is part of the *foreign key* referencing table *L_town*.
 - value is the string with a length of (*maximally*) 3 characters.
 - Example:
 - SK

Primary key

The primary key is composite, formed by attributes *zip* and *id_company*.

Foreign key

The table has no foreign keys.

SQL script for table creation

```
Create table L_air_company
(
    id_company          Integer          NOT NULL,
    company_name        Varchar2(60)    NOT NULL,
    street              Varchar2(30),
    zip                 Char(5)          NOT NULL,
    id_country           Char(3)         NOT NULL,
    primary key (id_company)
);
```

Script for relationship definition

```
Alter table L_air_company
add foreign key (zip, id_country)
references L_town (zip, id_country);
```


APPENDIX C – MODEL LIBRARY

Data model *Library* consists of seven tables (*K_person*, *K_reader*, *K_rent_books*, *K_book*, *K_title*, *K_authors_of_book*, and *K_author*). It deals with person management (persons, readers), book management (authors, titles, physical books), and book renting. For management simplicity, each table is prefixed by the “K_”. Therefore, it is easy to distinguish the model, which it belongs to.

Table K_PERSON

This table contains information about the details of the person. Such table is connected to the table *K_reader*.

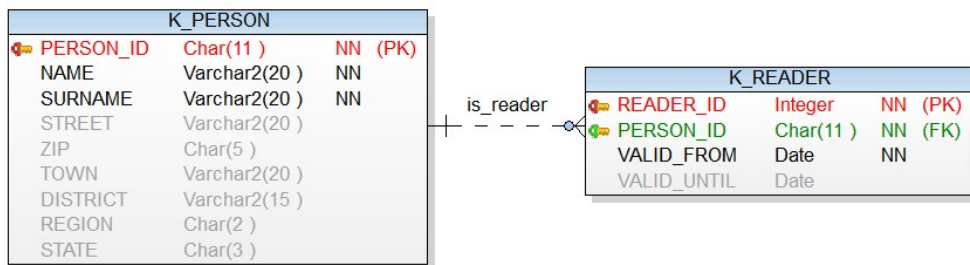


Fig. C.1: *K_person* submodel

Attributes

- ❖ **person_id** – unique identifier of the person.
 - attribute *person_id* is the *primary key* of the table.
 - The data type is the string with exactly 11 characters. It follows this structure:
 - YYMMDD/XXXX where:
 - YY is two digits for the year of birth of the person,
 - MM is two digits for the month of birth of the person,
 - DD is two digits for the day of birth of the person (for women, 50 is added to the appropriate value),
 - “/” – separator,
 - XXXX are four digits for defining the order number of the person.
 - Notice that in a standard environment, the *personal_id* value can be divided by 11 without the remainder.
 - Example:
 - 890811/0134 is the identification for the person born on 11th August 1989 with order number 0134. It reflects the man.
 - 895811/0137 is the identification for the person born on 11th August 1989 with order number 0137. It reflects the woman.
- ❖ **name** – first name of the person.
 - value is the string with variable length limited to 20 characters.
 - Example:
 - Karol

- ❖ **surname** – family name of the person.
 - value is the string with variable length limited to 20 characters.
 - Example:
 - Matiaško
- ❖ **street** – street and house number of the personal address.
 - value is the string with variable length limited to 20 characters.
 - Example:
 - Moyzesova 20
- ❖ **zip** – zip code of the address of the person.
 - value is the string with a fixed length of 5 numerical characters. Although it expresses numerical value, it is stored as a string due to possible initial zeros.
 - Example:
 - 97251
 - 01001
- ❖ **town** – the town of the address of the person.
 - value is the string with variable length limited to 20 characters.
 - Example:
 - Podunajske Biskupice
- ❖ **district** – district name where the town belongs.
 - value is the variable string with a maximal length of 20 characters.
 - Example:
 - Bratislava II ... *Bratislava* is divided into several parts, e.g., “*Podunajske Biskupice*” belongs to “*Bratislava II*” district.
- ❖ **region** – region abbreviation of the country.
 - value is the string with a fixed length of 2 characters.
 - Example:
 - BA ... expresses “Bratislava region”
- ❖ **state** – country abbreviation of the person.
 - value is the string with a fixed length of 3 characters.
 - Example:
 - SVK ... expresses “Slovakia”

Primary key

The primary key is attribute *person_id*.

Foreign key

The table has no foreign keys.

