CSCI 3410 - Database Systems
Lecture Notes (Draft)

Clément Aubert

February 22, 2019 (03:40:04 PM)
# Contents

## List of Problems

<table>
<thead>
<tr>
<th>Problem</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.1 Domains, Attributes, Tuples and Relations</td>
<td>19</td>
</tr>
<tr>
<td>1.1.2 Characteristics of Relations</td>
<td>20</td>
</tr>
<tr>
<td>1.1.3 Notation</td>
<td>20</td>
</tr>
<tr>
<td>2.2.1 Types of constraints</td>
<td>20</td>
</tr>
<tr>
<td>2.2.2 Keys</td>
<td>21</td>
</tr>
<tr>
<td>2.2.3 Foreign Keys</td>
<td>21</td>
</tr>
<tr>
<td>2.2.4 Example</td>
<td>22</td>
</tr>
</tbody>
</table>

## Preamble

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>How to use this guide</td>
<td>7</td>
</tr>
<tr>
<td>Planned Schedule</td>
<td>7</td>
</tr>
<tr>
<td>Previous Exams</td>
<td>8</td>
</tr>
<tr>
<td>Fall 2017</td>
<td>8</td>
</tr>
<tr>
<td>Spring 2018</td>
<td>8</td>
</tr>
<tr>
<td>Fall 2019</td>
<td>9</td>
</tr>
<tr>
<td>Typesetting and Acknowledgments</td>
<td>9</td>
</tr>
<tr>
<td>Various Resources</td>
<td>10</td>
</tr>
<tr>
<td>Copyright</td>
<td>10</td>
</tr>
<tr>
<td>Softwares</td>
<td>10</td>
</tr>
</tbody>
</table>

## 1 Introduction

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resources</td>
<td>11</td>
</tr>
<tr>
<td>1.1 Database</td>
<td>11</td>
</tr>
<tr>
<td>1.2 Database Management System (DBMS)</td>
<td>11</td>
</tr>
<tr>
<td>1.3 Subtasks</td>
<td>11</td>
</tr>
<tr>
<td>1.4 Design</td>
<td>13</td>
</tr>
<tr>
<td>1.5 An Example</td>
<td>13</td>
</tr>
<tr>
<td>1.5.1 Structure</td>
<td>14</td>
</tr>
<tr>
<td>1.5.2 Interactions</td>
<td>14</td>
</tr>
<tr>
<td>1.5.3 Organization</td>
<td>14</td>
</tr>
<tr>
<td>1.5.4 How is a database conceived?</td>
<td>15</td>
</tr>
<tr>
<td>1.6 Characteristics of the Database Approach</td>
<td>15</td>
</tr>
<tr>
<td>Exercises</td>
<td>15</td>
</tr>
<tr>
<td>Solution to Exercises</td>
<td>16</td>
</tr>
<tr>
<td>Problems</td>
<td>17</td>
</tr>
<tr>
<td>Solution to Selected Problems</td>
<td>17</td>
</tr>
</tbody>
</table>

## 2 The Relational Model

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resources</td>
<td>19</td>
</tr>
<tr>
<td>2.1 Concepts</td>
<td>19</td>
</tr>
<tr>
<td>2.1.1 Domains, Attributes, Tuples and Relations</td>
<td>19</td>
</tr>
<tr>
<td>2.1.2 Characteristics of Relations</td>
<td>20</td>
</tr>
<tr>
<td>2.1.3 Notation</td>
<td>20</td>
</tr>
<tr>
<td>2.2 Constraints</td>
<td>20</td>
</tr>
<tr>
<td>2.2.1 Types of constraints</td>
<td>20</td>
</tr>
<tr>
<td>2.2.2 Keys</td>
<td>21</td>
</tr>
<tr>
<td>2.2.3 Foreign Keys</td>
<td>21</td>
</tr>
<tr>
<td>2.2.4 Example</td>
<td>22</td>
</tr>
</tbody>
</table>
### Contents

2.3 Transactions and Operations .......................................... 22
  2.3.1 Insert / Delete / Update ........................................ 22
  2.3.2 Dealing with violations ........................................ 23
Exercises ............................................................................ 23
Solution to Exercises ......................................................... 25
Problems ............................................................................ 27
Solution to Selected Problems ............................................... 29

3 The SQL Programming Language ........................................... 30
  Resources ........................................................................... 30
  3.1 Actors ............................................................................. 30
    3.1.1 Technologies .......................................................... 30
    3.1.2 SQL ........................................................................ 31
  3.2 First Commands ............................................................. 32
  3.3 Overview of Constraints ................................................... 33
  3.4 Foreign Keys ..................................................................... 36
    3.4.1 A first example ......................................................... 36
    3.4.2 Foreign keys restrictions ............................................ 37
    3.4.3 Constructing and populating a new example .................. 39
  3.5 A First Look at Conditions ............................................... 41
  3.6 Three-Valued Logic .......................................................... 42
    3.6.1 Meaning of NULL ...................................................... 42
    3.6.2 Comparison with unknown values {truth_tables} ........... 43
  3.7 Various Tools ..................................................................... 43
    3.7.1 Transactions ............................................................. 43
    3.7.2 DISTINCT / ALL ....................................................... 44
    3.7.3 UNION ...................................................................... 44
    3.7.4 ORDER BY ............................................................... 44
    3.7.5 Aggregate functions .................................................. 44
    3.7.6 Aliases for columns ................................................... 44
  3.8 More Select Queries .......................................................... 45
    3.8.1 Select-project-join ..................................................... 45
    3.8.2 Aliasing tuples ......................................................... 46
    3.8.3 Nested queries .......................................................... 46
  3.9 Setting Up Your Work Environment ..................................... 47
    3.9.1 Installation ............................................................... 47
    3.9.2 Creating an user ....................................................... 50
    3.9.3 Loging-in as testuser .................................................. 51
    3.9.4 Creating our first database ........................................ 51
Exercises ............................................................................ 52
Solution to Exercises ......................................................... 55
Problems ............................................................................ 57
Solution to Selected Problems ............................................... 68

4 Designing a Good Database ................................................. 81
  Resources ........................................................................... 81
  4.1 Interest for High-Level Design .......................................... 81
  4.2 Entity-Relationship Model ............................................... 82
    4.2.1 Entities ................................................................. 82
    4.2.2 Relationships .......................................................... 84
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2.3</td>
<td>Weak entity types</td>
<td>89</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Alternative notation</td>
<td>90</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Enhanced Entity–Relationship Model</td>
<td>91</td>
</tr>
<tr>
<td>4.2.6</td>
<td>Reverse Engineering</td>
<td>91</td>
</tr>
<tr>
<td>4.3</td>
<td>E.R.-to-Relational Models Mapping</td>
<td>92</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Intro</td>
<td>92</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Algorithm</td>
<td>92</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Outro</td>
<td>93</td>
</tr>
<tr>
<td>4.4</td>
<td>Guidelines and Normal Form</td>
<td>94</td>
</tr>
<tr>
<td>4.4.1</td>
<td>General rules</td>
<td>94</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Example</td>
<td>95</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Functional dependencies</td>
<td>95</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Normal forms and keys</td>
<td>97</td>
</tr>
<tr>
<td>4.5</td>
<td>Unified Modeling Language Diagrams</td>
<td>100</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Overview</td>
<td>100</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Types of diagrams</td>
<td>101</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Zoom on classes diagrams</td>
<td>102</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Missing?</td>
<td>104</td>
</tr>
<tr>
<td>Exercises</td>
<td></td>
<td>104</td>
</tr>
<tr>
<td>Solution to Exercises</td>
<td></td>
<td>109</td>
</tr>
<tr>
<td>Problems</td>
<td></td>
<td>113</td>
</tr>
<tr>
<td>Solution to Selected Problems</td>
<td></td>
<td>121</td>
</tr>
<tr>
<td>5</td>
<td>Databases Applications</td>
<td>126</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview</td>
<td>126</td>
</tr>
<tr>
<td>5.2</td>
<td>Java’s Way</td>
<td>127</td>
</tr>
<tr>
<td>5.3</td>
<td>Flash Intro to Java</td>
<td>127</td>
</tr>
<tr>
<td>5.4</td>
<td>A First Program</td>
<td>128</td>
</tr>
<tr>
<td>5.4.1</td>
<td>The Java code</td>
<td>128</td>
</tr>
<tr>
<td>5.4.2</td>
<td>The database</td>
<td>129</td>
</tr>
<tr>
<td>5.4.3</td>
<td>The result</td>
<td>130</td>
</tr>
<tr>
<td>5.4.4</td>
<td>A variation</td>
<td>130</td>
</tr>
<tr>
<td>5.5</td>
<td>Mapping Datatypes</td>
<td>131</td>
</tr>
<tr>
<td>5.6</td>
<td>Differences Between executeQuery, executeUpdate and execute</td>
<td>131</td>
</tr>
<tr>
<td>5.7</td>
<td>A Second Program</td>
<td>131</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Passing options</td>
<td>131</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Creating a table</td>
<td>132</td>
</tr>
<tr>
<td>5.7.3</td>
<td>Inserting values</td>
<td>132</td>
</tr>
<tr>
<td>5.7.4</td>
<td>Prepared statements</td>
<td>132</td>
</tr>
<tr>
<td>5.7.5</td>
<td>Advanced Statements objects</td>
<td>132</td>
</tr>
<tr>
<td>Exercises</td>
<td></td>
<td>132</td>
</tr>
<tr>
<td>Solution to Exercises</td>
<td></td>
<td>134</td>
</tr>
<tr>
<td>5.8</td>
<td>Problems</td>
<td>135</td>
</tr>
<tr>
<td>Solution to Selected Problems</td>
<td></td>
<td>142</td>
</tr>
<tr>
<td>6</td>
<td>A Bit About Security</td>
<td>143</td>
</tr>
<tr>
<td>6.1</td>
<td>Usual Aspects</td>
<td>143</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Threat model</td>
<td>143</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Control measures</td>
<td>143</td>
</tr>
<tr>
<td>6.2</td>
<td>Specificities Of Databases</td>
<td>143</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Attack</td>
<td>143</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Protections</td>
<td>144</td>
</tr>
<tr>
<td>7</td>
<td>Presentation of NoSQL</td>
<td>145</td>
</tr>
<tr>
<td>7.1</td>
<td>A Bit of History</td>
<td>145</td>
</tr>
<tr>
<td>7.1.1</td>
<td>DB applications and application DB</td>
<td>145</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Clusters, clusters...</td>
<td>145</td>
</tr>
<tr>
<td>7.1.3</td>
<td>A first shift</td>
<td>146</td>
</tr>
<tr>
<td>7.1.4</td>
<td>Gathering forces</td>
<td>146</td>
</tr>
<tr>
<td>7.1.5</td>
<td>The future?</td>
<td>146</td>
</tr>
<tr>
<td>7.2</td>
<td>Comparison</td>
<td>147</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Overview</td>
<td>147</td>
</tr>
<tr>
<td>7.2.2</td>
<td>ACID Vs CAP</td>
<td>147</td>
</tr>
<tr>
<td>7.3</td>
<td>Categories of NoSQL Systems</td>
<td>147</td>
</tr>
<tr>
<td>7.4</td>
<td>MongoDB</td>
<td>148</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Resources</td>
<td>148</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Introduction</td>
<td>149</td>
</tr>
<tr>
<td>7.4.3</td>
<td>Document-oriented database</td>
<td>149</td>
</tr>
<tr>
<td>7.4.4</td>
<td>General organization of MongoDB databases</td>
<td>150</td>
</tr>
<tr>
<td>7.4.5</td>
<td>First elements of syntax</td>
<td>151</td>
</tr>
<tr>
<td>7.4.6</td>
<td>Set up</td>
<td>152</td>
</tr>
<tr>
<td>7.4.7</td>
<td>Example</td>
<td>152</td>
</tr>
<tr>
<td>7.5</td>
<td>Principles</td>
<td>153</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>153</td>
</tr>
<tr>
<td></td>
<td>Solution to Exercises</td>
<td>154</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>155</td>
</tr>
</tbody>
</table>
List of Problems

1.1 A database catalog for a campus ........................................... 17
2.1 Finding candidate key in a CLASS relation .............................. 27
2.2 Relational model for a cinema company ................................. 28
2.3 Relational model for bills .................................................... 28
3.1 Discovering the documentation ............................................ 57
3.2 Creating and using a simple table in SQL ............................... 58
3.3 Duplicate rows in SQL ......................................................... 59
3.4 Revisiting the PROF table .................................................... 59
3.5 TRAIN table and more advanced SQL coding .......................... 60
3.6 Insert, Select and Project in the COFFEE database .................... 62
3.7 Select queries for the COMPUTER table ................................. 64
3.8 Improving a role-playing game ............................................. 65
3.9 A simple database for books ............................................... 66
4.1 Design for your professor ..................................................... 113
4.2 Reading the MOVIES database ER schema ............................ 113
4.3 ER diagram for car insurance .............................................. 114
4.4 ER-to-Relation mapping for car insurance .............................. 115
4.5 ER-to-Relation mapping for Country .................................... 115
4.6 Normal form of a CAR_SALE relation ................................... 115
4.7 Normalizing the FLIGHT relation ........................................ 116
4.8 From business statement to dependencies .............................. 116
4.9 Normalization ................................................................. 116
4.10 Normal form of the BOOK relation ...................................... 117
4.11 From business statements to E.R. diagram – UNIVERSITY ........ 117
4.12 From UML to relational model – DRIVER ............................. 117
4.13 Normal form of the CONTACT relation ................................. 118
4.14 CONSULTATION relation: justification, primary key and normal form .......................... 118
4.15 From E.R. to relational schema and UML class diagram – CAR_INFO .......................... 119
4.16 Discovering MySQL Workbench ........................................ 119
4.17 Using MySQL Workbench’s reverse engineering ....................... 120
4.18 From business statements to dependencies – KEYBOARD ........ 120
4.19 Reverse engineering by hand .............................................. 121
5.1 MySQL’s batch mode and HW_EBOOKSHOP.sql ...................... 135
5.2 First Database Application .................................................. 135
Preamble

How to use this guide

These lecture notes are written in an elusive style: their are a support for the explanations I will be making at the board. They are not designed to be self-contained: they are rather a list of topics and reminders, along with handy examples, code and drawings. Reading them before coming to the lecture will help you getting a sense of the next topic we will be discussing, but you may sometimes have trouble deciphering their ... unique style.

When it comes to code, you can normally copy-and-paste it from the document and use it as it is. Some portion of code starts with a path in comment: you can browse the source code at https://rocketgit.com/user/caubert/CSCI_3410/source/tree/branch/master/tree/notes/code to download it directly.

On top of the notes, you will find in this document:

- References, at the very end of this document
- and for each chapter,
  - A list of additional resources,
  - A list of short exercises,
  - Solution to those exercises,
  - A list of problem,
  - Sometimes, solution to some of those problems.

Any feedback is greatly appreciated. Please refer to http://spots.augusta.edu/caubert/db/ln/README.html#contributing for how to contribute to those notes.

The syllabus is at http://spots.augusta.edu/caubert/db/, and the webpage for this notes is at http://spots.augusta.edu/caubert/db/ln/.

Planned Schedule

A typical (meeting twice a week, ±17 weeks, ±30 classes) semester is divided as follows:

- Lecture 1: Presentation and Syllabus
- Lecture 2–3: Introduction
- Lecture 4–5: The Relational Model
- Lecture 6–9: The SQL Programming Language
- Lecture 10–11: Review session and Exam #1
- Lecture 12: Return on Exam + Introduction to High-Level Design
- Lecture 13–15: Entity-Relationship Model
- Lecture 16: E.R.-to-Relational Models Mapping
- Lecture 17–20: Guidelines and Normal Form
- Lecture 23–24: Review session and Exam #2
• Lecture 25: Intro to Database Programming and to Java
• Lecture 26–28: Database Applications
• Lecture 29: Introduction to NoSQL
• Lecture 30: Introduction to MongoDB

For information purposes, an indication like this:

marks the (usual) separation between two lectures.

**Previous Exams**

To give you a sense of what you will be asked to do during the exams, please find below a description of the exams given previous semesters.

**Fall 2017**

• Exam #1:
  - Six small exercises (Exercise 1.8, Exercise 2.4, Exercise 2.8, Exercise 3.5, Exercise 3.7 and Exercise 3.10)
  - Problem 2.1 (Finding candidate key in a CLASS relation)
  - Problem 2.2 (Relational model for a cinema company)
  - A variation on (Elmasri and Navathe 2010, Exercise 3.11), (Elmasri and Navathe 2015, Exercise 5.11)
  - Problem 3.5 (TRAIN table and more advanced SQL coding)

• Exam #2:
  - Six small exercises,
  - Problem 4.11 (From business statements to E.R. diagram – UNIVERSITY)
  - Problem 4.8 (From business statement to dependencies)
  - Problem 4.13 (Normal form of the CONTACT relation)
  - A variation on Problem 4.5 (ER-to-Relation mapping for Country)

• Final:
  - A variation on (Exercise 5.10)
  - A variation on Problem 3.6 (Insert, Select and Project in the COFFEE database)
  - A variation on Problem 4.12 (From UML to relational model – DRIVER): students were asked to draw the ER diagram for that schema.
  - Problem 4.10 (Normal form of the BOOK relation)
  - Problem 4.14 (CONSULTATION relation: justification, primary key and normal form)

**Spring 2018**

• Exam #1:
  - Problem 2.2 (Relational model for a cinema company), except that I gave some of the relations and attributes, to help getting started with the problem.
  - Problem 4.1 (Design for your professor)
  - Problem 3.5 (TRAIN table and more advanced SQL coding)
  - Problem 3.6 (Insert, Select and Project in the COFFEE database)

• Exam #2:

---

1This exam was probably a bit too long, but students managed it pretty well.
- A variation on Problem 4.13 (Normal form of the CONTACT relation)
- A variation on Problem 4.14 (CONSULTATION relation: justification, primary key and normal form)
- Problem 3.8 (Improving a role-playing game)
- A variation on Problem 4.18 (From business statements to dependencies – KEYBOARD)
- Problem 4.5 (ER-to-Relation mapping for Country)

• Final:
  - Take the relational model of the solution of Problem 2.2 (Relational model for a cinema company), and “reverse-engineer” it to obtain a ER diagram (this problem was harder than I expected)
  - Six small exercises (Exercise 4.20, Exercise 7.1, Exercise 7.2, Exercise 7.3, Exercise 7.4)
  - Problem 4.7 (Normalizing the FLIGHT relation)
  - A variation on Problem 4.11 (From business statements to E.R. diagram – UNIVERSITY)
  - A variation on Problem 4.12 (From UML to relational model – DRIVER): students were asked to draw the ER diagram for that schema.