SQL script for table creation

```
Create table k_Person
(
    person_id      Char(11)           NOT NULL,
    name           Varchar2(20)       NOT NULL,
    surname        Varchar2(20)       NOT NULL,
    street         Varchar2(20),
    zip            Char(5),
    town           Varchar2(20),
    district       Varchar2(15),
    region         Char(2),
    state          Char(3),
    primary key (person_id)
);
```

Script for the relationship definition

None.

Table K_READER

This table contains details about a person registered as a reader. Such table is connected to the table *K_person* and *K_rent_books*.

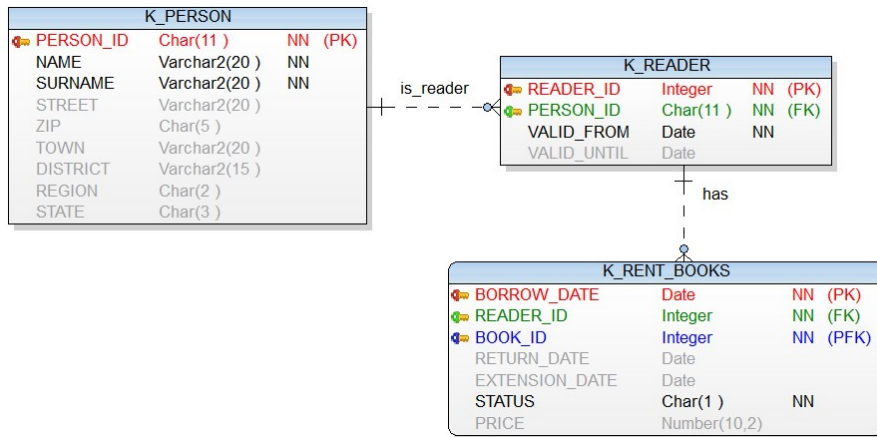


Fig. C.2: *K_reader* submodel

Attributes

- ❖ **reader_id** – identification number of the reader.
 - the attribute is the *primary key* of the table.
 - value is the integer.
 - Example:
 - 11111
- ❖ **person_id** – identification key of the person.
 - this attribute is the *foreign key* to the *K_person* table.
 - The data type is the string with exactly 11 characters. It follows this structure:
 - YYMMDD/XXXX where:
 - YY is two digits for the year of birth of the person,
 - MM is two digits for the month of birth of the person,
 - DD is two digits for the day of birth of the person (for women, 50 is added to the appropriate value),
 - “/” – separator,
 - XXXX are four digits for defining the order number of the person.
 - Notice that in a standard environment, the personal_id value can be divided by 11 without the remainder.
 - Example:
 - 890811/0134 is the identification for the person born on 11th August 1989 with order number 0134. It reflects the man.
 - 895811/0137 is the identification for the person born on 11th August 1989 with order number 0137. It reflects the woman.
- ❖ **valid_from** – value expresses the start date of the evidence validity.
 - Example:
 - 15.6.2017

- ❖ **valid_until** – value expresses the end date of the evidence validity.
 - Example:
 - 15.12.2017

Primary key

The primary key is attribute *reader_id*.

Foreign key

Attribute *person_id* is the foreign key to the *K_person* table.

SQL script for table creation

```
Create table k_Reader
(
    reader_id      Integer      NOT NULL,
    person_id      Char(11)     NOT NULL,
    valid_from     Date         NOT NULL,
    valid_until    Date,
    primary key (reader_id)
);
```

Script for relationship definition

```
Alter table k_Reader
add foreign key (person_id)
references k_Person (person_id);
```

Table K_RENT_BOOKS

This table contains information about the details of the rent books. Such table is connected to the table *K_reader* and *K_book*.

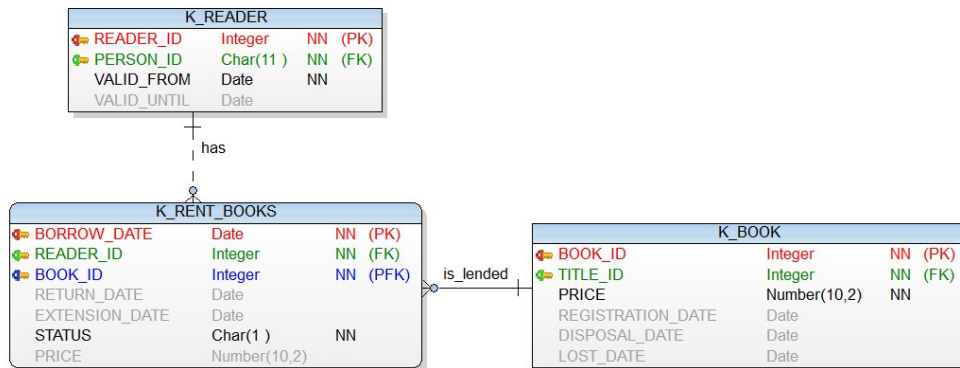


Fig. C.3: *K_rent_books* submodel

Attributes

- ❖ **borrow_date** – value is the day the book was borrowed.
 - the attribute is the part of the *primary key* of the table.
 - value has the *Date* data type.
 - Example:
 - 25.6.2015
- ❖ **book_id** – identification number of the book.
 - the attribute is the part of the *primary key* of the table.
 - value is the integer number.
 - Example:
 - 11111
- ❖ **reader_id** – identification number of the reader.
 - attribute *reader_id* is the foreign *key* referencing table *K_reader*.
 - value is the integer number.
 - Example:
 - 15
- ❖ **price** – the amount the reader has to pay the rent (e.g., due to book damage, loss, or late return).
 - value is the number composed of 10 digits with two decimal numbers.
 - Example:
 - 123.20
- ❖ **return_date** – value expresses a day when a particular book was returned to the library.
 - value has the *Date* data type.
 - Example:
 - 25.6.2015
 - NULL

- ❖ **extension_date** – value expresses a day rent was extended.
 - value has the *Date* data type.
 - Example:
 - 25.6.2015
 - NULL
- ❖ **status** – status of the book in a rent.
 - value is the one character:
 - *L – lost*
 - *B – borrowed*
 - *R – returned*
 - *D – damaged*

Primary key

The primary key is composite, formed by attributes *borrow_date* and *book_id*.

Foreign key

Attribute *reader_id* is the foreign key to the *K_reader* table.

Attribute *book_id* is the foreign key to the *K_book* table.

SQL script for table creation

```
Create table k_Rent_books (  
    borrow_date      Date           NOT NULL,  
    reader_id        Integer        NOT NULL,  
    book_id          Integer        NOT NULL,  
    return_date      Date,  
    extension_date   Date,  
    status           Char(1)        NOT NULL,  
    price            Number(10, 2),  
    primary key (borrow_date, book_id)  
);
```

Script for relationship definition

```
Alter table k_Rent_books  
    add foreign key (reader_id)  
        references k_Reader (reader_id);  
  
Alter table k_Rent_books  
    add foreign key (book_id)  
        references k_Book (book_id);
```

Table K_BOOK

This table contains information about the details of the physical books in the library. Such table is connected to the table *K_rent_books* and *K_title*.

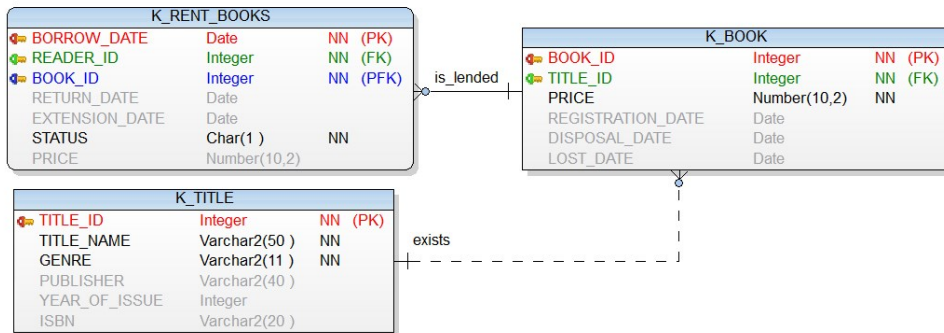


Fig. C.4: *K_book* submodel

Attributes

- ❖ **book_id** – identification key of the book.
 - attribute *book_id* is the primary key of the table.
 - value is the integer.
 - Example:
 - 111
- ❖ **title_id** – identification number of the title of the book.
 - attribute *title_id* is the *foreign key* referencing the *K_title* table.
 - value is the integer number.
 - Example:
 - 1234
- ❖ **price** – information about the price of the book (for how much the book was bought).
 - value is the number composed of 10 digits with two decimal numbers.
 - Example:
 - 150
- ❖ **registration_date** – value is the day of the registration of the book.
 - value has the *Date* data type.
 - Example:
 - 25.6.2015
 - NULL
- ❖ **disposal_date** – value is the day of the disposal of the book.
 - value has the *Date* data type.
 - Example:
 - 25.6.2015
 - NULL
- ❖ **lost_date** – value is the day of the loss of the book.
 - value has the *Date* data type.
 - Example:
 - 25.6.2015
 - NULL

Primary key

The primary key is attribute *book_id*.