Fall 2019

• Exam #1:
  - Problem 3.9 (A simple database for books)
  - Five exercises (Exercise 1.6, Exercise 2.12, Exercise 3.29, Exercise 2.13, Exercise 3.36)
  - A variation on Problem 2.3 (Relational model for bills)

Typesetting and Acknowledgments

The source code for those notes is hosted at rocketgit\(^2\), typeset in markdown, and then compiled using pandoc\(^3\) and multiple filters (pandoc-numbering\(^4\), pandoc-citeproc\(^5\), pandoc-include-code\(^6\)). The document uses Linux Libertine fonts\(^7\)\(^8\), and the drawings use various LaTeX\(^9\) packages, including PGF, TikZ\(^10\), tikz-er\(^11\) and tikz-dependency\(^12\). The help from the TeX - LaTeX Stack Exchange\(^13\) community greatly improved this document. The underline text is obtained from https://yaytext.com/underline/, the unicode symbols are searched in http://milde.users.sourceforge.net/LUCR/Math/unimathsymbols.html.

To clone and compile this document, refer to the README.md file, at https://rocketgit.com/user/caubert/CSCI_3410/source/tree/branch/master/blob/README.md or http://spots.augusta.edu/caubert/db/ln/README.html.

\(^2\)https://rocketgit.com/user/caubert/CSCI_3410
\(^3\)http://pandoc.org/
\(^4\)https://github.com/chdemko/pandoc-numbering
\(^5\)https://github.com/jgm/pandoc-citeproc
\(^6\)https://github.com/owickstrom/pandoc-include-code
\(^7\)http://libertine-fonts.org/
\(^8\)At least, the pdf version.
\(^9\)https://www.latex-project.org/
\(^10\)https://sourceforge.net/projects/pgf/
\(^11\)https://bitbucket.org/pavel_calado/tikz-er2/raw/da9f9f7f169647cad6d91d7975400b1605ae67a/tikz-er2.sty
\(^12\)https://ctan.org/pkg/tikz-dependency
\(^13\)https://tex.stackexchange.com/
Various Resources

On top of the references and of the “resources” listed at the beginning of each chapter, I consulted while writing those notes:

- https://www.1keydata.com/datawarehousing/data-modeling-levels.html
- https://creately.com/blog/diagrams/class-diagram-relationships/
- https://www.ntu.edu.sg/home/ehchua/programming/java/JDBC_Basic.html

Copyright

This work is under Creative Commons Attribution 4.0 International License\textsuperscript{14} or later.

Some figures and ressources are borrowed from other sources, like (Elmasri and Navathe 2010), (Elmasri and Navathe 2015), or the resources mentionned previously, in which case it is indicated clearly.

Softwares

I would like here to include some of the softwares I believe could make your semester easier, but still have to work on it.

\textsuperscript{14}https://creativecommons.org/licenses/by/4.0/
1 Introduction

Resources

- (Elmasri and Navathe 2010, ch. 1.1–1.6)
- (Elmasri and Navathe 2015, ch. 1.1–1.6)

1.1 Database

A database (DB) is a collection of related data. Data (= information, can be anything, really) + management (= logical organization of the data), through Database Management System).

1. Represent a mini-world, a Universe of Disclosure (UoD).
2. Logically coherent, with a meaning.
3. Populated for a purpose.

Refer to the "A simplified database environment" figure, where

- The program can be written in any language, be a web interface, etc.
- Most DBMS software include a Command-Line Interface (CLI),
- Sometimes, meta-data and data are closer than pictured (you can have “self-describing meta-data”, that is, they can’t be distinguished). Note also that “meta-data” has numerous definition (“data about the data”): we use it here to refer to the description of the organization of the data, and not e.g. statistical data about our data.

1.2 Database Management System (DBMS)

A DBMS is a general purpose software that is used to

1. Define (= datatype, constraints, structures, etc.)
2. Construct / Create (= storing the data)
3. Manipulate / Maintain (= query, update, etc.)
4. Share / Control access (= among users, softwares.)

You can think of a tool to

1. specify a storage unit,
2. fill it,
3. allow to change its content, as well as its organization,
4. allow multiple users to access it at the same time.

1.3 Subtasks

1. Organization (DB designer)
Figure 1.1: A simplified database environment
2. Modification, retrieval (end-user)
3. Maintenance (DB administrator)
4. Software engineer, web developer, programers, ...

The focus will be on design, but we will have to do a little bit of everything.

1.4 Design

Refer to the “The cycle of design” figure.

Add a line from SQL to Program, a line from NoSQL to CLI. Make the arrow from Bus. Statements to NoSQL dashed?

---

1.5 An Example

<table>
<thead>
<tr>
<th>Name</th>
<th>Student_number</th>
<th>Class</th>
<th>Major</th>
</tr>
</thead>
<tbody>
<tr>
<td>Morgan</td>
<td>18</td>
<td>2</td>
<td>IT</td>
</tr>
<tr>
<td>Bob</td>
<td>17</td>
<td>1</td>
<td>CS</td>
</tr>
</tbody>
</table>
1.5 An Example

<table>
<thead>
<tr>
<th>Course_name</th>
<th>Course_number</th>
<th>Credit_hours</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intro. to CS</td>
<td>1301</td>
<td>4</td>
<td>CS</td>
</tr>
<tr>
<td>DB Systems</td>
<td>3401</td>
<td>3</td>
<td>CS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section_identifier</th>
<th>Course_num</th>
<th>Semester</th>
<th>Year</th>
<th>Instructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>2910</td>
<td>1301</td>
<td>Fall</td>
<td>2019</td>
<td>Kate</td>
</tr>
<tr>
<td>9230</td>
<td>2103</td>
<td>Spring</td>
<td>2020</td>
<td>Todd</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Student_number</th>
<th>Section_identifier</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>2910</td>
<td>A</td>
</tr>
<tr>
<td>18</td>
<td>2910</td>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Course_number</th>
<th>Prerequisite_number</th>
</tr>
</thead>
<tbody>
<tr>
<td>2910</td>
<td>1301</td>
</tr>
<tr>
<td>1302</td>
<td>1301</td>
</tr>
</tbody>
</table>

1.5.1 Structure

- Database structure and records, 5 files (=collection of records), each containing data records of the same type. **Persistent storage**
- Each record has a structure, different data elements, each has a data type.
- Records have relationships between them.

1.5.2 Interactions

- Can I retrieve the name of 1301? Can I know what classes Kate is teaching this semester? Can I know what instructor Bob had?
- Queries, updates, removal, addition of records. **Efficiency** (using auxiliary files (indexes), optimization)
- Selection (for any operation) requires care: do we want all the records, some of them, exactly one?

1.5.3 Organization

Why are the files separated like that? Why don’t we store the section with the course with the students?
• **Avoiding redundancy** ("data normalization"), or having it controlled
• **Levels of access** (multiple user interface)
• And we still have the same usability!

But need to be careful about **consistency / referential integrity**.

### 1.5.4 How is a database conceived?

- Specification and analysis. "Each student number will be unique, but they can have the same name. We want to access the letter grade, but not the numerical grade", etc.
- Conceptual design
- Logical design
- Physical design

Gradation, from really abstract specification that is easy to modify, to more solidified description of what needs to be coded. We’ll see when we’ll study high-level models what that means. It’s easier to move things around early in the conception.

### 1.6 Characteristics of the Database Approach

1. A database is more than just data: it also contains a complete description of the structure and constraints. ⇒ We have Data & Meta-data (we can also have self-describing data, where meta-data and data are interleaved).
2. Data-abstraction: DBMS provides a conceptual representation, and hides implementation details.
   a) Program-data independence: changing the database doesn’t require to change the DBMS. Compare with changing a custom data-type in a program.
   b) Program-operation independence: an operation has an interface (or signature) and an implementation (or method)
3. Support of multiple views of the data: view is a subset of the database, or virtual data.
4. Sharing and multiuser transaction processing: concurrency control using transactions (= series of instructions that is supposed to execute a logically correct database access if executed in its entirety). Isolation, atomicity (all or nothing): ACID principles, that we will study later on.

### Exercises

**Exercise 1.1** Is a pile of trash a database? Why, or why not?

**Exercise 1.2** Which one comes first, the physical design, the conceptual design, or the logical design?

**Exercise 1.3** What is the difference between a database and the meta-data of the database?

**Exercise 1.4** Do I have to change my DBMS if I want to change the structure of my data?

**Exercise 1.5** What is a virtual data? How can I access it?

**Exercise 1.6** Name two DBMS.

**Exercise 1.7** Why do DBMS include concurrency control?

**Exercise 1.8** What is independence between program and data? Why does it matter?
Exercise 1.9 Define the word “miniworld”.

Exercise 1.10 Expand the acronym “DBMS”.

Exercise 1.11 Name the four different kinds of action that can be performed on data.

Exercise 1.12 Assume that I have a file where one record corresponds to one student. Should the information about the classes a student is taking (e.g. room, instructor, code, etc.) being stored in the same file? Why, or why not?

Solution to Exercises

Solution 1.1 No, because it lacks a logical structure.

Solution 1.2 The conceptual design.

Solution 1.3 The data is the information we want to store, the meta-data is its organization, how we are going to store it. Meta-data is information about the data, but of no use on its own.

Solution 1.4 Normally no, data and programs are independent. But actually, this is true only if the model doesn’t change: shifting to a “less structured model”, e.g., one of the NoSQL models, can require to change the DBMS.

Solution 1.5 It is a set of information that is derived from the database but not directly stored in it. It is accessed through queries.

Solution 1.6 Oracle RDBMS, IBM DB2, Microsoft SQL Server, MySQL, PostgreSQL, Microsoft Access, etc., are valid answers. Are not valid “SQL”, “NoSQL”, “Relational Model”, or such: we are asking for the names of actual softwares!

Solution 1.7 To ensure that several users trying to update the same data will do so in a controlled manner. To avoid inconsistency.

Solution 1.8 The application should not be sensible to the “internals” of the definition and organization of the data. It matters because having this independance means that changing the data will not require to change the programs.

Solution 1.9 The mini-world is the part of the universe we want to represent in the database. It is supposed to be meaningful and will serve a purpose.

Solution 1.10 Database Management System

Solution 1.11 The four actions are:

- Add / Insert
- Update / Modify
- Search / Query
- Delete / Remove

Solution 1.12 If we were to store all the information about the classes in the student records, then we would have to store it as many times as its number of students! It is better to store it in a different file, and then to “link” the two files, to avoid redundancy.
1.6 Characteristics of the Database Approach

Problems

Problem 1.1 (A database catalog for a campus) We want to define a CAMPUS database organized into three files as follows:

- A BUILDING file storing the name and GPS coordinates of each building.
- A ROOM file storing the building, number and floor of each room.
- A PROF file storing the name, phone number, email and room number where the office is located for each professor.

Pb 1.1 – Question 1 A database catalog is made of two part: a table containing the relations’ name and their number of columns, and a table containing the columns’ name, their data type, and the relation to which they belong. (You can see an example in (Elmasri and Navathe 2010, Figure 1.3), or (Elmasri and Navathe 2015, Figure 1.3)). Write the database catalog corresponding to the CAMPUS database.

Pb 1.1 – Question 2 Invent data for such a database, with two buildings, three rooms and two professors.

Pb 1.1 – Question 3 Answer the following, assuming all the knowledge you have of the situation comes from the CAMPUS database, which is an up-to-date and accurate representation of its miniworld:

1. Is it possible to list all the professors?
2. Is it possible to tell in which department is a professor?
3. Is it possible to get the office hours of a professor?
4. Is it possible to list all the professors whose offices are in the same building?
5. Is it possible to list all the rooms?
6. If a new professor arrives, and has to share his office with another professor, do you have to revise your database catalog?
7. Can you list which professors are at the same floor?
8. Can you tell which professor has the highest evaluations?

Solution to Selected Problems

Solution to Pb 1.1 Pb 1.1 – Solution to Q. 1 The database catalog should be similar to the following:

RELATIONS

<table>
<thead>
<tr>
<th>Relation_name</th>
<th>No_of_columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUILDING</td>
<td>3</td>
</tr>
<tr>
<td>ROOM</td>
<td>3</td>
</tr>
<tr>
<td>PROF</td>
<td>4</td>
</tr>
</tbody>
</table>

COLUMNS

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Data_type</th>
<th>Belongs_to_relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building_Name</td>
<td>Character(30)</td>
<td>Building</td>
</tr>
<tr>
<td>GPSLat</td>
<td>Decimal(9,6)</td>
<td>Building</td>
</tr>
<tr>
<td>GPSLon</td>
<td>Decimal(9,6)</td>
<td>Building</td>
</tr>
</tbody>
</table>
### Column Name Data Type Belongs to Relation

<table>
<thead>
<tr>
<th>Column_name</th>
<th>Data_type</th>
<th>Belongs to_relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building_Name</td>
<td>Character(30)</td>
<td>ROOM</td>
</tr>
<tr>
<td>Room_Number</td>
<td>Integer(1)</td>
<td>ROOM</td>
</tr>
<tr>
<td>Floor</td>
<td>Integer (1)</td>
<td>ROOM</td>
</tr>
<tr>
<td>Prof_Name</td>
<td>Character (30)</td>
<td>PROF</td>
</tr>
<tr>
<td>Phone</td>
<td>Integer (10)</td>
<td>PROF</td>
</tr>
<tr>
<td>Email</td>
<td>Character (30)</td>
<td>PROF</td>
</tr>
<tr>
<td>Room_Number</td>
<td>Integer (1)</td>
<td>PROF</td>
</tr>
</tbody>
</table>

**Pb 1.1 – Solution to Q. 2** For the data, you could have:

- For the BUILDING file, we could have:
  
  (Allgood Hall, 33.47520, -82.02503),
  
  (Institut Galilé, 48.959001, 2.339999)

- For the ROOM file, we could have:
  
  (Allgood Hall, 128, 1),
  
  (Institut Galilé, 205, 3),
  
  (Allgood Hall, 228, 2)

- For the PROF file, we could have:
  
  (Aubert, 839401, dae@ipn.net, 128),
  
  (Mazza, 938130, Dm@fai.net, 205)

**Pb 1.1 – Solution to Q. 3** If everything we knew about the campus came from that database, then

1. Yes, we could list all the professors.
2. No, we could not tell in which department is a professor.
3. No, we could not get the office hours of a professor.
4. Yes, we could list all the professors whose offices are in the same building.
5. Yes, we could list all the rooms.
6. If a new professor arrives, and has to share his office with another professor, we would not have to revise our database catalog (it is fine for two professor to have the same room number, in our model).
7. Yes, we could list which professors are at the same floor.
8. No, we could not tell which professor has the highest evaluations.
2 The Relational Model

Resources

- (Elmasri and Navathe 2010, Ch. 3), (Elmasri and Navathe 2015, Ch. 5), including the exercises (look at the exercises 15 and 16, for instance).
- The wikipedia page for Relational model\(^1\) and the category “Relational database management systems”\(^2\).

2.1 Concepts

The relational data model (or relational database schema) is:

- a mathematical model (use mathematical relations, set-theory, first-order predicate logic)
- with multiple implementations (“engineering approximation”)

2.1.1 Domains, Attributes, Tuples and Relations

- **Domain** (or type) = set of atomic (as far as the relation is concerned) values. You can compare it to datatype and literals, and indeed it can be given in the form of a data type, but it can be named and carry a logical definition (i.e., `List_of_major` as an enumerated data type, instead of just `String`).
- **Attribute** = Attribute name + attribute domain (but we’ll just write the name).
- **Relation Schema** (or scheme) = description of a relation, often written “RELATION_NAME(Attribute\(_1\), ..., Attribute\(_n\))”, where \(n\) is the degree (arity) of the relation, and the domain of Attribute\(_i\) is written \(\text{dom}(\text{Attribute}_i)\).

\(^1\)https://en.wikipedia.org/wiki/Relational_model
\(^2\)https://en.wikipedia.org/wiki/Category:Relational_database_management_systems
2.2 Constraints

2.2.1 Types of constraints

We now study constraints on the tuples. There are constraints on the scheme, for instance, “a relation can’t have two attributes with the same name”.

2.2.1.1 Inherent model-based constraints (implicit)

Those are part of the definition of the relational model.

- No duplicate tuple
- Arity must match

2.2.1.2 Schema-based constraints (explicit)

Those are parts of the schema.

- Value must match domain (“Domain constraint”), knowing that a domain can have constraints (NOT NULL, UNIQUE)
- Entity integrity constraint (no primary key value can be NULL)
- Referential integrity constraint (referred values must exist)

The last two will be studied in the next section.
2.2 Constraints

2.2.1.3 Application-based constraints (semantics)

Constraints that cannot be expressed in the schema, and hence must be enforced by some other way. Example: "the date of birth of an employee must be greater than xxx", "this year’s salary increase must be more than last year’s".

2.2.2 Keys

Tuples can’t be equal, so a subset of values must distinguish them, we study the corresponding subset of attributes.

- A **superkey** is the subset of attributes SK is a superkey for the relation R, if for all relation state r of R, all tuples t₁, t₂ in r are such that t₁[SK] ≠ t₂[SK].
- A **key** is a minimal superkey (i.e., removing any attribute from SK would break the uniqueness property).
- A **candidate key** is a key, a **primary key** is the selected candidate key (it is underlined¹).

Note: here we “retro-fit” those definitions, in DB design, they come first! We are making the assumption that the data pre-exist to the specification to make the concept clearer.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yellow</td>
<td></td>
<td>Rectangle</td>
<td>10</td>
<td>(5, 3)</td>
</tr>
<tr>
<td>Blue</td>
<td></td>
<td>Rectangle</td>
<td>10</td>
<td>(3, 9)</td>
</tr>
<tr>
<td>Blue</td>
<td></td>
<td>Circle</td>
<td>9</td>
<td>(4, 6)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>{A, B, C, D}</th>
<th>{A}</th>
<th>{B, C}</th>
<th>{D}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superkey ?</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Key ?</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

2.2.3 Foreign Keys

A **foreign key** (FK) is a set of attributes denoted FK in the relation schema R₁, such that FK is a foreign key of R₁ (“referencing relation”) that references R₂ (“referenced relation”) if

- FK refers to R₂ (i.e., the attributes in FK have the same domain(s) as the primary key PK of R₂)
- a value of FK in a tuple t₁ of r₁(R₁) either
  - occurs as a value of PK for some tuple t₂ of r₂(R₂), i.e., t₁[FK] = t₂[PK]
  - is NULL
  - in the first case, we say that "t₁ refers to t₂ ."

There is a referential integrity constraint from R₁ to R₂. We draw it with an arrow, note that it is possible that R₁ = R₂.

¹For technical reasons, underlined words cannot be searched in the document.
(Yes, we do need the state and the licence number to uniquely identify a driver’s licence, since many states use the same licence format\(^4\). Entity Integrity Constraint and Referential Integrity Constraint.

### 2.3 Transactions and Operations

Operations are of two kinds: retrievals and updates.

- Retrievals leave the relation state as it is and output a result relation. That is, retrieval: relation state → result relation
- Updates change the relation state. That is, update: relation state → relation state

They are two constraints for updates:

1. The new relation state must be “valid” (i.e., comply with the state constraints).
2. There might be transition constraints (your balance can’t become negative, for instance).

A transaction is a series of retrievals and updates performed by an application program, that leaves the DB in a consistent state.

#### 2.3.1 Insert / Delete / Update

(1.), (2.) and (3.) refers to the “remedies”, discussed in the next section.