Foreign key

Attribute *title_id* is the foreign key to the *K_title* table.

SQL script for table creation

```
Create table k_Book
(
    book_id            Integer        NOT NULL,
    title_id           Integer        NOT NULL,
    price              Number(10, 2)  NOT NULL,
    registration_date   Date,
    disposal_date       Date,
    lost_date          Date,
    primary key (book_id)
);
```

Script for relationship definition

```
Alter table k_Book
add foreign key (title_id)
references k_Title (title_id);
```

Table K_TITLE

This table contains information about the details of the book titles in the library. Such table is connected to the table *K_rent_books* and *K_title*.

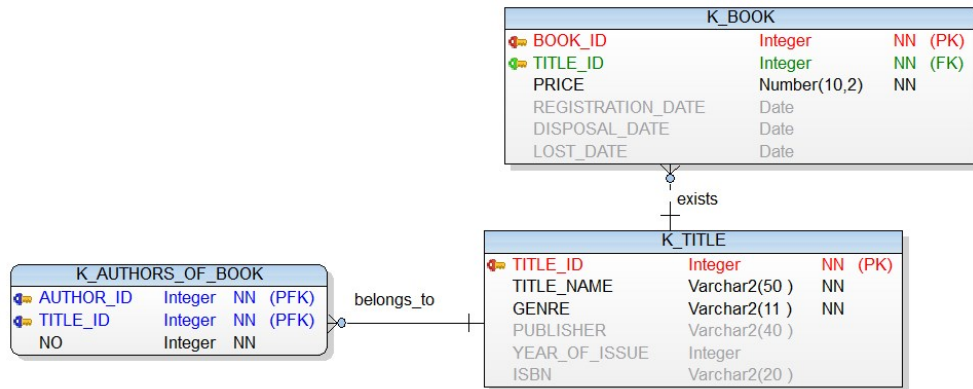


Fig. C.5: *K_title* submodel

Attributes

- ❖ **title_id** – identification number of the title of the book.
 - attribute *title_id* is the *primary key* of the table.
 - value is the integer number.
 - Example:
 - 11111
- ❖ **title_name** – the name of the book.
 - string value composed maximally of 50 characters.
 - Example:
 - Database systems
- ❖ **genre** – category of literature.
 - string value composed maximally of 11 characters.
 - Example:
 - Science
- ❖ **publisher** – the publisher is a commercial name of the publisher.
 - string value composed maximally of 40 characters.
 - Example:
 - Pearson Prentice Hall
- ❖ **year_of_issue** – value expresses the year of the publishing.
 - value is the integer number.
 - Example:
 - 2017
- ❖ **isbn** – the International Standard Book Number (*ISBN*) is a unique numeric commercial book identifier.
 - string value composed maximally of 20 characters.
 - Example:
 - 978-80-554-1311-2

Primary key

The primary key is attribute *title_id*.

Foreign key

The table has no foreign keys.

SQL script for table creation

```
Create table k_Title
(
    title_id          Integer          NOT NULL,
    title_name        Varchar(50)     NOT NULL,
    genre             Varchar(11)     NOT NULL,
    publisher         Varchar(40),
    year_of_issue     Integer,
    isbn              Varchar(20),
    primary key (title_id)
);
```

Script for the relationship definition

None.

Table K_AUTHOR

This table contains information about the details of the authors. Such table is connected to the table *K_authors_of_book*.

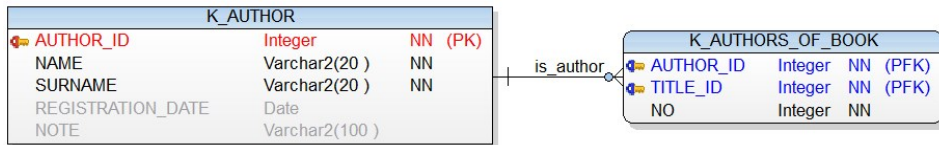


Fig. C.6: *K_author submodel*

Attributes

- ❖ **author_id** – identification number of the author.
 - the attribute is the *primary key* of the table.
 - value is the integer.
 - Example:
 - 11111
- ❖ **name** – first name of the author.
 - value is the string with variable length limited to 20 characters.
 - Example:
 - Karol
- ❖ **surname** – family name of the author.
 - value is the string with variable length limited to 20 characters.
 - Example:
 - Matiaško
- ❖ **registration_date** – value is the day of the registration of the author.
 - value has the *Date* data type.
 - Example:
 - 25.6.2015
 - NULL
- ❖ **note** – some remarks and comments about the author.
 - value is the string with variable length limited to 100 characters.
 - Example:
 - Interested in Computer science

Primary key

The primary key is attribute *author_id*.