\(^4\)https://ntsi.com/drivers-license-format/
2.3 Transactions and Operations

2.3.1.1 Insert

Insert <109920, Honda, Accord, 2012> into CAR.

How things can go wrong:

- NULL for the primary key (1.)
- Duplicate value for the primary key (1.)
- Wrong number of arguments (1.)
- Fail to reference an existing value for a foreign key (1.)

2.3.1.2 Delete

Delete the DRIVER tuple with State = GA and Licence_number = 123

How things can go wrong:

- Deleting tuples inadvertently (meta)
- Deleting tuples that are referenced (1., 2., 3.)

2.3.1.3 Update (a.k.a. modify)

Update Name of tuple in DRIVER where State = GA and Licence_number = 123 to Georges

How things can go wrong:

- NULL for the primary key (1.)
- Duplicate value for the primary key (1.)
- Change value that are referenced (1., 2., 3.)
- Change foreign key to a non-existing value (1.)

2.3.2 Dealing with violations

1. Reject (restrict)
2. Cascade (propagate)
3. Set default, or set NULL

Go back to the CAR example and populate it with some data to “see” how those options affects how the database react to various operations.

Exercises

Exercise 2.1 Connect the dots:

| Row • | Column header • | Table • | Attribute • | Tuple • |

Exercise 2.2 What do we call the number of attributes in a relation?

Exercise 2.3 At the logical level, does the order of the tuples in a relation matter?
Exercise 2.4 What is the difference between a database schema and a database state?

Exercise 2.5 What should we put in an attribute if its value is unknown?

Exercise 2.6 What, if any, is the difference between a superkey, a key, and a primary key?

Exercise 2.7 Name the two kind of integrities that must be respected by the tuples in a relation.

Exercise 2.8 What is entity integrity? Why is it useful?

Exercise 2.9 Are we violating an integrity constraint if we try to set the value of a primary key to NULL? If yes, which one?

Exercise 2.10 If in a relation $R_1$, an attribute $A_1$ is a foreign key referencing an attribute $A_2$ in a relation $R_2$, what does this implies about $A_2$?

Exercise 2.11 Give three examples of operations.

Exercise 2.12 What is the difference between an operation and a transaction?

Exercise 2.13 Give three different ways to deal with operations whose execution in isolation would result in the violation of one of the constraint.

Exercise 2.14 Consider the following two relations:

- COMPUTER(Owner, RAM, Year, Brand)
- OS(Name, Version, Architecture)

For each, give a) The arity of the relation, b) A (preferably plausible) example of tuple to insert.

Exercise 2.15 Define what is the domain constraint.

Exercise 2.16 Consider the following three relations:

- AUTHOR
  | Ref | Name | Address |
- BOOK
  | ISSN | AuthorRef | Title |
- GAINED-AWARD
  | Ref | Name | BookISSN | Year |

For each relation, answer the following:

1. What is, presumably, the primary key?
2. Are they, presumably, any foreign key?
3. Using the model you defined, could we determine which author won the greatest number of awards a particular year?
**Exercise 2.17** Consider the following three relations

![Diagram of relations](image)

1. What are the foreign keys in the ASSIGNED-TO relation? What are they referring to?
2. In the ASSIGNED-TO relation, explain why the Date attribute is part of the primary key. What would happen if it was not?
3. Assuming the database is empty, are the following instructions valid? If not, what integrity constraint are they violating?
   - a) Insert <'AM-356', 'Surfliner', 2012> into TRAIN
   - b) Insert <NULL, 'Graham Palmer', 'Senior'> into CONDUCTOR
   - c) Insert <'XB-124', 'GPalmer', '02/04/2018'> into ASSIGNED-TO
   - d) Insert <'BTed', 'Bobby Ted', 'Senior'> and <'BTed', 'Bobby Ted Jr.', 'Junior'> into CONDUCTOR

**Exercise 2.18** Consider the following relation schema and state:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Blue</td>
<td>Austin</td>
<td>true</td>
</tr>
<tr>
<td>1</td>
<td>Yellow</td>
<td>Paris</td>
<td>true</td>
</tr>
<tr>
<td>1</td>
<td>Purple</td>
<td>Pisa</td>
<td>false</td>
</tr>
<tr>
<td>2</td>
<td>Yellow</td>
<td>Augusta</td>
<td>true</td>
</tr>
</tbody>
</table>

Assuming that this is all the data we will ever have, discuss whenever \{A, B, C, D\}, \{A, B\} and \{B\} are superkeys and/or keys.

**Solution to Exercises**

**Solution 2.1** Row is Tuple, Column header is Attribute, Table is Relation.

**Solution 2.2** The degree, or arity, of the relation.

**Solution 2.3** No, it is a set.
Solution 2.4 The schema is the organization of the database (the meta-data), while the state is the content of the database (the data).

Solution 2.5 NULL

Solution 2.6 A superkey is a subset of attributes such that no two tuples have the same combination of values for all those attributes. A key is a minimal superkey, i.e., a superkey from which we cannot remove any attribute without losing the uniqueness constraint. The primary key is one of the candidate key, i.e., the key that was chosen.

Solution 2.7 Referential integrity and entity integrity.

Solution 2.8 Entity integrity ensures that each row of a table has a unique and non-null primary key value. It allows to make sure that every tuple is different from the others, and helps to “pick” elements in the database.

Solution 2.9 Yes, the entity integrity constraint.

Solution 2.10 Then we know that $A_2$ is the primary key of $R_2$.

Solution 2.11 Reading from the database, performing UPDATE or DELETE operations.

Solution 2.12  
  a) COMPUTER has for arity 4, and OS has for arity 3.

Solution 2.13 An operation is an “atomic action” that can be performed on the database (adding an element, updating a value, removing an element, etc.). A transaction is a series of such operations, and the assumption is that, even if it can be made of operations that, taken individually, could violate a constraint, an operation will leave the database in a consistent state.

Solution 2.14 An operation whose execution in isolation would result in the violation of a constraint can either a) be “restricted” (i.e., not executed), b) result in a propagation (i.e., the tuples that would violate a constraint are updated or deleted accordingly), or c) result in some values in tuples that would violate a constraint to be set to a default value, or the NULL value (this last option works only if the constraint violated is the referential entity constraint).

Solution 2.15 The requirement that each tuple must have for an attribute $A$ an atomic value from the domain dom($A$), or NULL.

Solution 2.16 To answer 1 and 2, the diagram would become:
For the last question, the answer is yes: based on the ISSN of the book, we can retrieve the author of the book. Hence, knowing which book was awarded which year, by looking in the GAINED-AWARD table, gives us the answer to that question.

**Solution 2.17**

1. In ASSIGNED-TO, TrainRef is a FK to TRAIN.Ref, and ConductorID is a FK to CONDUCTOR.CompanyID.

2. In this model, a conductor can be assigned to different trains on different days. If Date was not part of the PK of ASSIGNED-TO, then a conductor could be assigned to only one train.

3. a) Yes, this instruction is valid.
   b) No, it violates the entity integrity constraint: NULL can be given as a value to an attribute that is part of the PK.
   c) No, it violates the referential integrity constraint: 'XB-124 and 'GPalmer' are not values in TRAIN.Ref and CONDUCTOR.CompanyID.
   d) No, it violates the key constraint: two tuples can’t have the same value for the primary key.

**Solution 2.18**

- \{A, B, C, D\} is a superkey (the set of all the attributes is always a superkey), but not a superkey, as removing e.g. D would still make it a superkey.
- \{A, B\} is a superkey and a key, as neither \{A\} nor \{B\} are keys.
- \{A\} is not a key, and not a superkey: multiple tuples have the value 1.

**Problems**

**Problem 2.1 (Finding candidate key in a CLASS relation)** Consider the relation

CLASS(Course_Number, Univ_Section_Number, Instructor_Name, Semester, Building_Code, Room_Number, Time, Weekdays, Credit_Hours)

This relation represents classes taught in a university. The goal is to be able to have multiple offerings (classes) of courses over several semesters. List three possible candidate keys and
describe under what conditions each candidate key would be valid. Each candidate key should have between one and three attributes.

Here are some examples of values for the attributes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Possible Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Course_Number</td>
<td>CSCI3410, CSCI1302</td>
</tr>
<tr>
<td>Building_Code</td>
<td>AH, UH, ECC</td>
</tr>
<tr>
<td>Univ_Section_Number</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>Room_Number</td>
<td>E127, N118</td>
</tr>
<tr>
<td>Instructor_Name</td>
<td>John Smith, Sophie Adams</td>
</tr>
<tr>
<td>Time</td>
<td>1400, 1230, 0900</td>
</tr>
<tr>
<td>Semester</td>
<td>Spring 2015, Fall 2010, Summer 2012</td>
</tr>
<tr>
<td>Weekdays</td>
<td>M, MW, MWF, T, TH</td>
</tr>
<tr>
<td>Credit_Hours</td>
<td>1, 2, 3, 4</td>
</tr>
</tbody>
</table>

**Problem 2.2 (Relational model for a cinema company)** A cinema company wants you to design a relational model for the following set-up:

- The company has movie stars. Each star has a name, birth date, and unique id.
- The company has information about movies: title, year, length, and genre. Each movie has a unique id, and features multiple stars.
- The company owns movie theaters as well. Each theater has a name, address, and a unique id.
- Furthermore, each theater has a set of auditoriums. Each auditorium has a unique number, and seating capacity.
- Each theater can schedule movies at show-times. Each show-time has a unique id, a start time, and is for a specific movie, at a theater auditorium.
- The company sells tickets for scheduled show-times. Each ticket has a unique ticket id, and a price.

**Problem 2.3 (Relational model for bills)** Propose a relational model for the following situation:

- You want to store the bills that are debated and voted on at the U.S. House of Representatives. Each bill has a name, a (unique) sponsor which needs to be a member of the house, a date when it was discussed (which may not have been fixed yet).
- You want to record the name, political group, beginning and (expected) end of the term of the members of the house.
- You also want to record the name of the Speaker, Majority Leader, Minority Leader, Majority Whip, and Minority Whip, which are all members of the house.
- Finally, for each bill, you want to record the vote of every member of the house.
Solution to Selected Problems

Solution to Pb 2.2

Solution to Pb 2.3 Be careful: saying that a bill has a unique sponsor does not imply that a
the sponsor is a good primary key for the bills: a house member could very well be the
sponsor of multiple bills! It just implies that a single attribute is enough to hold the name
of the sponsor.

For simplicity, we added an Id to our MEMBER and BILL relations. Note that having a “role”
in the MEMBER relation to store the information about speaker, etc., would be extremely
inefficient, since we would add an attribute to the ~435 members that would be NULL in
~430 of them.
3 The SQL Programming Language

Resources

- (Elmasri and Navathe 2010, Ch. 4–5), (Elmasri and Navathe 2015, Ch. 6–7) describes SQL, not one of its implementation.

This chapter will be “code-driven”: the code will illustrate and help you understand some concepts. You may want to have a look at the “Setting Up Your Work Environment” Section, as early as possible in this lecture. On top of being a step-by-step guide to install and configure a relational database management system, it contains a list of useful links.

3.1 Actors

3.1.1 Technologies

- There are other models: document-based, graph, column-based, and key-value models. “NoSQL” data-model, more flexible, but only defined by opposition.
- Most commons DBMS are relational database management system (RDBMS):
  - Oracle Database
  - MySQL\(^2\) and its fork, MariaDB\(^3\)
  - Microsoft SQL Server\(^4\)
  - PostgreSQL\(^5\)
  - IBM DB2\(^6\)
  - Microsoft Access\(^7\)
  - SQLite\(^8\)

Most of them supports semi-structured data, i.e., models that are not strictly speaking relational, some are "multi-model DBMS".
- Structured Query Language is the language for RDBMS, it is made of 4 sublanguages:

\(^2\)https://www.mysql.com/
\(^3\)https://mariadb.org/
\(^5\)https://www.postgresql.org/
\(^7\)https://products.office.com/en-us/access
\(^8\)https://www.sqlite.org/index.html
3.1 Actors

- Data Query Language,
- Data Definition Language (schema creation and modification),
- Data Control Language (authorizations, users),
- Data Manipulation Language (insert, update and delete).

The three last sublanguages being dubbed "Data Manipulation Language".

3.1.2 SQL

3.1.2.1 Yet another vocabulary

<table>
<thead>
<tr>
<th>&quot;Common&quot; / Relational</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Database&quot;</td>
<td>Schema</td>
</tr>
<tr>
<td>&quot;Set of databases&quot;</td>
<td>Catalog (collection of named schema)</td>
</tr>
<tr>
<td>Relation</td>
<td>Table</td>
</tr>
<tr>
<td>Tuple</td>
<td>Row</td>
</tr>
<tr>
<td>Attribute</td>
<td>Column, or Field</td>
</tr>
</tbody>
</table>

3.1.2.2 Schema elements

A schema is made of

- Tables
- Type
- Domain
- View (result set of a stored query on the data)
- Assertion (constraints, transition constraints)
- Triggers (action to take after certain operations are performed)

Type and domains are two different things in some implementations, cf. for instance PostgreSQL, where a domain is defined to be essentially a datatype with constraint.\(^9\)

3.1.2.3 Syntax

- A programming language: strict, cryptic error messages, tricky, evolves
- SQL is “kind of” case-insensitive\(^10\), doesn’t care about spaces and new lines
- Comments are with -- or */ ...*/
- Every statement ends with a ;
- Syntax is in Pb 3.1.
- We will stick to what’s in MariaDB and MySQL here (no domain, limited data type definition)

---


\(^10\) The SQL keywords are case-insensitive, but the table and schema names are sometimes case-sensitive, it depends of the actual implementation. For instance, MySQL is completely case-insensitive (reserved words, tables, attributes), MariaDB isn’t (case for table names matter).
3.2 First Commands

```sql
/* code/sql/HW_FACULTY.sql */

CREATE SCHEMA HW_FACULTY;

CREATE TABLE HW_FACULTY.PROF
    (Fname VARCHAR(15), -- No String!
     Room INT, -- shorthad for INTEGER, are also available: SMALLINT, FLOAT, REAL, DEC
     Title CHAR(3), -- fixed-length string, padded with blanks if needed
     Tenured BIT(1),
     Nice BOOLEAN, -- True / False (= 0) / Unknown
     Hiring DATE,
     Last_seen TIME,
     FavoriteFruit ENUM('apple', 'orange', 'pear'),
     PRIMARY KEY(Fname, Hiring)
    );

/* Or, instead of using the fully qualified name HW_FACULTY.PROF, we could have:

USE HW_FACULTY;
CREATE TABLE PROF(...)
*/

USE HW_FACULTY;

INSERT INTO PROF VALUES
    ("Clément" -- Or 'Clément', but '"' and '"' are neat!
     , 290
     , 'PhD'
     , 0
     , NULL
     , '19940101' -- Or '940101', '1994-01-01', '94/01/01'
     , '090500' -- Or '09:05:00', '9:05:0', '9:5:0', '090500'
     -- Note also the existence of DATETIME, with 'YYYY-MM-DD HH:MM:SS'
     , 'apple'
    );
```

The following commands are particularly useful:

SHOW SCHEMAS; -- List the schemas.
SHOW TABLES; -- List the tables in a schema.
SHOW CREATE TABLE <TableName> -- Gives the command to "re-construct" TableName.
DESCRIBE <TableName>; -- Show the structure of TableName.
SELECT * FROM <TableName> -- List all the rows in TableName.
3.3 Overview of Constraints

1. Primary Key
2. Foreign Key
3. NOT NULL
4. UNIQUE
5. DEFAULT
6. CHECK

We know 1. and 2. from the Relational Model, here comes new constraints that can’t be describe in our relations.

```sql
CREATE TABLE HURRICANE(
    Name VARCHAR(25) PRIMARY KEY,
    WindSpeed INT DEFAULT 76,
    Above VARCHAR(25)
);
-- WindSpeed INT CHECK (WindSpeed > 74 AND WindSpeed < 500),
-- would be parsed but wouldn't have any effect!
CREATE TABLE STATE(
    Name VARCHAR(25) UNIQUE, -- NULL can be inserted
    Postal_abbr CHAR(2) NOT NULL
);
```

MySQL can output a description of those tables for us:

```sql
MariaDB [HW_CONSTRAINTS_PART1]> DESCRIBE HURRICANE;
+-------------+-------------+------+-----+---------+-------+
| Field       | Type        | Null | Key | Default | Extra |
|-------------+-------------+------+-----+---------+-------+
| Name        | varchar(25) | NO   | PRI | NULL    |       |
| WindSpeed   | int(11)     | YES  |     | 76      |       |
| Above       | varchar(25) | YES  |     | NULL    |       |
+-------------+-------------+------+-----+---------+-------+
3 rows in set (0.01 sec)

MariaDB [HW_CONSTRAINTS_PART1]> DESCRIBE STATE;
+-------------+-------------+------+-----+---------+-------+
| Field       | Type        | Null | Key | Default | Extra |
|-------------+-------------+------+-----+---------+-------+
| Name        | varchar(25) | NO   | PRI | NULL    |       |
| Postal_abbr | char(2)     | NO   | UNI | NULL    |       |
+-------------+-------------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```
3.3 Overview of Constraints

-- Primary Keys

-- Adding a primary key:
ALTER TABLE STATE ADD PRIMARY KEY (Name);

-- Removing the primary key:
ALTER TABLE STATE DROP PRIMARY KEY;

-- UNIQUE Constraint

-- Adding a UNIQUE constraint
ALTER TABLE STATE ADD UNIQUE (Postal_abbr);

-- Removing a UNIQUE constraint
ALTER TABLE STATE DROP INDEX Postal_abbr;

-- NOT NULL Constraint

-- Adding the NOT NULL constraint
ALTER TABLE STATE MODIFY Postal_abbr CHAR(2) NOT NULL;

-- Removing the NOT NULL constraint
ALTER TABLE STATE MODIFY Postal_abbr CHAR(2);

-- Default value

-- Changing the default value
ALTER TABLE HURRICANE ALTER COLUMN WindSpeed SET DEFAULT 74;

-- Removing the default value
ALTER TABLE HURRICANE ALTER COLUMN WindSpeed DROP DEFAULT;

-- Foreign key

-- Adding a foreign key constraint
ALTER TABLE HURRICANE ADD FOREIGN KEY (Above) REFERENCES STATE(Name);

-- Removing a foreign key constraint is out of the scope of this lecture.
-- If you are curious, you can have a look at
→ https://www.w3schools.com/sql/sql_foreignkey.asp
-- Dropping a foreign key constraint requires your constraint to have a name, something we did not introduce.

- **NOT NULL** is to some extend part of the datatype.
- Note the difference between adding and removing the UNIQUE constraint: in one case, the parenthesis are mandatory, in the other, they would cause an error! In this example, the parenthesis around (Name) and (Postal_abbr) are mandatory.
- The datatype of the foreign key has to be the same as what we are referring.

A bit of testing:

```sql
INSERT INTO STATE VALUES ('Georgia', 'GA');
INSERT INTO STATE VALUES ('Texas', 'TX');
INSERT INTO STATE VALUES ('FLORIDA', 'FL');
UPDATE STATE SET Name = 'Florida'
  WHERE Postal_abbr = 'FL';