Foreign key

The table has no foreign keys.

SQL script for table creation

```
Create table k_Author
(
    author_id          Integer          NOT NULL,
    name               Varchar2(20)    NOT NULL,
    surname            Varchar2(20)    NOT NULL,
    registration_date   Date,
    note               Varchar2(100),
    primary key (author_id)
);
```

Script for the relationship definition

None.

Table K_AUTHORS_OF_BOOK

This table contains information about the associations of the authors to the titles of the book. Such table is connected to the table *K_author* and *K_title*.

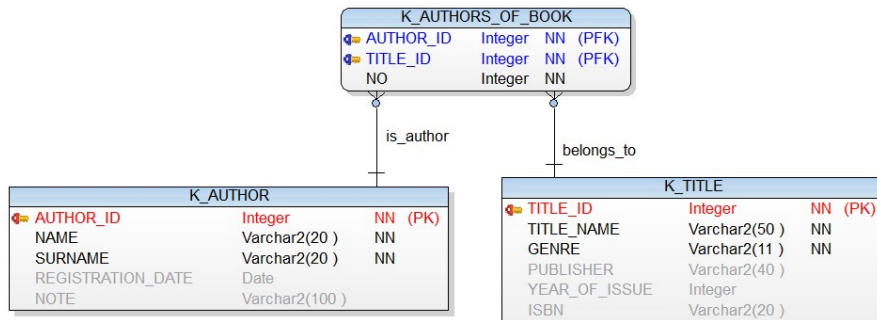


Fig. C.7: *K_authors_of_book* submodel

Attributes

- ❖ **author_id** – identification number of the author.
 - the attribute is the part of the *primary key* of the table.
 - the attribute is the part of the *foreign key* referencing *K_author* table.
 - value is the integer number.
 - Example:
 - 11111
- ❖ **title_id** – identification number of the title of the book.
 - the attribute is the part of the *primary key* of the table.
 - the attribute is the part of the *foreign key* referencing *K_title* table.
 - value is the integer number.
 - Example:
 - 11111
- ❖ **no** – attribute represents the order of the author in the particular title.
 - value is the integer number.
 - Example:
 - 2

Primary key

The primary key is composite, formed by attributes *author_id* and *title_id*.

Foreign key

Attribute *author_id* is the foreign key to the *K_author* table.

Attribute *title_id* is the foreign key to the *K_title* table.

SQL script for table creation

```
Create table k_Rent_books
(
    borrow_date      Date          NOT NULL,
    reader_id        Integer       NOT NULL,
    book_id          Integer       NOT NULL,
    return_date      Date,
    extension_date    Date,
    status           Char(1)       NOT NULL,
    price            Number(10, 2),
    primary key (borrow_date, book_id)
);
```

Script for relationship definition

```
Alter table k_Authors_of_book
add foreign key (author_id)
references k_Author (author_id);

Alter table k_Authors_of_book
add foreign key (title_id)
references k_Title (title_id);
```