-- There's an error with the following request. Why?
INSERT INTO HURRICANE VALUES ('Irma', 150, 'FL');

/*
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails
  'HW_CONSTRAINTS_PART1`.`HURRICANE`, CONSTRAINT
  `HURRICANE_ibfk_1` FOREIGN KEY (`Above`) REFERENCES `STATE` (`Name`)*/

INSERT INTO HURRICANE VALUES ('Harvey', DEFAULT , 'Texas');
INSERT INTO HURRICANE VALUES ('Irma', 150 , 'Florida');
DELETE FROM HURRICANE
  WHERE Name = 'Irma';
INSERT INTO HURRICANE VALUES ('Irma', 150 , 'Georgia');

UPDATE HURRICANE SET Above = 'Georgia'
  WHERE Name = 'Irma';

/*
MariaDB [HW_CONSTRAINTS_PART1]> SELECT * FROM HURRICANE;
+--------+-----------+---------+
| Name   | WindSpeed | Above   |
|--------+-----------+---------|
| Harvey | 74        | Texas   |
| Irma   | 150       | Georgia |
+--------+-----------+---------+
*/

-- There's an error with the following request. Why?
UPDATE HURRICANE SET Above = 'North Carolina'
  WHERE Name = 'Irma';

-- Let's patch it, by adding North Carolina to our STATE table.
INSERT INTO STATE VALUES ('North Carolina', 'NC');
UPDATE HURRICANE SET Above = 'North Carolina'
```
3.4 Foreign Keys

3.4.1 A first example

```sql
CREATE TABLE STORM(
    Name VARCHAR(25) PRIMARY KEY,
    Kind ENUM('Tropical Storm', 'Hurricane'),
    WindSpeed INT,
    Creation DATE
); 

-- I can change my enumerated datatype:
ALTER TABLE STORM MODIFY Kind ENUM('Tropical Storm', 'Hurricane', 'Typhoon');

CREATE TABLE STATE(
    Name VARCHAR(25) UNIQUE,
    Postal_abbr CHAR(2) PRIMARY KEY,
    Affected_by VARCHAR(25),
    FOREIGN KEY (Affected_by) REFERENCES STORM(Name)
        ON DELETE SET NULL
        ON UPDATE CASCADE
); 

INSERT INTO STORM VALUES('Harvey', 'Hurricane', 130, '2017-08-17');
-- In the following, the entry gets created, but date is 0000-00-00!
INSERT INTO STORM VALUES('Dummy', 'Hurricane', 120, '2017-17-08');
-- In the following, there's an error, and nothing gets inserted.
INSERT INTO STORM VALUES('Dummy2', 'Hurricane', 120, DATE'2017-17-08');
-- The next one sets NULL for DATE.
INSERT INTO STORM VALUES('Irma', 'Tropical Storm', 102, DEFAULT);
```

MySQL will always notify you if there is an error in a date attribute

```sql
INSERT INTO STATE VALUES('Georgia', 'GA', NULL);
INSERT INTO STATE VALUES('Texas', 'TX', NULL);
INSERT INTO STATE VALUES('Florida', 'FL', NULL);

-- This instruction is not using the primary key, is that a problem?
UPDATE STATE SET Affected_by = 'Harvey'
    WHERE Name = 'Georgia';

UPDATE STORM SET Name = 'Harley' WHERE Name = 'Harvey';
DELETE FROM STORM
    WHERE Name = 'Harley';
```
3.4.2 Foreign keys restrictions

```sql
CREATE TABLE F_Key(
    Attribute VARCHAR(25) PRIMARY KEY
);

CREATE TABLE Table_default(
    Attribute1 VARCHAR(25) PRIMARY KEY,
    Attribute2 VARCHAR(25),
    FOREIGN KEY (Attribute2) REFERENCES F_Key(Attribute)
);

-- By default, this foreign key will restrict.

CREATE TABLE Table_restrict(
    Attribute1 VARCHAR(25) PRIMARY KEY,
    Attribute2 VARCHAR(25),
    FOREIGN KEY (Attribute2) REFERENCES F_Key(Attribute)
        ON DELETE RESTRICT
        ON UPDATE RESTRICT
);

CREATE TABLE Table_cascade(
    Attribute1 VARCHAR(25) PRIMARY KEY,
    Attribute2 VARCHAR(25),
    FOREIGN KEY (Attribute2) REFERENCES F_Key(Attribute)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);

CREATE TABLE Table_set_null(
    Attribute1 VARCHAR(25) PRIMARY KEY,
    Attribute2 VARCHAR(25),
    FOREIGN KEY (Attribute2) REFERENCES F_Key(Attribute)
        ON DELETE SET NULL
        ON UPDATE SET NULL
);

/*
 * You might encounter a
 * ON UPDATE SET DEFAULT
 * but this reference option (cf.
 * worked only with a particular engine ( 
 * and won't be treated here.
 */

INSERT INTO F_Key VALUES('First Test');
```
### 3.4 Foreign Keys

```sql
INSERT INTO Table_default VALUES('Default', 'First Test');
INSERT INTO Table_restrict VALUES('Restrict', 'First Test');
INSERT INTO Table_cascade VALUES('Cascade', 'First Test');
INSERT INTO Table_set_null VALUES('Set null', 'First Test');

SELECT * FROM Table_default;
SELECT * FROM Table_restrict;
SELECT * FROM Table_cascade;
SELECT * FROM Table_set_null;

-- The following will fail because of the Table_default table:
UPDATE F_Key SET Attribute = 'After Update'
WHERE Attribute = 'First Test';
DELETE FROM F_Key
WHERE Attribute = 'First Test';

-- Let us drop this table, and try again.
DROP TABLE Table_default;

-- The following fails too, this time because of the Table_restrict table:
UPDATE F_Key SET Attribute = 'After Update'
WHERE Attribute = 'First Test';
DELETE FROM F_Key
WHERE Attribute = 'First Test';

-- Let us drop this table, and try again.
DROP TABLE Table_restrict;

-- Let's try again:
UPDATE F_Key SET Attribute = 'After Update' WHERE Attribute = 'First Test';

-- And let's print the situation after this update:
SELECT * FROM Table_cascade;
SELECT * FROM Table_set_null;

/*
MariaDB [HW_CONSTRAINTS_PART3]> SELECT * FROM Table_cascade;
+------------+--------------+
| Attribute1 | Attribute2   |
+------------+--------------+
| Cascade    | After Update |
+------------+--------------+
1 row in set (0.00 sec)

MariaDB [HW_CONSTRAINTS_PART3]> SELECT * FROM Table_set_null;
+------------+------------+
| Attribute1 | Attribute2 |
+------------+------------+
*/
3.4 Foreign Keys

-- Let's make a second test.
INSERT INTO F_Key VALUES('Second Test');
INSERT INTO Table_cascade VALUES('Default', 'Second Test');
INSERT INTO Table_set_null VALUES('Restrict', 'Second Test');
DELETE FROM F_Key
    WHERE Attribute = 'Second Test';

/*
MariaDB [HW_CONSTRAINTS_PART3]> SELECT * FROM Table_cascade;
+------------+--------------+
| Attribute1 | Attribute2 |
+------------+--------------+
| Cascade    | After Update |
+------------+--------------+
1 row in set (0.00 sec)

MariaDB [HW_CONSTRAINTS_PART3]> SELECT * FROM Table_set_null;
+------------+------------+
| Attribute1 | Attribute2 |
+------------+------------+
| Restrict   | NULL       |
| Set null   | NULL       |
+------------+------------+
2 rows in set (0.00 sec)
*/

3.4.3 Constructing and populating a new example

3.4.3.1 Construction

- Remember, we start by creating a schema and tables inside of it.
- What if foreign keys are mutually dependent? What if we have something like:

```
CREATE TABLE PROF(
    Login VARCHAR(25) PRIMARY KEY,
    Name, Department
);
CREATE TABLE DEPARTMENT(
    Code, Name, Head
);
```

![Diagram of PROF and DEPARTMENT tables with foreign keys](image)
3.4 Foreign Keys

```
    CREATE TABLE DEPARTMENT(
        Code CHAR(5) PRIMARY KEY,
        Name VARCHAR(25),
        Head VARCHAR(25),
        FOREIGN KEY (Head) REFERENCES PROF(Login)
            ON UPDATE CASCADE
    );

    ALTER TABLE PROF ADD FOREIGN KEY (Department)
        REFERENCES DEPARTMENT(Code);
```

Note the structure of the `ALTER TABLE` command:

- `... KEY Department REFERENCES Code;` => error
- `... KEY (Department) REFERENCES (Code);` => error
- `... KEY PROF(Department) REFERENCES DEPARTMENT(Code);` => ok

```
    CREATE TABLE STUDENT(
        Login VARCHAR(25) PRIMARY KEY,
        Name VARCHAR(25),
        Registered DATE,
        Major CHAR(5),
        FOREIGN KEY (Major) REFERENCES DEPARTMENT(Code)
    );

    CREATE TABLE GRADE(
        Login VARCHAR(25),
        Grade INT,
        PRIMARY KEY(Login, Grade),
        FOREIGN KEY (Login) REFERENCES STUDENT(Login)
    );
```

3.4.3.2 Populating

We can insert multiple values at once:

```
    INSERT INTO DEPARTMENT VALUES
        ('MATH', 'Mathematics', NULL),
        ('CS', 'Computer Science', NULL);
```

We can specify which attributes we are giving:

```
    INSERT INTO DEPARTMENT (Code, Name) VALUES
        ('CYBR', 'Cyber Security');
```

And we can even specify the order (even the trivial one):

```
    INSERT INTO PROF (Login, Department, Name) VALUES
        ('caubert', 'CS', 'Clément Aubert');
```
3.5 A First Look at Conditions

Order of clauses does not matter, not even for optimization purpose.

UPDATE <table>
SET <attribute1> = <value1>, <attribute2> = <value2>, ...
WHERE <condition>;

SELECT <attribute list, called projection attributes>
FROM <table list>
WHERE <condition>;

DELETE FROM <table list>
WHERE <condition>;

Conditions can
- be compounded
  - condition1 AND condition2
  - condition1 OR condition2
  - NOT condition
  - Usage of parenthesis is possible
- be trivial / empty
- use regular expressions
  - uses the expression LIKE,
  - escape character is \\,
  - _ will match one character (any character), % will match any number of character,
  - advanced regular expression possible using the REGEXP keyword.

SELECT Login FROM STUDENT;
3.6 Three-Valued Logic

Cf. (Elmasri and Navathe 2010, 5.1.1), (Elmasri and Navathe 2015, 7.1.1)

3.6.1 Meaning of NULL

NULL is

1. Unknown value ("Nobody knows")
   - What is the date of birth of Jack the Ripper\(^{11}\)?
   - Does P equal NP?\(^{12}\)
2. Unavailable / Withheld ("I don’t have that information with me at the moment")
   - What is the number of english spies in France?
   - What is the VIN of your car?
   - What is the identity of the Tiananmen Square person?
3. Not Applicable ("Your question doesn’t make sense")
   - What is the US SSN of a french person?
   - What is the email address of an author of the XIXth century?

\(^{11}\)https://en.wikipedia.org/wiki/Jack_the_Ripper
3.6.2 Comparison with unknown values \{truth\_tables\}

If NULL is involved in a comparison, the result evaluates to “Unknown”.

\[
\begin{array}{c|ccc}
\text{AND} & T & F & U \\
T & T & F & U \\
F & F & F & F \\
U & U & F & U \\
\end{array}
\]

\[
\begin{array}{c|ccc}
\text{OR} & T & F & U \\
T & T & T & T \\
F & T & F & U \\
U & T & U & U \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{NOT} & \text{\textbackslash} \\
T & F \\
F & T \\
U & U \\
\end{array}
\]

You can test if a value is NULL with IS NULL.

1. `INSERT INTO DEPARTMENT VALUES ('Hist', 'History', NULL);`
2. `SELECT * FROM DEPARTMENT WHERE Head IS NULL;`
3. `SELECT * FROM DEPARTMENT WHERE Head IS NOT NULL;`
4. `SELECT COUNT(*) FROM GRADE WHERE Grade IS NULL;`

3.7 Various Tools

For DISTINCT, ALL and UNION, cf. (Elmasri and Navathe 2010, 4.3.4) or (Elmasri and Navathe 2015, 6.3.4). For ORDER BY, cf. (Elmasri and Navathe 2010, 4.3.6) or (Elmasri and Navathe 2015, 6.3.6). For aggregate functions, cf. (Elmasri and Navathe 2010, 5.1.7) or (Elmasri and Navathe 2015, 7.1.7).

3.7.1 Transactions

We can save the current state, and start a series of transactions, with the command

\`START TRANSACTION;\`

All the commands that follows are “virtually” executed: you can undo them all using

\`ROLLBACK;\`

This puts you back in the state you were in before starting the transaction. If you want all the commands you typed in-between to be actually enforced, you can use the command

\`COMMIT;\`
Nested transactions are technically possible, but they are counter-intuitive and should be avoided, cf. https://www.sqlskills.com/blogs/paul/a-sql-server-dba-myth-a-day-2630-nested-transactions-are-real/.

### 3.7.2 DISTINCT / ALL

The result of a SELECT query, for instance, is a table, and SQL treats tables as multi-set, hence there can be repetitions in the result of a query, but we can remove them:

```sql
SELECT DISTINCT Major FROM STUDENT;
```

The default behaviour is equivalent to specifying ALL, and it display the duplicates.

### 3.7.3 UNION

```sql
(SELECT Login FROM STUDENT) UNION (SELECT Login FROM PROF);
```

There is also INTERSECT and EXCEPT in the specification, but MySQL does not implement them (cf. https://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems#Database_capabilities).

### 3.7.4 ORDER BY

You can have ORDER BY specifications:

```sql
SELECT Login FROM GRADE
WHERE Grade > 3.0
ORDER BY Grade;
```

```sql
SELECT Login FROM GRADE
WHERE Grade > 3.0
ORDER BY Grade DESC;
```

```sql
SELECT Login, Major FROM STUDENT
ORDER BY Major, Name;
```

ORDER BY order by ascending order by default.

### 3.7.5 Aggregate functions

You can use MAX, SUM, MIN, AVG, COUNT:

```sql
SELECT MAX(Registered) FROM STUDENT;
```

```sql
SELECT COUNT(Name) FROM STUDENT;
```

```sql
SELECT COUNT(DISTINCT Name) FROM STUDENT;
```

### 3.7.6 Aliases for columns

```sql
SELECT Login FROM PROF;
```
<table>
<thead>
<tr>
<th>Login</th>
</tr>
</thead>
<tbody>
<tr>
<td>aturing</td>
</tr>
<tr>
<td>caubert</td>
</tr>
<tr>
<td>bgates</td>
</tr>
<tr>
<td>perdos</td>
</tr>
<tr>
<td>---------</td>
</tr>
</tbody>
</table>

```
SELECT Login AS Username FROM PROF;
```

```
+---------+
| Username |
+---------+
| aturing  |
| caubert  |
| bgates   |
| perdos   |
+---------+

The purpose of aliases will be clearer as we study select-project-join queries.

### 3.8 More Select Queries

For select-project-join, cf. (Elmasri and Navathe 2010, 4.3.1) or (Elmasri and Navathe 2015, 6.3.1). For aliases, cf. (Elmasri and Navathe 2010, 4.3.2) or (Elmasri and Navathe 2015, 6.3.2). For nested queries, cf. (Elmasri and Navathe 2010, 5.1.2) or (Elmasri and Navathe 2015, 7.1.2).

#### 3.8.1 Select-project-join

```
SELECT Login FROM PROF, DEPARTMENT
WHERE DEPARTMENT.Name = 'Mathematics'
AND Department = Code;
```

- `Department.Name = 'Mathematics'` is the selection condition
- `Department = Code` is the join condition, because it combines two tuples.
- Why do we use the fully qualified name attribute for `Name`?
- We have to list all the tables we want to consult, even if we use fully qualified names.

```
SELECT Name FROM STUDENT, GRADE
WHERE Grade > 3.0
AND STUDENT.Login = GRADE>Login;
```

- `Grade > 3.0` is the selection condition
- `STUDENT/Login = GRADE/Login` is the join condition

We can have two join conditions!

```
SELECT PROF.Name FROM PROF, DEPARTMENT, STUDENT
WHERE STUDENT.Name = 'Ava Alyx'
AND STUDENT.Major = DEPARTMENT.Code
AND DEPARTMENT.Head = PROF.Login;
```
3.8.2 Aliasing tuples

We can use aliases to shorten the previous query:

```sql
SELECT PROF.Name FROM PROF, DEPARTMENT, STUDENT AS B
WHERE B.Name = 'Ava Alyx'
AND B.Major = DEPARTMENT.Code
AND DEPARTMENT.Head = PROF.Login;
```

We can use multiple aliases:

```sql
SELECT A.Name FROM PROF AS A, DEPARTMENT, STUDENT AS B
WHERE B.Name = 'Ava Alyx'
AND B.Major = DEPARTMENT.Code
AND DEPARTMENT.Head = A.Login;
```

For those two, aliases were convenient, but not required to write the query. In some cases, we can’t do without aliases, for instance if we want to compare two rows in the same table:

```sql
SELECT OthersLogin FROM GRADE AS Mine, GRADE AS Others
WHERE Mine.Login = 'aalyx'
AND Mine.Grade < Others.Grade;
```

```sql
SELECT Fellow.Name AS 'Fellow of Ava'
FROM STUDENT AS Me, STUDENT AS Fellow
WHERE Me.Name = 'Ava Alyx'
AND Fellow.Major = Me.Major
AND NOT Fellow.Name = 'Ava Alyx';
```

AND NOT Me = Fellow would not work. Note that AS 'Fellow of Ava' is another kind of aliasing, mentioned in a previous section.

3.8.3 Nested queries

```sql
SELECT Login FROM GRADE
WHERE Grade > (SELECT AVG(Grade) FROM GRADE);
```

A nested query is made of an outer query (SELECT Login...) and an inner query (SELECT AVG(Grade)...).

(Average of all non-NULL values.)