APPENDIX D – SYNTAX

```
CREATE USER user_name
IDENTIFIED { BY password | EXTERNALLY | GLOBALLY AS 'CN=user' }
[ DEFAULT TABLESPACE tablespace ]
[ TEMPORARY TABLESPACE tablespace ]
[ QUOTA { number [K|M] | UNLIMITED } ON tablespace ]
  [, QUOTA { number [K|M] | UNLIMITED } ON tablespace ]
[ PROFILE profile_name ]
[ PASSWORD EXPIRE ]
[ { ACCOUNT LOCK | ACCOUNT UNLOCK } ]
```

```
CREATE TABLE [schema.]table_name
[(column_name datatype [DEFAULT expr]
  [ { [column_constraint] } [...]]
  |
  table_constraint
  )
] [...]

column_constraint ::=
[CONSTRAINT constraint_name]
{
  [NOT] NULL
  |
  { UNIQUE | PRIMARY KEY }
  |
  REFERENCES [schema.]table_name [(column_name1 [, column_name2, ...] )]
    [ ON DELETE CASCADE ]
  |
  CHECK (condition)
}

table_constraint ::=
[CONSTRAINT constraint_name]
{
  { UNIQUE | PRIMARY KEY } ( { column_name1 } [, column_name2, ...] )
  |
  FOREIGN KEY (column_name1 [, column_name2, ...])
    REFERENCES [schema.]table_name [ column_name1 [, column_name2,
      ...] ] [ ON DELETE CASCADE ]
  |
  CHECK (condition)
}
```

```
DROP TABLE table_name [ CASCADE CONSTRAINTS | PURGE ];
```

```
PURGE TABLE table_name;
PURGE INDEX index_name;
PURGE RECYCLEBIN;
PURGE TABLESPACE tablespace_name;
PURGE TABLESPACE tablespace_name USER user_name;
```

```
FLASHBACK TABLE table_name TO BEFORE DROP;
```

```
FLASHBACK TABLE table_name TO BEFORE DROP
RENAME TO new_table_name;
```

```
CREATE [UNIQUE] INDEX index_name
  ON table_name (column_name1 [ASC | DESC], ...);
```

```
DROP INDEX index_name;
```

```
CREATE SEQUENCE sequence_name
  [ INCREMENT BY integer_value ]
  [ START WITH integer_value ]
  [ {MAXVALUE integer_value | NOMAXVALUE} ]
  [ {MINVALUE integer_value | NOMINVALUE} ]
  [ {CYCLE | NOCYCLE} ]
  [ {CACHE positive_integer_value | NOCACHE} ]
  [ {ORDER | NOORDER} ];
```

```
ALTER SEQUENCE sequence_name INCREMENT BY integer_value;
ALTER SEQUENCE sequence_name MAXVALUE integer_value;
ALTER SEQUENCE sequence_name {CYCLE | NOCYCLE};
ALTER SEQUENCE sequence_name {CACHE positive_integer_value | NOCACHE};
ALTER SEQUENCE sequence_name {ORDER | NOORDER};
```

```
DROP SEQUENCE sequence_name;
```

```
schema_name.object_name@dblink_name
schema_name.object_name
object_name
```

```
INSERT INTO table_name [(column_list)]
{
  VALUES (list_of_values)
  |
  SELECT-statement
}
```

```
DELETE FROM table_name
  [WHERE conditions];
```

```
UPDATE table_name SET
{
  column_name1 = expression1 [, ...]
  |
  { (column_list)
    |
    *
  } = (expression_list)
}
[WHERE conditions]
```

```

SELECT [ALL | DISTINCT]
  { * | column_name1 | function_name1[(parameters1)] } [, ...]
FROM table_reference1 [table_alias1] [, ...]
  [WHERE conditions]
  [GROUP BY column_list]
  [HAVING conditions]
  [ORDER BY column_list [ASC | DESC], ...]
FROM table_name1 [table_alias1]
  { [ {LEFT | RIGHT | FULL} [OUTER] ] JOIN table_name2
                                     [table_alias2]
    { ON (join_conditions1) | USING(column_list_join1) }
  |
  [INNER] JOIN table_name3 [table_alias3]
    { ON (join_conditions3) | USING(column_list_join3) }

  | {CROSS | NATURAL [INNER]} JOIN table_name4
                                     [table_alias4]
  }

```

```

expression1 relational_operation expression2
expression [NOT] BETWEEN expression1 AND expression2
expression [NOT] IN (item_set)
expression [NOT] LIKE 'string' [ESCAPE escape-character]
expression relational_operation
  {ALL | [ANY | SOME]} (SELECT-statement)
expression [NOT] IN (SELECT-statement)
[NOT] EXISTS (SELECT-statement)
expression IS [NOT] NULL

```

```

SELECT-statement1
  {UNION [ALL] | INTERSECT | MINUS}
SELECT-statement2
  [ {UNION [ALL] | INTERSECT | MINUS}
    SELECT-statement3
  ] ...