```sql
SELECT Login FROM GRADE
WHERE Grade >= ALL (SELECT Grade FROM GRADE WHERE Grade IS NOT NULL);
```

```sql
SELECT Login FROM PROF
WHERE DEPARTMENT IN (SELECT Major FROM STUDENT
WHERE Login LIKE '%a');
```

Why can’t we use =?
Actually, nested query that uses = can often be rewritten without being nested:

```
1 SELECT  Login
2 FROM PROF
3 WHERE DEPARTMENT = (   SELECT Major
4          FROM STUDENT
5         WHERE Login = "cjoella";
```

becomes

```
1 SELECT PROF.Login
2 FROM PROF, STUDENT
3 WHERE DEPARTMENT = Major AND STUDENT.Login = "cjoella";
```

Conversely, you can sometimes write select-project-join as nested queries For instance,

```
1 SELECT Name FROM STUDENT, GRADE
2    WHERE Grade > 3.0
3    AND STUDENT.Login = GRADE.Login;
```

becomes

```
1 SELECT Name FROM STUDENT
2    WHERE Login IN (SELECT Login FROM GRADE WHERE Grade > 3.0);
```

### 3.9 Setting Up Your Work Environment

This part is a short tutorial to install and configure a working relational DBMS.

#### 3.9.1 Installation

You will install the MySQL¹³ DataBase Management System, or its community-developed fork, MariaDB¹⁴. Below are the instruction to install MySQL Community Edition on Windows 10, and MariaDB on Linux-based distribution, but both are developed for every major operating system (Mac OS, Windows, Debian, Ubuntu, etc.): feel free to pick one or the other, it won’t really make a difference in this course (up to some minor aspects). MySQL is more common, MariaDB is growing, both are released under GNU General Public License¹⁵, well-documented and free of charge for their “community” versions.

It is perfectly acceptable to install MySQL or MariaDB on a virtual machine for this class. Remember that using your on the hub¹⁶ account, you should have access to VMware¹⁷ and Windows licences, and that using Virtual Box¹⁸ should also work fine. Once this step is completed, or if MySQL or MariaDB is already installed on your computer, go to the next step.

I tried to give precise and up-to-date instructions below, follow them carefully, read the messages displayed on your screen, make sure a step was correctly executed before moving to the next one, and everything should be all right. Also, remember:

¹³https://www.mysql.com/
¹⁴https://mariadb.org/
¹⁵https://www.gnu.org/licenses/licenses.html#GPL
¹⁶https://onthehub.com/
¹⁷https://www.vmware.com/
¹⁸https://www.virtualbox.org/wiki/Downloads
1. Don’t wait, set your system early.
2. I will be happy to assist you, but there’s nothing I can do with an email like “It doesn’t work”: if you look for help, be detailed and clear about what you think went wrong.

The following links could be useful:

- https://dev.mysql.com/doc/refman/8.0/en/mysql-installer-workflow.html, and particularly the page on windows installation and the page on Linux installation using package-managers,
- https://www.linode.com/docs/databases/mysql/how-to-install-mysql-on-debian-8

### 3.9.1.1 Installing MySQL on Windows 10


where XXX is a number version (e.g., 8.0.13.0.), and YYY is the size of the file (e.g., 16.3M). On the next page, click on the (somewhat hidden) "No thanks, just start my download." button.

2. Save the “mysql-installer-web-community-XXX.msi” file, and open it. If there is an updated version of the installer available, agree to download it. Accept the license term.

3. We will now install the various components needed for this class, leaving all the choices by defaults. This means that you need to do the following:
   a) Leave the first option on "Developer Default" and click on "Next"
   b) Click on “Next” even if you don’t meet all the requirements
   c) Click on “Execute”. The system will download and install several softwares (this may take some time).
   d) Click on “Next” twice, leave “Type and Networking” on "Standalone MySQL Server / Classic MySQL Replication" and click "Next", and leave the next options as they are (unless you know what you do and want to change the port, for instance) and click on "Next".
   e) You now need to choose a password for the MySQL root account. It can be anything, just make sure to memorize it. Click on “Next”.
   f) On the "Windows Service" page, leave everything as it is and click on “Next”.
   g) On the "Plugins and Extensions" page, leave everything as it is and click on "Next".
   h) Finally, click “Execute” on the “Apply Configuration” page, and then on “Finish”.
   i) Click on “Cancel” on the “Product Configuration” page and confirm that you don’t want to add products: we only need to have MySQL Server XXX configured.

4. We now want to make sure that MySQL is running: launch Windows’ “Control Panel”, then click on “Administrative Tools”, and on “Services”. Look for “MySQLXX”, its status should be “Running”. If it is not, right-click on it and click on “Start”.

---

5. Open a command prompt (search for `cmd`, or use PowerShell\(^{21}\)) and type

```
cd "C:\Program Files\MySQL\MySQL Server 8.0\bin"
```

If this command fails, it is probably because the version number changed: open the file explorer, go to `C:\Program Files\MySQL\`, look for the right version number, and update the command accordingly.

Then, enter

```
mysql -u root -p
```

and enter the password you picked previously for the root account. You are now logged as root in your database management system, you should see a brief message, followed by a prompt

```
mysql >
```

### 3.9.1.2 Installing MySQL on macOS

The instructions are almost the same as for Windows. Read [https://dev.mysql.com/doc/refman/5.6/en/osx-installation-pkg.html](https://dev.mysql.com/doc/refman/5.6/en/osx-installation-pkg.html) and download the file from [https://dev.mysql.com/downloads/mysql/](https://dev.mysql.com/downloads/mysql/) once you selected "macOS" as your operating system. Install it, leaving everything by default but adding a password (refer to the instructions for windows). Then, open a command-line interface (the terminal), enter

```powershell
mysql -u root -p
```

and enter the password you picked previously for the root account.

You are now logged as root in your database management system, you should see a brief message, followed by a prompt

```powershell
mysql >
```

### 3.9.1.3 Installing MariaDB on Linux

1. Install, through your standard package management system (apt or aptitude for debian-based systems, pacman for Arch Linux, etc.), the packages `mysql-client` and `mysql-server` as well as their dependencies\(^{22}\).

2. Open a terminal and type

```
/etc/init.d/mysql status
```

to see if MySQL is running: if you read

```
Active: active (running)
```

then you can move on to the next step, otherwise run (as root)

---


\(^{22}\)Yes, the package is called `mysql-server`, but it actually install the package `mariadb-server-10.1`... So don’t be confused: we are, indeed, installing MariaDB!
3.9 Setting Up Your Work Environment

3. As root, type in your terminal

```
mysql_secure_installation
```

You’ll be asked to provide the current password for the root MySQL user: this password has not been defined yet, so just hit “Enter”. You’ll be asked if you want to set a new password (that you can freely chose, just make sure to memorize it). Then, answer “n” to the question “Remove anonymous users?”, “Y” to “Disallow root login remotely?”, “n” to “Remove test database and access to it?” and finally “Y” to “Reload privilege tables now?”.

4. Still as root, type in your terminal

```
mysql -u root -p
```

and enter the password you picked previously for the root account. You are now logged as root in your database management system: you should see a brief message, followed by a prompt

```
MariaDB [(none)]>
```

### 3.9.2 Creating an user

This step will create a non-root user23 and grant it some rights. Copy-and-paste or type the following three commands, one by one (that is, enter the first one, hit “enter”, enter the second, hit “enter”, etc.):

We first create a new user called `testuser` on our local installation, and give it the password `password`:

```
CREATE USER 'testuser'@'localhost' IDENTIFIED BY 'password';
```

Then, we grant it all the privileges on the databases whose name starts with `HW_`:

```
GRANT ALL PRIVILEGES ON `HW_`.* TO 'testuser'@'localhost';
```

Be careful: backticks (`) are surrounding `HW_` whereas single quotes (') are surrounding `testuser` and `localhost`.

And then we quit the DBMS, using

```
EXIT;
```

The message displayed after the two first commands should be

```
Query OK, 0 rows affected (0.00 sec)
```

and the message displayed after the last command should be

```
Bye
```

---

23By default, MySQL and MariaDB only create a root user with all privileges and no password, but we added a password at the previous step.
3.9.3 Logging-in as testuser

We now log in as the normal user called “testuser”.

Linux users should type as a normal user, i.e., not as root, in their terminal the following, and Windows users should type in their command prompt the following:\n
```
mysql -u testuser -p
```

Enter password as your password. If you are prompted with a message

```
ERROR 1045 (28000): Access denied for user 'testuser'@'localhost' (using password: YES)
```

then you probably typed the wrong password. Otherwise, you should see a welcoming message from MySQL or MariaDB and a prompt.

To save yourself the hassle of typing the password, you can use

```
mysql -u testuser -ppassword
```

or

```
mysql -u testuser -p --password=password
```

If at some point you want to know if you are logged as root or testuser, simply enter
```
\s;
```

3.9.4 Creating our first database

Now, let us create our first schema, our first table, populate it with data, and display various information.

We first create the schema (or database), HW_FirstTest:

```
CREATE DATABASE HW_FirstTest; -- Or CREATE SCHEMA HW_FirstTest;
```

Let us make sure that we created it:

```
SHOW DATABASES;
```

Let us use it:

```
USE HW_FirstTest;
```

And see what it contains now:

```
SHOW TABLES;
```

We now create a table called TableTest, with two integer attributes called Attribute1 and Attribute2:

```
CREATE TABLE TableTest (Attribute1 INT, Attribute2 INT);
```

And can make sure that the table was indeed created:

```
SHOW TABLES;
```

---

\(24\) Provided the working directory is still C:\Program Files\MySQL\MySQL Server 5.7\bin. Cf. https://dev.mysql.com/doc/mysql-windows-excerpt/5.7/en/mysql-installation-windows-path.html to add the MySQL bin directory to your Windows system PATH environment variable.
We can further ask our DBMS to display the structure of the table we just created:

```
DESCRIBE TableTest; -- Can be abbreviated as DESC TableTest;
```

And even ask to get back the code that would create the exact same structure (but without the data!):

```
SHOW CREATE TABLE TableTest;
```

Now, let us populate it with some data:

```
INSERT INTO TableTest
VALUES (1,2),
(3,4),
(5,6);
```

Note that the SQL syntax and your DBMS are completely fine with your statement spreading over multiple lines. Let us now display the data stored in the table:

```
SELECT * FROM TableTest;
```

After that last command, you should see

```
+-----------------+-----------------+
| Attribute1      | Attribute2      |
+-----------------+-----------------+
| 1               | 2               |
| 3               | 4               |
| 5               | 6               |
+-----------------+-----------------+
```

Finally, we can erase the content of the table, then erase ("drop") the table, and finally the schema:

```
DELETE FROM TableTest; -- Delete the rows
DROP TABLE TableTest; -- Delete the table
DROP DATABASE HW_FirstTest; -- Delete the schema
```

You’re all set! All you have to do is to quit, using the command

```
EXIT;
```

**Exercises**

**Exercise 3.1** What does it mean to say that SQL is at the same time a “data definition language” and a “data manipulation language”?

**Exercise 3.2** Name three kind of objects (for lack of a better word) a `CREATE` statement can create.

**Exercise 3.3** Write a SQL statement that adds a primary key constraint to an attribute named `Id` in an already existing table named `STAFF`.

**Exercise 3.4** Complete each row of the following table with either a datatype or two different examples:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>4, -32</td>
<td></td>
</tr>
</tbody>
</table>
### Data type Examples

<table>
<thead>
<tr>
<th>Data type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char(4)</td>
<td>'Train', 'Michelle'</td>
</tr>
<tr>
<td>VarChar(10)</td>
<td>'Train', 'Michelle'</td>
</tr>
<tr>
<td>Bit(4)</td>
<td>TRUE, UNKNOWN</td>
</tr>
</tbody>
</table>

**Exercise 3.5** In the datatype CHAR(3), what does the 3 indicate?

**Exercise 3.6** Explain this query: `CREATE SCHEMA FACULTY;`.

**Exercise 3.7** Explain this query:

```
ALTER TABLE TABLEA
DROP INDEX Attribute1;
```

**Exercise 3.8** If I want to enter January 21, 2016, as a value for an attribute with the DATE datatype, what value should I enter?

**Exercise 3.9** Write a statement that inserts the values "Thomas" and 4 into the table TRAINS.

**Exercise 3.10** If PkgName is the primary key in the table MYTABLE, what can you tell about the number of rows returned by the following statement?

```
SELECT * FROM MYTABLE WHERE PkgName = 'MySQL';
```

**Exercise 3.11** If you want that every time a referenced row is deleted, all the referring rows are deleted as well, what mechanism should you use?

**Exercise 3.12** If a database designer is using the ON UPDATE SET NULL for a foreign key, what mechanism is (s)he implementing (i.e., describe how the database will react a certain operation)?

**Exercise 3.13** If the following is part of the design of a table:

```
FOREIGN KEY (DptNumber) REFERENCES DEPARTMENT(Number)
    ON DELETE SET DEFAULT
    ON UPDATE CASCADE;
```

What happen to the rows whose foreign key DptNumber are set to 3 if the row in the DEPARTMENT table with primary key Number set to 3 is...

1. ... deleted?
2. ... updated to 5?

**Exercise 3.14** If the following is part of the design of a WORKER table:

```
FOREIGN KEY WORKER(DptNumber) REFERENCES DEPARTMENT(DptNumber)
    ON UPDATE CASCADE;
```

What happen to the rows whose foreign key DptNumber are set to 3 if the row in the DEPARTMENT table with primary key Number set to 3 is...

1. ... deleted?
2. ... updated to 5?

**Exercise 3.15** Given a relation TOURIST(Name, EntryDate, Address), write a SQL statement printing the name and address of all the tourists who entered the territory after the 15 September, 2012.
Exercise 3.16  Describe what the star do in the statement

```sql
SELECT ALL * FROM MYTABLE;
```

Exercise 3.17  What is the fully qualified name of an attribute? Give an example.

Exercise 3.18  If DEPARTMENT is a database, what is DEPARTMENT.*?

Exercise 3.19  What is a multi-set? What does it mean to say that SQL treats tables as multisets?

Exercise 3.20  What is the difference between

```sql
SELECT ALL * FROM MYTABLE;
```

and

```sql
SELECT DISTINCT * FROM MYTABLE;
```

How are the results the same? How are they different?

Exercise 3.21  What is wrong with the statement

```sql
SELECT * WHERE Name = 'CS' FROM DEPARTMENT;
```

Exercise 3.22  Write a query that returns the number of row (i.e., of entries, of tuples) in a table named BOOK.

Exercise 3.23  When is it useful to use a select-project-join query?

Exercise 3.24  When is a tuple variable useful?

Exercise 3.25  Write a query that changes the name of the professor whose Login is 'caubert' to 'Hugo Pernot' in the table PROF.

Exercise 3.26  Can an UPDATE statement have a WHERE condition using an attribute that isn’t the primary key? If no, justify, if yes, tell what could happen.

Exercise 3.27  What are the possible meanings or interpretations for a NULL value?

Exercise 3.28  What are the values of the following expressions (i.e., do they evaluate to TRUE, FALSE, or UNKNOWN)?

- TRUE AND FALSE
- TRUE AND UNKNOWN
- NOT UNKNOWN
- FALSE OR UNKNOWN

Exercise 3.29  Write the truth table for AND for the three-valued logic of SQL.

Exercise 3.30  What comparison expression should you use to test if a value is different from NULL?

Exercise 3.31  Explain this query:

```sql
SELECT Login
FROM PROF
WHERE Department IN ( SELECT Major
                        FROM STUDENT
                        WHERE Login = 'jrakesh');
```

Can you rewrite it without nesting queries?
Exercise 3.32  What is wrong with this query?

```
SELECT Name FROM STUDENT
WHERE Login IN
  ( SELECT Code FROM Department WHERE head = 'aturing');
```

Exercise 3.33  Write a query that returns the sum of all the values stored in the Pages attribute of a BOOK table.

Exercise 3.34  Write a query that adds a Pages attribute of type INT into a (already existing) BOOK table.

Exercise 3.35  Write a query that removes the default value for a Pages attribute in a BOOK table.

Exercise 3.36  Under which conditions does SQL allow you to enter the same row in a table twice?

Exercises

Solution to Exercises

Solution 3.1  It can specify the conceptual and internal schema, and it can manipulate the data.

Solution 3.2  Database (schema), table, view, assertion, trigger, etc.

Solution 3.3  `ALTER TABLE STAFF ADD PRIMARY KEY(Id);`

Solution 3.4  

<table>
<thead>
<tr>
<th>Data type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>4, -32</td>
</tr>
<tr>
<td>Char(4)</td>
<td>'trai', 'plol'</td>
</tr>
<tr>
<td>VarChar(10)</td>
<td>'Train', 'Michelle'</td>
</tr>
<tr>
<td>Bit(4)</td>
<td>B'1010', B'0101'</td>
</tr>
<tr>
<td>Boolean</td>
<td>TRUE, UNKNOWN</td>
</tr>
</tbody>
</table>

Solution 3.5  That we can store exactly three characters.

Solution 3.6  It creates a schema, i.e., a database, named Faculty.

Solution 3.7  It removes the UNIQUE constraint on the Attribute1 in the TABLEA table.

Solution 3.8  `DATE '2016-01-21', '2016-01-21', '2016/01/21', '20160121'.`

Solution 3.9  `INSERT INTO TRAINS VALUES('Thomas', 4);`

Solution 3.10  Yes.

Solution 3.11  We should use a referential triggered action clause: `ON DELETE CASCADE`.

Solution 3.12  If the referenced row is updated, then the attribute of the referencing rows are set to NULL.

Solution 3.13  In the referencing rows,

1. the department number is set to the default value.
2. the department number is updated accordingly.
Solution 3.14  
1. This operation is rejected: the row in the DEPARTMENT table with primary key Number set to 3 can’t be deleted if a row in the WORKER table references it. 
2. In the referencing rows, the department number is updated accordingly.

Solution 3.15  
We could use the following:

```sql
SELECT Name, Address
FROM TOURIST
WHERE EntryDate > DATE'2012-09-15';
```

Solution 3.16  
It selects all the attributes, it is a wildcard.

Solution 3.17  
The name of the relation with the name of its schema and a period beforehand. 
An example would be EMPLOYEE. Name.

Solution 3.18  
All the tables in that database.

Solution 3.19  
A set where the same value can occur twice. The same row can occur twice in a table.

Solution 3.20  
They both select all the rows in the MYTABLE table, but ALL will print the duplicate values, whereas DISTINCT will print them only once.

Solution 3.21  
You cannot have the WHERE before FROM.

Solution 3.22  
```sql
SELECT COUNT(*) FROM BOOK;
```

Solution 3.23  
We use those query that projects on attributes using a selection and join conditions when we need to construct for information based on pieces of data spread in multiple tables.

Solution 3.24  
It makes the distinction between two different rows of the same table, it is useful when we want to select a tuple in a relation that is in a particular relation with a tuple in the same relation. Quoting https://stackoverflow.com/a/7698796/:

They are useful for saving typing, but there are other reasons to use them:

- If you join a table to itself you must give it two different names otherwise referencing the table would be ambiguous.
- It can be useful to give names to derived tables, and in some database systems it is required... even if you never refer to the name.

Solution 3.25  
We could use the following:

```sql
UPDATE PROF SET Name = 'Hugo Pernot'
WHERE Login = 'caubert';
```

Solution 3.26  
Yes, we can have select condition that does not use primary key. In that case, it could be the case that we update more than one tuple with such a command (which is not necessarily a bad thing).

Solution 3.27  
Unknown value, unavailable / withheld, N/A.

Solution 3.28  
- TRUE AND FALSE → FALSE
- TRUE AND UNKNOWN → UNKNOWN
- NOT UNKNOWN → UNKNOWN
- FALSE OR UNKNOWN → FALSE

Solution 3.29  
- TRUE AND TRUE → TRUE
3.9 Setting Up Your Work Environment

- TRUE AND FALSE → FALSE
- TRUE AND UNKNOWN → UNKNOWN
- FALSE AND FALSE → FALSE
- UNKNOWN AND UNKNOWN → UNKNOWN
- FALSE AND UNKNOWN → FALSE
- The other cases can be deduced by symmetry.

For a more compact presentation, refer to (3.29truth_tables).

Solution 3.30 IS NOT

Solution 3.31 It list the login of the professors teaching in the department where the student whose login is “jrakesh” is majoring. It can be rewritten as