```

```

SUBSTR(string, m [, n])
LENGTH(string)
UPPER(string)
LOWER(string)
INITCAP(string)
Operator ||
CONCAT(string1, string2)
INSTR(string, substring, [m [, n]])

```

```

LIKE '%\_%' ESCAPE '\';
%   any number of characters
_   one character only

```

```

ABS(expression)
ROUND(n [, m])
TRUNC(n [, m])

```

ALTER SESSION

```
SET nls_date_format='DD.MM.YYYY HH24:MI:SS';
```

ALTER SESSION

```
SET nls_timestamp_format='DD.MM.YYYY HH24:MI:SS:FF';
```

ALTER SESSION

```
SET nls_date_language='English';
```

ALTER SESSION

```
SET nls_territory='Slovakia'; -- 1 (day number) - Monday
```

ALTER SESSION

```
SET nls_territory= 'America'; -- 1 (day number) - Sunday
```

```
TO_CHAR(date_value, [format [, nls_param]])
```

```
TO_DATE(string_value, [format [, nls_param]])
```

SYSDATE**SYSTIMESTAMP**

```
ADD_MONTHS(d, n)
```

```
NEXT_DAY(d, day_value)
```

```
LAST_DAY(d)
```

```
TRUNC(d [, format])
```

```
ROUND(d [, format])
```

```
EXTRACT(format FROM d)
```

```
MONTHS_BETWEEN(d1, d2)
```

```
COALESCE(expr1, expr2, ..., exprn)
```

```
DECODE(expression, if1, then1 [, ifn, thenn] [, else])
```

```
NVL(expression1, expression2)
```

```
NVL2(expression1, expression2, expression3)
```

```
case expression
```

```
  when value1 then result1
```

```
  [when valuen then valuen] [...]
```

```
  [else result]
```

```
end
```

```
case
```

```
  when condition1 then result1
```

```
  [when conditionn then resultn] [...]
```

```
  [else result]
```

```
end
```

ROWID**USER**

```
row_number() over ( [ partition by expression ]  
                    ORDER BY column_list )
```

```
rank() over ( [ partition by expression ]  
              ORDER BY column_list )
```

```
dense_rank() over ( [ partition by expression ]  
                    ORDER BY column_list )
```

```
GRANT database_privilege_list
```

```
  TO {PUBLIC | list_of_users}
```

```
  [WITH ADMIN OPTION]
```

```
GRANT object_privilege_list ON object_name
```

```
  TO {PUBLIC | list_of_users}
```

```
  [WITH GRANT OPTION]
```

```
REVOKE { privilege_name ON object_name
         |
         database_privilege_name
         |
         role_name }
FROM { PUBLIC | list_of_users }
```

```
CREATE ROLE role_name;
```

```
BEGIN WORK
COMMIT [WORK]
ROLLBACK [WORK]
SAVEPOINT savepoint_name
ROLLBACK TO SAVEPOINT savepoint_name
```

```
IF condition THEN
    statements;
END IF;
```

```
IF condition1 THEN
    statements;
ELSIF condition2 THEN
    statements;
[ELSE
    statements;]
END IF;
```

```
IF condition THEN
    statements;
[ELSE
    statements;]
END IF;
```

```
LOOP
    ...
    IF condition THEN
        EXIT;
    END IF;
    ...
END LOOP;
```

```
LOOP
    ...
    EXIT WHEN condition;
END LOOP;
```

```
WHILE condition LOOP
    statements;
END LOOP;
```

```
FOR variable_name IN min..max LOOP
    statements;
END LOOP;
```

```
FOR variable_name IN REVERSE min..max LOOP
    statements;
END LOOP;
```

```
[DECLARE                                -- variable declaration part
    variable_name data_type [:= init_value];
]
BEGIN
    statements;                        -- execution part
[EXCEPTION                             -- exception processing
    WHEN exception_type1 THEN
        statements;
    WHEN exception_type2 THEN
        statements;
    ...
]
END;
/
```

```
CREATE [OR REPLACE] PROCEDURE procedure_name
    [( parameter1 [ mode1 ] data_type1,
      parameter2 [ mode2 ] data_type2, ... )]
IS|AS
    [ variable_name data_type [:= init_value]; ]
BEGIN
    statements;
    [ EXCEPTION
        WHEN exception_type1 THEN
            statements;
        [WHEN ...]
    ]
END [procedure_name];
/
```

```
CREATE [OR REPLACE] FUNCTION function_name
    [( parameter1 [ mode1 ] datatype1,
      parameter2 [ mode2 ] datatype2, ... )]
RETURN datatype
IS|AS
    [ variable_name data_type [:= init_value]; ]
BEGIN
    statements;
    RETURN expression;
    [ EXCEPTION
        WHEN exception_type1 THEN
            statements;
        [WHEN ...]
    ]
END [function_name];
/
```

```
DROP PROCEDURE procedure_name;
```

```
DROP FUNCTION function_name;
```

```
RAISE_APPLICATION_ERROR(error_code, error_text [, {TRUE | FALSE} ]);
```

```

CREATE [OR REPLACE] TRIGGER [schema.]trigger
{ {BEFORE | AFTER}
  {DELETE | INSERT | UPDATE [ OF column1 [, column2 [, ...]] ]}
  [ OR {DELETE | INSERT | UPDATE [ OF column1 [, column2 [, ...]] ] } ]}
  [...]
  |
  INSTEAD OF {DELETE | INSERT | UPDATE}
}
ON [schema.][table_name | view_name]
[ REFERENCING { OLD [AS] renamed_old | NEW [AS] renamed_new} ]
[ FOR EACH ROW ]
[ WHEN (condition) ]
    Trigger_body