```sql
SELECT PROF.Login
FROM PROF, STUDENT
WHERE Department = Major
AND STUDENT.Login = 'jrakesh';
```

Solution 3.32 It tries to find a Login in a Code.

Solution 3.33 SELECT SUM(Pages) FROM BOOK;

Solution 3.34 ALTER TABLE BOOK ADD COLUMN Pages INT;

Solution 3.35 ALTER TABLE BOOK ALTER COLUMN Pages DROP DEFAULT;

Solution 3.36 Essentially, if there are no primary key in the relation, and if no attribute has the UNIQUE constraint. Cf. also Solution 3.36 ().

Problems

Problem 3.1 (Discovering the documentation) The goal of this problem is to learn where to find the documentation for your DBMS, and to understand how to read the syntax of SQL commands.

You can consult your textbook, (Elmasri and Navathe 2010, Table 5.2, p. 140) or (Elmasri and Navathe 2015, Table 7.2, p. 235), for a very quick summary of the most common commands. Make sure you are familiar with the Backus–Naur form (BNF) notation commonly used:

- non-terminal symbols (i.e., variables, parameters) are enclosed in angled brackets, `<…>`
- optional parts are shown in square brackets, `[…]`
- repetitons are shown in braces `{…}`
- alternatives are shown in parenthesis and separated by vertical bars, `(...|...|...)`

The most complete lists of commands are probably at

- https://mariadb.com/kb/en/library/sql-statements/ and
Those are the commands implemented in the DBMS you are actually using. Since there are small variations from one implementation to the other, it is better to take one of this link as a reference in the future.

**Problem 3.2 (Creating and using a simple table in SQL)** This problem will guide you in manipulating a very simple table in SQL.

**Pb 3.2 – Question 1** Log in as testuser, create a database named HW_Address, use it, and create two tables:

```sql
CREATE TABLE NAME(
    FName VARCHAR(15),
    LName VARCHAR(15),
    Id INT,
    PRIMARY KEY(Id)
);

CREATE TABLE ADDRESS(
    StreetName VARCHAR(15),
    Number INT,
    Habitant INT,
    PRIMARY KEY(StreetName, Number)
);
```

**Pb 3.2 – Question 2** Observe the output produced by the command `DESC ADDRESS`.

**Pb 3.2 – Question 3** Add a foreign key to the ADDRESS table, using

```sql
ALTER TABLE ADDRESS
    ADD FOREIGN KEY (Habitants)
    REFERENCES NAME(Id);
```

And observe the new output produced by the command `DESC ADDRESS`.

Is it what you would have expected? How informative is it? Can you think of a command that would output more detailed information, including a reference to the existence of the foreign key?

**Pb 3.2 – Question 4** Draw the relational model corresponding to that database, including the primary, as well as foreign, keys.

**Pb 3.2 – Question 5** Add some data in the NAME table:

```sql
INSERT INTO NAME VALUES ('Barbara', 'Liskov', 003);
INSERT INTO NAME VALUES ('Tuong Lu', 'Kim', 004);
INSERT INTO NAME VALUES ('Samantha', NULL, 080);
```

What command can you use to display this information back? Do you notice anything regarding the values we entered for the Id attribute?

**Pb 3.2 – Question 6** Add some data into the ADDRESS table:

```sql
INSERT INTO ADDRESS VALUES
```
What difference do you note with the insertions we made in the NAME table. Which syntax seem more easy to you?

Pb 3.2 – Question 7 Write a SELECT statement that returns the Id number of the person whose first name is “Samantha”.

Pb 3.2 – Question 8 Write a statement that violate the entity integrity constraint. What is the error message returned?

Pb 3.2 – Question 9 Execute an UPDATE statement that violate the referential integrity constraint. What is the error message returned?

Pb 3.2 – Question 10 Write a statement that violate another kind of constraint. Explain what constraint you are violating, and explain the error message.

Problem 3.3 (Duplicate rows in SQL) Log in as testuser and create a database HW_REPETITION. Create in that database a table (I will assume you named it EXAMPLE in the following, but you are free to name it the way you want) with at least two attributes that have different data types. Don’t declare a primary key yet. Answer the following:

Pb 3.3 – Question 1 Add a tuple to your table using

```sql
INSERT INTO EXAMPLE VALUES(X, Y);
```

where X and Y are values with the right datatype. Try to add this tuple again. What do you observe? (You can use SELECT * FROM EXAMPLE; to observe what is stored in this table.)

Pb 3.3 – Question 2 Alter your table to add a primary key, using

```sql
ALTER TABLE EXAMPLE ADD PRIMARY KEY (Attribute);
```

where Attribute is the name of the attribute you want to be a primary key. What do you observe?

Pb 3.3 – Question 3 Empty your table using

```sql
DELETE FROM EXAMPLE;
```

and alter your table to add a primary key, using the command we gave at the previous step. What do you observe?

Pb 3.3 – Question 4 Try to add the same tuple twice. What do you observe?

Problem 3.4 (Revisiting the PROF table) Create the PROF, STUDENT, DEPARTMENT and GRADE tables as in the “Constructing and populating a new example” section. Populate them with some data (copy it from the notes or invent it, it won’t matter).

Pb 3.4 – Question 1 Draw the complete relational model for this database (i.e., for the PROF, DEPARTMENT, STUDENT and GRADE relations).

Pb 3.4 – Question 2 Create and populate a LECTURE table as follows:
• It should have four attributes, Name, Instructor, Code, and Year, of type VARCHAR(25) for the two first, CHAR(5), and Year(4).
• The Year and Code attributes should be the primary key (yes, have two attributes be the primary key).
• The Instructor attribute should be a foreign key referencing the Login attribute in PROF.
• Populate the LECTURE table with some made-up data.

Try to think about some of the weaknesses of this representation (for instance, can it accommodate two instructors for the same class?), and write down two possible scenarios where this schema would not be appropriate.

Pb 3.4 – Question 3 The GRADE table had some limitations, too, since every student could have only one grade. Add two columns to the GRADE table, using

```
ALTER TABLE GRADE
ADD COLUMN LectureCode CHAR(5),
ADD COLUMN LectureYear YEAR(4);
```

and add a foreign key:

```
ALTER TABLE GRADE
ADD FOREIGN KEY (LectureYear, LectureCode)
REFERENCES LECTURE(Year, Code);
```

Use DESCRIBE and SELECT to observe the schema of the GRADE table and its rows. Is that what you would have expected?

Pb 3.4 – Question 4 Update the tuples in GRADE with some made-up data: now, every row should contain, on top of a login and a grade, a lecture year and a lecture code.

Pb 3.4 – Question 5 Update the relational model you previously drawn to reflect the new situation of your tables.

Pb 3.4 – Question 6

Write SELECT statements answering the following questions (where PROF.Name, LECTURE.Name, YYYY, LECTURE.Code and STUDENT.Login should be relevant values considering your data):

1. “Could you give me the logins and grades of the students who took LECTURE.Name in ‘YYYY’?”
2. “Could you list the instructors who taught in year YYYY?” (and, please, avoid duplicates)
3. “Could your list the name and grade of all the student who took class LECTURE.Code (no matter the year)?”
4. “Could you tell me which years was the class LECTURE.Code taught?”
5. “Could you list the classes taught the same year as class LECTURE.Code?”
6. “Could you print the name of the students who registered after STUDENT.Login?”
7. “How many departments’ heads are teaching this year?”

Problem 3.5 (TRAIN table and more advanced SQL coding) Look at the SQL code below, and then answer the following questions.
CREATE TABLE TRAIN(
    Id VARCHAR(30),
    Model VARCHAR(30),
    ConstructionYear YEAR(4)
);

CREATE TABLE CONDUCTOR(
    Id VARCHAR(20),
    Name VARCHAR(20),
    ExperienceLevel VARCHAR(20)
);

CREATE TABLE ASSIGNED_TO(
    TrainId VARCHAR(20),
    ConductorId VARCHAR(20),
    Day DATE,
    PRIMARY KEY(TrainId, ConductorId)
);

Pb 3.5 – Question 1 Modify the CREATE statement that creates the TRAIN table (l. 1–5), so that Id would be declared as the primary key. You can write only the line(s) that need to change.

Pb 3.5 – Question 2 Write an ALTER statement that makes Id become the primary key of the CONDUCTOR table.

Pb 3.5 – Question 3 Modify the CREATE statement that creates the ASSIGNED_TO table (l. 13–18), so that it has two foreign keys: ConductorId would be referencing the Id attribute in CONDUCTOR and TrainId would be referencing the Id attribute in TRAIN. You can write only the line(s) that need to change.

Pb 3.5 – Question 4 Write INSERT statements that insert one tuple of your invention in each relation (without NULL values). Your statements should respect all the constraints (including the ones we added at the previous questions) and result in actual insertions. (Remember that four digits is a valid value for an attribute with the YEAR(4) datatype.)

Pb 3.5 – Question 5 Write a statement that sets the value of the ExperienceLevel attribute to “Senior” in all the tuples where the Id attribute is “GP1029” in the CONDUCTOR relation.

Pb 3.5 – Question 6 For each of the following questions, write a SELECT statement that would answer it:

1. "What are the identification numbers of the trains?"
2. "What are the names of the conductors with a "Senior" experience level?"
3. "What are the construction years of the "Surfliner" and "Regina" models that we have?"
4. "What is the id of the conductor that was responsible of the train referenced "K-13" on 2015/12/14?"
5. "What are the models that were ever conducted by the conductor whose id is "GP1029"?"
Problem 3.6 (*Insert, Select and Project in the COFFEE database*) Suppose we have the relational model depicted below, with the indicated data in it:

**COFFEE**

<table>
<thead>
<tr>
<th>Ref</th>
<th>Origin</th>
<th>TypeOfRoast</th>
<th>PricePerPound</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Brazil</td>
<td>Light</td>
<td>8.90</td>
</tr>
<tr>
<td>121</td>
<td>Bolivia</td>
<td>Dark</td>
<td>7.50</td>
</tr>
<tr>
<td>311</td>
<td>Brazil</td>
<td>Medium</td>
<td>9.00</td>
</tr>
<tr>
<td>221</td>
<td>Sumatra</td>
<td>Dark</td>
<td>10.25</td>
</tr>
</tbody>
</table>

**CUSTOMER**

<table>
<thead>
<tr>
<th>CardNo</th>
<th>Name</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Bob Hill</td>
<td><a href="mailto:b.hill@isp.net">b.hill@isp.net</a></td>
</tr>
<tr>
<td>002</td>
<td>Ana Swamp</td>
<td><a href="mailto:swampa@nca.edu">swampa@nca.edu</a></td>
</tr>
<tr>
<td>003</td>
<td>Mary Sea</td>
<td><a href="mailto:brig@gsu.gov">brig@gsu.gov</a></td>
</tr>
<tr>
<td>004</td>
<td>Pat Mount</td>
<td><a href="mailto:pmount@fai.fr">pmount@fai.fr</a></td>
</tr>
</tbody>
</table>

**SUPPLY**

<table>
<thead>
<tr>
<th>Provider</th>
<th>Coffee</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coffee Unl.</td>
<td>001</td>
</tr>
<tr>
<td>Coffee Unl.</td>
<td>121</td>
</tr>
<tr>
<td>Coffee Exp.</td>
<td>311</td>
</tr>
<tr>
<td>Johns &amp; Co.</td>
<td>221</td>
</tr>
</tbody>
</table>

**PROVIDER**

<table>
<thead>
<tr>
<th>Name</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coffee Unl.</td>
<td><a href="mailto:bob@cofunl.com">bob@cofunl.com</a></td>
</tr>
<tr>
<td>Coffee Exp.</td>
<td><a href="mailto:pat@coffeex.dk">pat@coffeex.dk</a></td>
</tr>
<tr>
<td>Johns &amp; Co.</td>
<td>NULL</td>
</tr>
</tbody>
</table>

In the following, we will assume that this model was implemented in a DBMS (MySQL or MariaDB), the primary keys being COFFEE.Ref, CUSTOMER.CardNo, SUPPLY.Provider and SUPPLY.Coffee, and PROVIDER.Name, and the foreign keys being as follow:

<table>
<thead>
<tr>
<th>FavCoffee in the CUSTOMER relation</th>
<th>Ref in the COFFEE relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provider in the SUPPLY</td>
<td>Name in the PROVIDER relation</td>
</tr>
<tr>
<td>Coffee in the SUPPLY</td>
<td>Ref in the COFFEE relation</td>
</tr>
</tbody>
</table>

You will be asked to read and write SQL commands. You should assume that

1. Datatype doesn’t matter: we use only strings and appropriate numerical datatypes.
2. Every statement respects SQL’s syntax (there’s no “a semi-colon is missing” trap).
3. None of those commands are actually executed: the data is always in the state depicted above, you are asked to answer “what if” questions.

You can use COFFEE.1 to denote the first tuple (or row) in COFFEE, and similarly for other relations and tuples (so that, for instance SUPPLY.4 corresponds to "Johns & Co.", 221).

**Pb 3.6 – Question 1** Draw the relational model of this table.

**Pb 3.6 – Question 2** Determine if the following insertion statements would violate the the entity integrity constraint, ("primary key cannot be NULL and should be unique), the referential integrity constraint ("the foreign key must refer to something that exists"), if there would be some other kind of error (ignoring the plausability / revelance of inserting that tuple), or if it would result in successful insertion.

```sql
1. INSERT INTO CUSTOMER VALUES(005, 'Bob Hill', NULL, 001);
2. INSERT INTO COFFEE VALUES(002, "Peru", "Decaf", 3.00);
3. INSERT INTO PROVIDER VALUES(NULL, "contact@localcof.com");
4. INSERT INTO SUPPLY VALUES("Johns Co.", 121);
5. INSERT INTO SUPPLY VALUES("Coffee Unl.", 311, 221);
```

**Pb 3.6 – Question 3** Assuming that the referential triggered action clause ON UPDATE CASCADE is used for every foreign keys in this database, list the tuples modified by the following statements:

```sql
1. UPDATE CUSTOMER SET FavCoffee = 001
   WHERE CardNo = 001;
2. UPDATE COFFEE SET TypeOfRoast = 'Decaf'
   WHERE Origin = 'Brazil';
3. UPDATE PROVIDER SET Name = 'Coffee Unlimited'
   WHERE Name = 'Coffee Unl.';
4. UPDATE COFFEE SET PricePerPound = 10.00
   WHERE PricePerPound > 10.00;
```

**Pb 3.6 – Question 4** Assuming that the referential triggered action clause ON DELETE CASCADE is used for every foreign keys in this database, list the tuples modified by the following statements:

```sql
1. DELETE FROM CUSTOMER
   WHERE Name LIKE '%%S%%';
2. DELETE FROM COFFEE
   WHERE Ref = 001;
3. DELETE FROM SUPPLY
   WHERE Provider = 'Coffee Unl.'
   AND Coffee = 001;
4. DELETE FROM PROVIDER
   WHERE Name = 'Johns & Co.';
```
**Pb 3.6 – Question 5** Starting here, assume that there is more data in our table than what was given at the beginning of the problem. Write queries that answer the following questions:

1. "What are the origins of your dark coffees?"
2. "What is the reference of Bob’s favorite coffee?" (nota: it doesn’t matter if you return the favorite coffee of all the Bobs in the database.)
3. “What are the names of the providers who didn’t give their email?”
4. "How many coffees does Johns & co. provide us with?"
5. "What are the names of the providers of my dark coffees?"

**Problem 3.7 (Select queries for the COMPUTER table)** Consider the following SQL code:

```sql
/* code/sql/COMPUTER.sql */
CREATE TABLE COMPUTER(
    Id VARCHAR(20) PRIMARY KEY,
    Model VARCHAR(20)
);

CREATE TABLE PRINTER(
    Id VARCHAR(20) PRIMARY KEY,
    Model VARCHAR(20)
);

CREATE TABLE CONNEXION(
    Computer VARCHAR(20),
    Printer VARCHAR(20),
    PRIMARY KEY (Computer, Printer),
    FOREIGN KEY (Computer) REFERENCES COMPUTER(Id),
    FOREIGN KEY (Printer) REFERENCES PRINTER(Id)
);

INSERT INTO COMPUTER VALUES
    ('A', 'DELL A'),
    ('B', 'HP X'),
    ('C', 'ZEPTO D'),
    ('D', 'MAC Y');

INSERT INTO PRINTER VALUES
    ('12', 'HP-140'),
    ('13', 'HP-139'),
    ('14', 'HP-140'),
    ('15', 'HP-139');

INSERT INTO CONNEXION VALUES
    ('A', '12'),
    ('A', '13'),
    ('B', '13'),
    ('C', '14');
```
Write a query that returns ... (in parenthesis, the values returned in this set-up, but you have to be general)

1. ... the number of computers connected to the printer whose Id is 13 (i.e., 2).
2. ... the number of different models of printers (i.e., 2).
3. ... the model(s) of the printer connected to the computer whose Id is 'A' (i.e., 'HP-140' and 'HP-139').
4. ... the id of the computer(s) not connected to any printer (i.e., 'D').

**Problem 3.8 (Improving a role-playing game)** A friend of yours want you to review and improve the code for a role-playing game.

The original idea was that each character should have a name, a class (e.g., Bard, Assassin, Druid), a certain amount of experience, a level, one or more weapons (providing bonuses) and can complete quests. A quest have a name, and rewards the characters that completed it with a certain amount of experience, and sometimes (but rarely) with a special item.

Your friend came up with the following code:

```sql
CREATE TABLE CHARACTER(
    Name VARCHAR(30) PRIMARY KEY,
    Class VARCHAR(30),
    XP INT,
    LVL INT,
    Weapon_Name VARCHAR(30),
    Weapon_Bonus INT,
    Quest_Completed VARCHAR(30)
);  
CREATE TABLE QUEST(
    Id VARCHAR(20) PRIMARY KEY,
    Completed_By VARCHAR(30),
    XP_Gained INT,
    Special_Item VARCHAR(20),
    FOREIGN KEY (Completed_By) REFERENCES CHARACTER(Name)
);  
ALTER TABLE CHARACTER
    ADD FOREIGN KEY (Quest_Completed) REFERENCES QUEST(Id);  
```

But there are several problems.

- As of now, a character can have only one weapon. All the attempts to “hack” the CHARACTER table to add an arbitrary number of weapons ended up creating horrible messes.
- Every time a character completes a quest, a copy of the quest must be created. Your friend is not so sure why, but nothing else works. Also it seems that a character can complete only one quest, but your friend is not so sure about that.
- It would be nice to be able to store features that are tied to the class, and not to the character, like the bonus they provide and the associated element (e.g., all bards use fire, all assassins use wind, etc.). But you friend simply can’t figure out how to do that.
Can you provide a relational model (no need to write the SQL code, but remember to indicate the primary and foreign keys) that would solve all of your friend’s troubles?

**Problem 3.9 (A simple database for books)** Consider the following code:

```sql
/*
code/sql/SIMPLE_BOOK.sql

The following is in case you want to run your program on your installation:

-- DROP SCHEMA HW_SIMPLE_BOOK;
CREATE SCHEMA HW_SIMPLE_BOOK;
USE HW_SIMPLE_BOOK;

*/

CREATE TABLE AUTHOR(
    FName VARCHAR(30),
    LName VARCHAR(30),
    Id INT PRIMARY KEY
);

CREATE TABLE PUBLISHER(
    Name VARCHAR(30),
    City VARCHAR(30),
    PRIMARY KEY (Name, City)
);

CREATE TABLE BOOK(
    Title VARCHAR(30),
    Pages INT,
    Published DATE,
    PublisherName VARCHAR(30),
    PublisherCity VARCHAR(30),
    FOREIGN KEY (PublisherName, PublisherCity)
        REFERENCES PUBLISHER(Name, City),
    Author INT,
    FOREIGN KEY (Author)
        REFERENCES AUTHOR(Id),
    PRIMARY KEY (Title, Published)
);

INSERT INTO AUTHOR VALUES
    ("Virginia", "Wolve", 01),
    ("Paul", "Bryant", 02),
    ("Samantha", "Carey", 03);

INSERT INTO PUBLISHER VALUES
```
The values inserted in the database will guide some examples, but you should assume there is more data than what we inserted. In this long problem, you will be asked to write commands to select, update, delete, insert data, and to enhance the model.

1. Write a command that selects...
   a) the title of all the books.
   b) the (distinct) name of the publishers.
   c) the title and publication date of the books published since January 31, 2012.
   d) the first and last names of the authors published by Gallimard (no matter the city).
   e) the first and last names of the authors who were not published by an editor in New-York.
   f) the id of the authors who published a book whose name starts with "Where".
   g) the total number of pages in the database.
   h) the number of pages in the longest book written by the author whose last name is "Wolve".
   i) the title of the books published in the XIXth century.

2. Write a command that updates the title of all the books written by the author whose id is 1 to "BANNED". Is there any reason for this command to be rejected by the system? If yes, explain which one.

3. Write one or multiple commands that would delete the author whose id is 3, and all the book written by that author. Make sure you don’t violate any foreign key constraint.

4. Write a command that would create a table used to record the awards granted to authors for particular books. You should assume that each award has its own name, is awarded every year, and that it is awarded to an author for a particular book. Pick appropriate attributes, datatypes, primary as well as foreign keys, and avoid all redundancy.

5. Draw the relational model of the database you created (i.e., including all the relations given in the code and the one you added).

6. Discuss two limitations of the model and how to improve it.

---

25You can use the DATE datatype to store a year.
Solution to Selected Problems

Solution to Problem 3.2 (Creating and using a simple table in SQL) This problem is supposed to be a straightforward application of what we studied in class. Look back at Setting Up Your Work Environment if you feel like you are stuck before rushing to this solution.

**Pb 3.2 – Solution to Q. 1** We simply log-in as indicated in the “Login as testuser” Section. Then, we enter

```sql
1. CREATE DATABASE HW_Address;
2. USE HW_Address;
```

We then create the tables as in the problem.

**Pb 3.2 – Solution to Q. 2** Omitting the Extra column, we have:

MariaDB [HW_Address]> DESC ADDRESS;
```
+------------+-------------+------+-----+---------+
| Field      | Type        | Null | Key | Default |
+------------+-------------+------+-----+---------+
| StreetName | varchar(15) | NO   | PRI | NULL    |
| Number     | int(11)     | NO   | PRI | NULL    |
| Habitants  | int(11)     | YES  | MUL | NULL    |
+------------+-------------+------+-----+---------+
```

**Pb 3.2 – Solution to Q. 3** We add the foreign key, then (still omitting the Extra column):

MariaDB [HW_Address]> DESC ADDRESS;
```
+------------+-------------+------+-----+---------+
| Field      | Type        | Null | Key | Default |
+------------+-------------+------+-----+---------+
| StreetName | varchar(15) | NO   | PRI | NULL    |
| Number     | int(11)     | NO   | PRI | NULL    |
| Habitants  | int(11)     | YES  | MUL | NULL    |
+------------+-------------+------+-----+---------+
```

The only difference is the MUL value, which is a bit surprising: quoting https://dev.mysql.com/doc/refman/8.0/en/show-columns.html,

If Key is MUL, the column is the first column of a nonunique index in which multiple occurrences of a given value are permitted within the column.

In other word, this doesn’t really carry any information about the fact that ADDRESS.Habitants is now a foreign key referencing NAME.Id. A way of displaying information about that foreign key is using **SHOW CREATE TABLE**:

MariaDB [HW_Address]> SHOW CREATE TABLE ADDRESS;
```
+---------+----------------------+
| Table   | Create Table         |
+---------+----------------------+
| ADDRESS | CREATE TABLE `ADDRESS` ( `StreetName` varchar(15) NOT NULL, `Number` int(11) NOT NULL, |
```

```sql
MariaDB [HW_Address]> SHOW CREATE TABLE ADDRESS;
```
```sql
+---------+----------------------+
| Table   | Create Table         |
+---------+----------------------+
| ADDRESS | CREATE TABLE `ADDRESS` ( `StreetName` varchar(15) NOT NULL, `Number` int(11) NOT NULL, |
```

```sql
MariaDB [HW_Address]> SHOW CREATE TABLE ADDRESS;
```
```sql
+---------+----------------------+
| Table   | Create Table         |
+---------+----------------------+
| ADDRESS | CREATE TABLE `ADDRESS` ( `StreetName` varchar(15) NOT NULL, `Number` int(11) NOT NULL, |
```

```sql
MariaDB [HW_Address]> SHOW CREATE TABLE ADDRESS;
```
```sql
+---------+----------------------+
| Table   | Create Table         |
+---------+----------------------+
| ADDRESS | CREATE TABLE `ADDRESS` ( `StreetName` varchar(15) NOT NULL, `Number` int(11) NOT NULL, |
```

```sql
MariaDB [HW_Address]> SHOW CREATE TABLE ADDRESS;
```
```sql
+---------+----------------------+
| Table   | Create Table         |
+---------+----------------------+
| ADDRESS | CREATE TABLE `ADDRESS` ( `StreetName` varchar(15) NOT NULL, `Number` int(11) NOT NULL, |
```

```sql
MariaDB [HW_Address]> SHOW CREATE TABLE ADDRESS;
```
```sql
+---------+----------------------+
| Table   | Create Table         |
+---------+----------------------+
| ADDRESS | CREATE TABLE `ADDRESS` ( `StreetName` varchar(15) NOT NULL, `Number` int(11) NOT NULL, |
```

```sql
MariaDB [HW_Address]> SHOW CREATE TABLE ADDRESS;
```
```sql
+---------+----------------------+
| Table   | Create Table         |
+---------+----------------------+
| ADDRESS | CREATE TABLE `ADDRESS` ( `StreetName` varchar(15) NOT NULL, `Number` int(11) NOT NULL, |
```

```sql
MariaDB [HW_Address]> SHOW CREATE TABLE ADDRESS;
```
```sql
+---------+----------------------+
| Table   | Create Table         |
+---------+----------------------+
| ADDRESS | CREATE TABLE `ADDRESS` ( `StreetName` varchar(15) NOT NULL, `Number` int(11) NOT NULL, |
```

```sql
MariaDB [HW_Address]> SHOW CREATE TABLE ADDRESS;
```
```sql
+---------+----------------------+
| Table   | Create Table         |
+---------+----------------------+
| ADDRESS | CREATE TABLE `ADDRESS` ( `StreetName` varchar(15) NOT NULL, `Number` int(11) NOT NULL, |
```

```sql
MariaDB [HW_Address]> SHOW CREATE TABLE ADDRESS;
```
```sql
+---------+----------------------+
| Table   | Create Table         |
+---------+----------------------+
| ADDRESS | CREATE TABLE `ADDRESS` ( `StreetName` varchar(15) NOT NULL, `Number` int(11) NOT NULL, |
```
`Habitants` int(11) DEFAULT NULL,
PRIMARY KEY (`StreetName`, `Number`),
KEY `Habitants` (`Habitants`),
CONSTRAINT `ADDRESS_ibfk_1` FOREIGN KEY (`Habitants`) REFERENCES
`NAME` (`Id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 |
+
|-----------------+-------------------------------|
| 1 row in set (0.01 sec) |

**Pb 3.2 – Solution to Q. 4** We have.

**NAME**

<table>
<thead>
<tr>
<th>FName</th>
<th>LName</th>
<th>Id</th>
</tr>
</thead>
</table>

**ADDRESS**

<table>
<thead>
<tr>
<th>StreetName</th>
<th>Number</th>
<th>Habitants</th>
</tr>
</thead>
</table>

**Pb 3.2 – Solution to Q. 5** To display the information back, we can use

```sql
SELECT * FROM NAME;
```

We can remark that the Id attributes lost their leading zeros\(^{26}\).

**Pb 3.2 – Solution to Q. 6** This syntax is better for “bulk insertion”, since it allows to write less command, and to focus on the data to insert. However, if an error occurs, then nothing gets inserted.

**Pb 3.2 – Solution to Q. 7**

```sql
SELECT Id FROM NAME WHERE FName = 'Samantha';
```

**Pb 3.2 – Solution to Q. 8** The command

```sql
INSERT INTO NAME VALUES ('Maria', 'Kashi', NULL);
```

returns

ERROR 1048 (23000): Column 'Id' cannot be null

Another way of violating the entity integrity constraint is

```sql
INSERT INTO NAME VALUES ('Maria', 'Kashi', 80);
```

which returns

ERROR 1062 (23000): Duplicate entry '80' for key 'PRIMARY'

**Pb 3.2 – Solution to Q. 9** The command

```sql
UPDATE ADDRESS SET Habitants = 340 WHERE Number = 120;
```

returns

ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fails ('HW_Address'.'ADDRESS', CONSTRAINT `ADDRESS_ibfk_1` FOREIGN KEY ('Habitants') REFERENCES `NAME` (`Id`))

\(^{26}\)https://en.wikipedia.org/wiki/Leading_zero
Pb 3.2 – Solution to Q. 10 The query

\texttt{INSERT INTO NAME VALUE ('Hi');}

returns

\texttt{ERROR 1136 (21S01): Column count doesn't match value count at row 1}

I was violating an implicit constraint, trying to insert a row with fewer values than there are attributes in the table. Note that

\texttt{INSERT INTO ADDRESS VALUES ('Maria', 'Random', 98);}

is a violation of explicit constraint, which is that the value must match the domain (i.e., datatype) of the attribute where they are inserted. However, MySQL or MariaDB doesn’t return an error, and simply replace 'Random' with 0.

------------------------

Solution to Problem 3.3 (Duplicate rows in SQL) As a preamble, we create our own table:

\texttt{CREATE SCHEMA HW_REPETITION;}

\texttt{USE HW_REPETITION;}

\texttt{CREATE TABLE EXAMPLE(}
\hspace{1cm} \texttt{X VARCHAR(15),}
\hspace{1cm} \texttt{Y INT}
\texttt{);}\

Pb 3.3 – Solution to Q. 1 The command is

\texttt{INSERT INTO EXAMPLE VALUES('Train', 4);}\

If we execute this command twice, then SQL is OK with it, and insert the same tuple twice.

\texttt{SELECT * FROM EXAMPLE;}

displays

\texttt{+-------+------+}
\hspace{1cm} \texttt{| X | Y |}
\texttt{+-------+------+}
\hspace{1cm} \texttt{| Train | 4 |}
\hspace{1cm} \texttt{| Train | 4 |}
\texttt{+-------+------+}

This is an illustration of the fact that a table in sql is not a set, as opposed to a relation in the relation model.

Pb 3.3 – Solution to Q. -.# - The command

---{.sqlmysql}
\texttt{ALTER TABLE EXAMPLE ADD PRIMARY KEY (X);}
---
We tried to declare that `X` was a primary key, but SQL disagreed, since two rows have the same value for that attribute.

**Pb 3.3 – Solution to Q. 2** Once the table is empty, `X` indeed is a primary key, so SQL stops complaining and let us make it a primary key.

**Pb 3.3 – Solution to Q. 3** We can’t introduce the same value twice:

```sql
INSERT INTO EXAMPLE VALUES ('Train', 4);
```

returns, the second time:

```sql
ERROR 1062 (23000): Duplicate entry 'Train' for key 'PRIMARY'
```

which is, by the way, exactly the same error message as when we tried to add the primary key in the first place!

---

**Solution to Problem 3.4 (Revisiting the PROF table)** For Question 1 and 5, we obtain:

```sql
    PROF
    Login Name Department
    DEPARTMENT
    Code Name Head
    LECTURE
    Code Year Name Instructor
    STUDENT
    Login Name Registered Major
    GRADE
    Login Grade LectureCode LectureYear
```

For the other questions, refer to this code.

```sql
/* code/sql/LECTURE.sql */
-- Question 2
```
CREATE TABLE LECTURE(
    Name VARCHAR(25),
    Instructor VARCHAR(25),
    Year YEAR(4),
    Code CHAR(5),
    PRIMARY KEY(Year, Code),
    FOREIGN KEY (Instructor) REFERENCES PROF/Login)
);

INSERT INTO LECTURE VALUES
    ('Intro to CS', 'caubert', 2017, '1304'),
    ('Intro to Algebra', 'perdos', 2017, '1405'),
    ('Intro to Cyber', 'aturing', 2017, '1234');

/* This representation can not handle the following situations:*/
/* - If multiple instructors teach the same class,*/
/* - If the lecture is taught more than once a year (either because it */
   /* is taught in the Fall, Spring and Summer, or if multiple sections */
   /* are offered at the same time),*/
/* - If a lecture is cross-listed, then some duplication of */
   /* information will be needed.*/

/* Question 3*/
ALTER TABLE GRADE
    ADD COLUMN LectureCode CHAR(5),
    ADD COLUMN LectureYear YEAR(4);

DESCRIBE GRADE;

SELECT * FROM GRADE;

ALTER TABLE GRADE
    ADD FOREIGN KEY (LectureYear, LectureCode)
    REFERENCES LECTURE(Year, Code);

/* The values for LectureCode and LectureYear are set to NULL in all */
/* the tuples.*/

/* Question 4*/
UPDATE GRADE SET LectureCode = '1304', LectureYear = 2017
    WHERE Login = 'jrakesh'
        AND Grade = '2.85';

UPDATE GRADE SET LectureCode = '1405', LectureYear = 2017
    WHERE Login = 'svlatka'
          OR (Login = 'jrakesh' AND Grade = '3.85');

UPDATE GRADE SET LectureCode = '1234', LectureYear = 2017
    WHERE Login = 'aalyx'
OR Login = 'cjoella';

-- Question 6
SELECT Login, Grade
FROM GRADE
WHERE Lecturecode='1304'
AND LectureYear = '2017';

SELECT DISTINCT Instructor
FROM LECTURE
WHERE Year = 2017;

SELECT Name, Grade
FROM STUDENT, GRADE
WHERE GRADE.LectureCode = 1405
AND STUDENT.Login = GRADE.Login;

SELECT Year
FROM LECTURE
WHERE Code = '1234';

SELECT Name
FROM LECTURE
WHERE Year IN
(SELECT Year
FROM LECTURE
WHERE CODE = '1234');

SELECT B.name
FROM STUDENT AS A, STUDENT AS B
WHERE A.Name = 'Ava Alyx'
AND A.Registered > B.Registered;

SELECT COUNT(DISTINCT PROF.Name) AS 'Head Teaching This Year'
FROM LECTURE, DEPARTMENT, PROF
WHERE Year = 2017
AND Instructor = Head
AND Head = PROF.Login;

Solution to Problem 3.5 (TRAIN table and more advanced SQL coding) Here is some code, commented.

/* code/sql/TRAIN.sql */

CREATE TABLE TRAIN(
   Id VARCHAR(30) PRIMARY KEY,
   Model VARCHAR(30),
   ConstructionYear YEAR(4)
);
Solution to Problem 3.6 (Insert, Select and Project in the COFFEE database)

Here is some code, commented.
/* code/sql/HW_DB_COFFEE.sql */

DROP SCHEMA HW_DB_COFFEE;

CREATE SCHEMA HW_DB_COFFEE;

USE HW_DB_COFFEE;

CREATE TABLE COFFEE(
    Ref VARCHAR(30) PRIMARY KEY,
    Origin VARCHAR(30),
    TipoDeTorrefaccio VARCHAR(30),
    PricePerPound DOUBLE
);

CREATE TABLE CUSTOMER(
    CardNo VARCHAR(30) PRIMARY KEY,
    Name VARCHAR(30),
    Email VARCHAR(30),
    FavCoffee VARCHAR(30),
    FOREIGN KEY (FavCoffee) REFERENCES COFFEE(Ref) ON UPDATE CASCADE ON DELETE CASCADE
);

CREATE TABLE PROVIDER(
    Name VARCHAR(30) PRIMARY KEY,
    Email VARCHAR(30)
);

CREATE TABLE SUPPLY(
    Provider VARCHAR(30),
    Coffee VARCHAR(30),
    PRIMARY KEY (Provider, Coffee),
    FOREIGN KEY (Provider) REFERENCES PROVIDER(Name) ON UPDATE CASCADE ON DELETE CASCADE,
    FOREIGN KEY (Coffee) REFERENCES COFFEE(Ref) ON UPDATE CASCADE ON DELETE CASCADE
);

INSERT INTO COFFEE VALUES
    (001, 'Brazil', 'Light', 8.9),
    (121, 'Bolivia', 'Dark', 7.5),
    (311, 'Brazil', 'Medium', 9.0),
    (221, 'Sumatra', 'Dark', 10.25);

INSERT INTO CUSTOMER VALUES
    (001, 'Bob Hill', 'b.hill@sip.net', 221),
    (002, 'Ana Swamp', 'swampa@nca.edu', 311),
    (003, 'Mary Sea', 'brig@gsu.gov', 121),
    (004, 'Pat Mount', 'pmount@fai.fr', 121);

INSERT INTO PROVIDER VALUES
    ('Coffee Unl.', 'bob@cofunl.com'),
    ('Coffee Exp.', 'pat@coffeeex.dk'),
    ('Johns & Co.', NULL);

INSERT INTO SUPPLY VALUES
    ('Coffee Unl.', 001),
3.9 Setting Up Your Work Environment

```
('Coffee Unl.', 121),
('Coffee Exp.', 311),
('Johns & Co.', 221);

START TRANSACTION;
INSERT INTO CUSTOMER VALUES(005, 'Bob Hill', NULL, 001);
INSERT INTO COFFEE VALUES(002, 'Peru', 'Decaf', 3.00);
INSERT INTO PROVIDER VALUES(NULL, 'contact@localcof.com'); -- ERROR
  1048 (23000): Column 'Name' cannot be null
INSERT INTO SUPPLY VALUES('Johns & Co.', 121);
INSERT INTO SUPPLY VALUES('Coffee Unl.', 311, 221); -- ERROR 1136
  (21S01): Column count doesn't match value count at row 1

-- COMMIT;
-- Rest the changes:
ROLLBACK;