```

```

ALTER TRIGGER [schema.]trigger_name {ENABLE | DISABLE};

```

```

ALTER TABLE [schema.]table_name {ENABLE | DISABLE}
    ALL TRIGGERS;

```

```

DROP TRIGGER [schema.]trigger_name;

```

```

CREATE [OR REPLACE] [FORCE | NOFORCE]
VIEW [schema.]view_name [(column_alias1 [, ...])]
    AS Select_statements
    [WITH [ READ ONLY | CHECK OPTION [CONSTRAINT constraint_def] ]]

```

```

SELECT column_list | function_calls | expressions
    INTO variable_list
    FROM table_list

```

```

SELECT expr1, expr2 ..., exprn
    BULK COLLECT INTO var1, var2 ..., varn
    FROM table_list

```

```

OPEN cursor_name;
FETCH cursor_name INTO list_of_variables;
CLOSE cursor_name;

```





```

cursor_name%ISOPEN
cursor_name%FOUND
cursor_name%NOTFOUND
cursor_name%ROWCOUNT


```


APPENDIX E – FILE MANAGEMENT

Tab. E.1: Storage

<i>File name</i>	<i>Location</i>
All materials	https://gofile.me/4voWB/07zl894BI 
Data models	https://gofile.me/4voWB/4HLYQykKg 
Flight	https://gofile.me/4voWB/ToX7d1Yc8 
Library	https://gofile.me/4voWB/avlOV2nhj 

<i>File name</i>	<i>Location</i>
Student	https://gofile.me/4voWB/XTdRsaluR 
exp_flight.exp	https://gofile.me/4voWB/PRwipZIXD 
exp_library.exp	https://gofile.me/4voWB/kcgfvaEpO 
exp_music.exp	https://gofile.me/4voWB/13teiA8f0 
exp_student_ENG.exp	https://gofile.me/4voWB/L5q4hB17p 

<i>File name</i>	<i>Location</i>
family_tree_script.txt	https://gofile.me/4voWB/G2QwSAT4f 
library_part.txp	https://gofile.me/4voWB/PVReZt0xE 
student_pref_script.sql	https://gofile.me/4voWB/98J8QZgv7 
SQL_load_library.zip	https://gofile.me/4voWB/Agb9Fuggz 

doc. Ing. Michal Kvet, PhD., prof. Ing. Karol Matiaško, PhD., Ing. Štefan Toth, PhD.

PRACTICAL SQL FOR ORACLE CLOUD

Copyright © University of Žilina

Printed by EDIS-Publishing House of the University of Žilina, 2022

First edition, AA 32,40

Number of copies 100 USB

ISBN 978-80-554-1880-3

This proposed textbook is the first edition intended for the students and practitioners to increase practical knowledge and skills in the area of Database systems. The content of the book is prepared based on our experiences with the education of the Database systems at the University of Žilina, Faculty of Management Science and Informatics, as well as discussion with the project consortium and Oracle Academy.

Each chapter contains a short description of the theory, examples, and tasks for evaluating the received knowledge. All tasks and the proposed solutions are critically discussed.

The book aims is to provide complex self-teaching material for the SQL and PL/SQL, with emphasis on data modeling, data integrity and user reports.

The selected environment is the Oracle Cloud, which provides robust autonomous database processing solution with no need for installation, configuration, and administration. It highlights the benefits of the Free Tier and Always Free option.

We hope that such textbook will drive you through the complexity of the SQL and PL/SQL language and you will enjoy the study.