START TRANSACTION;
UPDATE CUSTOMER SET FavCoffee = 001 WHERE CardNo = 001; -- Rows
  matched: 1 Changed: 1 Warnings: 0
SELECT * FROM CUSTOMER;
ROLLBACK;

START TRANSACTION;
UPDATE COFFEE SET TypeOfRoast = 'Decaf' WHERE Origin = 'Brazil'; --
  Rows matched: 2 Changed: 2 Warnings: 0
SELECT * FROM COFFEE;
ROLLBACK;

START TRANSACTION;
UPDATE PROVIDER SET Name = 'Coffee Unlimited' WHERE Name = 'Coffee
  Unl.'; -- Rows matched: 1 Changed: 1 Warnings: 0
SELECT * FROM PROVIDER;
SELECT * FROM SUPPLY;
ROLLBACK;

START TRANSACTION;
UPDATE COFFEE SET PricePerPound = 10.00 WHERE PricePerPound > 10.00;
  -- Rows matched: 1 Changed: 1 Warnings: 0
SELECT * FROM COFFEE;
ROLLBACK;

START TRANSACTION;
DELETE FROM CUSTOMER WHERE Name LIKE '%S%'; -- Query OK, 2 rows
  affected (0.01 sec)
SELECT * FROM CUSTOMER;
ROLLBACK;

START TRANSACTION;
```
DELETE FROM COFFEE WHERE Ref = '001'; -- Query OK, 1 row affected (0.00 sec)
SELECT * FROM COFFEE;
SELECT * FROM SUPPLY;
ROLLBACK;

START TRANSACTION;
DELETE FROM SUPPLY WHERE Provider = 'Coffee Unl.' AND Coffee = '001';
-- Query OK, 1 row affected (0.00 sec)
SELECT* FROM SUPPLY;
ROLLBACK;

START TRANSACTION;
DELETE FROM PROVIDER WHERE Name = 'Johns & Co.'; -- Query OK, 1 row affected (0.00 sec)
SELECT * FROM PROVIDER;
SELECT * FROM SUPPLY;
ROLLBACK;

SELECT Origin FROM COFFEE WHERE TypeOfRoast = 'Dark';
SELECT FavCoffee FROM CUSTOMER WHERE Name LIKE 'Bob%';
SELECT Name FROM PROVIDER WHERE Email IS NULL;
SELECT COUNT(*) FROM SUPPLY WHERE Provider = 'Johns & Co.';
SELECT Provider FROM COFFEE, SUPPLY WHERE TypeOfRoast = 'Dark' AND Coffee = Ref;
-- SELECT name from PROVIDER WHERE Name in (SELECT Provider from
  → SUPPLY WHERE COFFEE IN (SELECT Ref from COFFEE WHERE TypeOfRoast =
  → 'Dark'));

Solution to Problem 3.8 (Improving a role-playing game) The following solves all the mentioned issues. As quests only rarely provides a special item, we added a relation to avoid having a Special-item in the QUEST table that would be too often NULL.
Solution to Problem 3.9 (A simple database for books) Pb 3.9 – Solution to Q. 1

Here are possible ways of getting the required information:

- The title of all the books:

  SELECT Title FROM BOOK;

- The (distinct) name of the publishers.

  SELECT DISTINCT Name FROM PUBLISHER;

- The title and publication date of the books published since January 31, 2012.

  SELECT Title, Published FROM BOOK
  WHERE Published > DATE '20120131';

- The first and last names of the authors published by Gallimard (no matter the city).

  SELECT FName, LName FROM AUTHOR, BOOK
  WHERE PublisherName = "Gallimard"
  AND Author = Id;

- The first and last names of the authors who were not published by an editor in New-York.

  SELECT FName, LName FROM AUTHOR, BOOK
  WHERE NOT PublisherCity = "New-York"
  AND Author = Id;

- The id of the authors who published a book whose name starts with “Where”.

  SELECT Author FROM BOOK
  WHERE Title LIKE 'Where%';

- The total number of pages in the database.

  SELECT SUM(Pages) FROM BOOK;
• The number of pages in the longest book written by the author whose last name is “Wolve”.

```
SELECT MAX(PAGES) FROM BOOK, AUTHOR
WHERE LName = "Wolve"
    AND Author = Id;
```

• The title of the books published in the XIXth century.

```
SELECT Title FROM BOOK
WHERE Published >= DATE '18010101'
    AND Published <= DATE '19001231';
```

Pb 3.9 – Solution to Q. 2 We can use the following command:

```
UPDATE BOOK SET Title = "BANNED"
WHERE Author = 3;
```

But, as the pair (title, publication date) is the primary key in the BOOK table, if the author whose id is 3 has published more than one book at a particular date, then our update will be rejected, as applying it would result in violating the entity integrity constraint.

Pb 3.9 – Solution to Q. 3 To delete the required rows, we can use:

```
DELETE FROM BOOK WHERE Author = 3;
DELETE FROM AUTHOR WHERE Id = 3;
```

Note that trying to delete the rows in the AUTHOR table before deleting the rows in the BOOK table could cause a referential integrity violation, since some of the books would be “authorless”.

Pb 3.9 – Solution to Q. 4 We could design that table as follows:

```
CREATE TABLE AWARD(
    Name VARCHAR(30),
    Year DATE,
    BookTitle VARCHAR(30),
    BookPubDate DATE,
    FOREIGN KEY (BookTitle, BookPubDate)
        REFERENCES BOOK(Title, Published),
    PRIMARY KEY (Name, Year)
);  
```

Note that there is no need to store the name of the author in that relation: this information can be recovered by looking in the BOOK table for the name of the author of the awarded book.

Pb 3.9 – Solution to Q. 5 Drawing added soon.

Pb 3.9 – Solution to Q. 6 Two of the flaws that come to mind are:

• The choice of the primary key for the BOOK relation: two books with the same title cannot be published on the same day, and that is a serious limitation. Using a primary key like ISBN would be much more appropriate.

• The impossibility to deal with books written by multiple authors or published by multiple publishers. We could address this by having two separate tables,
IS_THE_AUTHOR_OF and PUBLISHED_BY, that “maps” book’s ISBN with author’s or editor’s primary key.
4 Designing a Good Database

Resources

This part of the lecture covers significantly more material than the other, hence we give the details of the references below:

- E.R. models: (Elmasri and Navathe 2010, Ch. 7) or (Elmasri and Navathe 2015, Ch. 3)
- The E.R. to Relational model: (Elmasri and Navathe 2010, Ch. 9.1) or (Elmasri and Navathe 2015, Ch. 9.1)
- Normalization: (Elmasri and Navathe 2010, Ch. 7) or (Elmasri and Navathe 2015, Ch. 3)
- UML: not so much in the textbook, but you can look at (Elmasri and Navathe 2010, Ch. 7.8, 10.3) or (Elmasri and Navathe 2015, Ch. 3.8).

4.1 Interest for High-Level Design

Previous relational models have mistakes and limitations:

- What if a hurricane is over more than one state?
- What if an insurance covers more than one car, more than one driver?
- Changing the code “on the fly”, as we did for the Lecture and Grade tables, is difficult and error-prone.

We could go back and forth between relational models (logical level) and SQL implementations (physical level), but we will use even more high-level tools:

- Entity Relationship Models (ER, static: DB)
- Unified Modelling Diagrams (UML, dynamic: DB + software)
- Enhanced Entity Relationship Models (EER, adds operations to ER)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Conceptual</th>
<th>Logical</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity Names</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Entity Relationships</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Attributes</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Primary Keys</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Foreign Keys</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Table Names</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Column Names</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Column Data types</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Remember that in relational models, relations were representing entities (Student) and relationships (Majors_In), here the distinction is made in this table (entity vs relationship).

Remember that a model is supposed to be DBMS independant, and that computer science is at the
4.2 Entity-Relationship Model

Data is organized into entity, relationships and attributes.

4.2.1 Entities

- Entity = Thing, object, with independent existence.
- Each entity has attributes (properties)

Entity A:

- Name = Clément
- Address = HCOB, HA, E. 128 ; Invented St., Auguta, GA
- Diploma = Ph.D in CS; BS in Math

Border between humans and computers. Cf. (Elmasri and Navathe 2010, Figure 7.1) or (Elmasri and Navathe 2015, Figure 3.1) for the "parallel journey" of operations:

Introduce topics to come.
Some vocabulary:

- Entity = actual thing (individuel)
- Entity type = collection of entities with the same attributes
- Entity set (or collection) = collection of all entities of a particular entity type.

### 4.2.1.1 Attributes

Attributes can be

- Composite (divided in smaller parts) or simple (atomic)
- Single-valued or multi-valued
- Stored vs derived
- Nested!

{} = multi-valued

(...) = complex

For instance, one could store multiple address using the “schema” \{Address(Street, Number, Apt, City, State, ZIP)\}.

### 4.2.1.2 Key attributes

A key attribute is an attribute whose value is distinct for each entity in the entity set.

- Serve to identify entity
- Can be more than one such attribute
- Cannot be multiple attributes: if more than one attribute is needed to make a key attribute, combine them into a composite attribute and make it the key.
- A composite attribute that is a key attribute should not still be a key attribute if we were to remove one of the attribute (similar to the minimality requirement).
- An entity with no key is called a weak entity type: it is an entity that will be identified thanks to its relation to other entities, and thanks to its partial key (we will discuss this later).

### 4.2.1.3 Drawing entity types

- Entity = squared box (name in upper case)
- Attribute = rounded box connected to square box (name in lower case)

<table>
<thead>
<tr>
<th>If the attribute is ...</th>
<th>then...</th>
</tr>
</thead>
<tbody>
<tr>
<td>composite</td>
<td>other attributes are connected to it</td>
</tr>
<tr>
<td>multivalued</td>
<td>the box have double lines</td>
</tr>
<tr>
<td>derived</td>
<td>the box have dotted lines</td>
</tr>
<tr>
<td>a key</td>
<td>the name of the attribute is underlined</td>
</tr>
</tbody>
</table>
### 4.2.2 Relationships

Reminder: entity = actual thing, entity set = collection of entities, entity type = abstraction.

#### 4.2.2.1 Vocabulary

- Relationship = actual relation (or action) between entities ("teaches", "loves", "possesses", etc.).
- Relationship instance = $r_1$ associates $n$ entities $e_1, ..., e_n$ ("Pr. X teaches CSCI YYY", "There is love between Mary and Paul", etc.)
4.2 Entity-Relationship Model

- Relationship set = collection of instances
- Relationship type = abstraction ("Every course belong to one instructor", "Love is a relation between two persons", etc).

\[ E_1, \ldots, E_n \] participate in \( R \), \( e_1, \ldots, e_n \) participate in \( r_1 \), \( n \) is the degree.

Note that we can have Entity Set 1 = Entity Set 2, in which case we say the relation is recursive.

Naming convention:
- Use a singular name for entity types.
- Use a verb for relationship.
- Relationship types are drawn in losanges.
- Drawing usually reads right to left, and up to down.
4.2 Entity-Relationship Model

4.2.2.2 Role names and recursive relations

Convenient, and sometimes mandatory, to give role names.

If we want to stress that we are considering only one aspect of an entity (that is, a person is not only an employee, a company is not only an employer, but this aspect is crucial for the "EMPLOYS" relation):

We can also use it to make the “right-side” and the “left-side” of a recursive relationship explicit:
4.2 Entity-Relationship Model

Finally, we will sometimes use “Role Name of Entity 1 : Role Name of Entity 2” as a notation for the relation between them. For instance, we can write “Employer:Employee” to denote the “EMPLOYS” relation, and we will also use this notation with entities, and write “PERSON:POSITION” for the “OCCUPIES” relation.

4.2.2.3 Constraints

Two constraints, called "structural constraints", applies to relationship types: cardinality ratio and participation constraint. They both concerns the number of relationship instances an entity can participate in (which is different from the cardinality of a relationship type).

4.2.2.3.1 Cardinality ratio

**Maximum** number of relationships instances that an entity can participat it.

For binary relations, can be 1 : 1, N : 1, M : N (1 is “at most”, M, N, O, P, is “no maximum” → in E.R. diagram, we don’t count (yet)).

- MENTOR : MENTEE is 1 : N (“A mentor can have multiple mentees, a mentee has at most one mentor.”)
- PERSON : SSN is 1 : 1 (“A person has one SSN, a SSN belongs to one person.”)
- COURSE : DEPARTMENT is N : 1 (“A course is offered by one department, a department can offer any number of courses.”)
- STUDENT : TEAM is M : N (“A student can participate in multiple teams, a team can have multiple students.”)

We indicate the ratio on the edges:
4.2.2.3.2 Participation constraint

**Minimum** number of relationships instances that an entity can participate in, a.k.a. “minimum cardinality constraint”.

The participation can be total (a.k.a. existence dependency) or partial.

Total is drawn with a double line, partial is drawn with a single line.

![Diagram](image)

4.2.2.4 Attributes

Relationships can have attributes too! The typical example is a date attribute.

- **TEACHING** relation between PROF and CLASS (\(N : M\)) could have a “Quarter” attribute.
- **MENTORING** relation between MENTOR and MENTEE (\(1 : N\)) could have a “Since” attribute.
- **EMITED_DRIVING_LICENCE** between DMV and PERSON (\(N : 1\)) could have a “Date” attribute.

We are dealing with moving aspects here: attributes on \(1 : 1\), \(1 : N\), \(N : 1\) relationships can be migrated (to the \(N\) side when there is one).

4.2.2.5 Relationships of degree higher than two

To determine cardinality ratio: fix all but one, wonder how many can be in that relationship.
Need to find a good 3-ary example

There is another, large example on paper.

### 4.2.3 Weak entity types

Two sorts of entity types:

- Strong (a.k.a. regular, the ones we studied so far), with a key attribute,
- Weak, without key attribute.

Weak (child) entity types are identified by **identifying / owner** type that is related to it, in conjunction with one attribute (**the partial key**). Relation is called identifying relationship, and weak entities have a total participation constraint.
4.2 Entity-Relationship Model

Choice between two representation: if pet is involved in other relationships!

- Weak entities types can sometimes be replaced by complex (composite, multi-valued) attributes, unless they are involved in other relationships.
- Owner can itself be weak!
- The degree of the identifying relationship can be more than 2!

4.2.4 Alternative notation

Drawings

You can even write actual values:

\[
\text{Carpool} \ (1,5) \quad \diamond \quad (0,3) \quad \text{Person}
\]

1 \ N \quad \Rightarrow \quad (0,N) \quad (0,1)

\[N \ 1 \quad \Rightarrow \quad (1,1) \quad (1,N)\]

Crow’s foot notation:
4.2 Entity-Relationship Model

Extended (or Enhanced) E.R. Models (E.E.R.) have additionally:

- Subtype / Subclass: “every professor is an employee”. There is a class / subclass relationship (you can proceed by specialization or generalization).
- Category (to represent UNION): an OWNER entity that can be either a PERSON, a BANK, or a COMPANY entity type.

Closer to object-oriented programming.

4.2.5 Enhanced Entity–Relationship Model

4.2.6 Reverse Engineering

From relational models to E.R. models (sometimes needed)
4.3 E.R.-to-Relational Models Mapping

4.3.1 Intro

We have to map all of the following:

<table>
<thead>
<tr>
<th>Entity</th>
<th>Strong, Weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td>Composite, Key, Atomic, Multi-valued, Partial Key</td>
</tr>
<tr>
<td>Relationships</td>
<td>Binary (1 : 1, N : 1, 1 : N, N : M), n-ary</td>
</tr>
</tbody>
</table>

Using four tools: Relations, Attributes, Primary Keys, Foreign Keys.

4.3.2 Algorithm
### 4.3 E.R.-to-Relational Models Mapping

<table>
<thead>
<tr>
<th>#</th>
<th>Relation Type</th>
<th>Mapping Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Strong Entity Relation</td>
<td>Relation with all the simple attributes. Decompose complex attributes. Pick a key to be the PK, if it is composite, take its elements.</td>
</tr>
<tr>
<td>2</td>
<td>Weak Entity Relation</td>
<td>Relation with all the simple attributes. Decompose complex attributes. Add as a foreign key the primary key of the relation corresponding to the owner entity type, and make it a primary key. If the owner entity type is itself weak, start with it.</td>
</tr>
<tr>
<td>3</td>
<td>Binary 1 : 1 Relationship Types</td>
<td>Foreign Key, Merge Relations or Cross-Reference approach</td>
</tr>
<tr>
<td>4</td>
<td>Binary 1 : N Relationship Types</td>
<td>Foreign Key or Cross-Reference approach</td>
</tr>
<tr>
<td>5</td>
<td>Binary M : N Relationship Types</td>
<td>Cross-Reference approach</td>
</tr>
<tr>
<td>6</td>
<td>n-ary Relationship Types</td>
<td>Cross-Reference approach</td>
</tr>
<tr>
<td>7</td>
<td>Multivalued Attributes</td>
<td>Create a new relation, whose primary key is the foreign key to the entity.</td>
</tr>
</tbody>
</table>

1. **Foreign Key Approach**: Chose one of the relation (preferably with total participation constraint, or on the N side), add a foreign key and all the attributes of the relationship.
2. **Merged Relation Approach**: If both participations are total, just merge them. Primary key = just pick one, and add a NOT NULL constraint on the other.
3. **Cross-Reference or Relationship Relation Approach**: Create a lookup table with two (or more!) foreign keys, pick one of them (or the one on the N side, or both if $M : N$, or all if $n$-ary) as the primary key.

**Bug**: + Propagate option? **Cascade, most of the time**: weak entity type, lookup tables, etc.

---

**Diagrams**

[Diagram showing E.R. model and relational mapping]
### 4.4 Guidelines and Normal Form

What makes a good database? At the logical (conceptual) and physical (implementation) levels.

**Goals:**
1. Information preservation (and avoid loss of information)
2. Minimum redundancy
3. Make queries easy (avoid redundant work, make select / join easy)

#### 4.4.1 General rules

**4.4.1.1 Semantics**

1 relation corresponds to 1 entity or 1 relationship type

**4.4.1.2 No anomalies**

1. **Insertion Anomalies** Having to invent values or to put NULL to insert tuples, especially on a key attribute!
2. **Deletion Anomalies** Loosing information inadvertently
3. **Modification Anomalies** Updated have to be consistent.

(Bad!) Example:

```
---------- (Login, Name, AdvisoryName, AdvisorOffice, Major, MajorHead)
```

```
----------(Office, PhoneNumber, Building)
```

1. Advisor without student
2. Delete last student of advisor
3. Advisor change name.

---

<table>
<thead>
<tr>
<th>E.R. Model</th>
<th>Relational Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity type</td>
<td>Entity relation</td>
</tr>
<tr>
<td>1 : 1 or 1 : N relationship type</td>
<td>Foreign key (or relationship relation)</td>
</tr>
<tr>
<td>M : N relationship type</td>
<td>Relationship relation and two foreign keys</td>
</tr>
<tr>
<td>n-ary relationship type</td>
<td>Relationship relation and n foreign keys</td>
</tr>
<tr>
<td>Simple attribute</td>
<td>Attribute</td>
</tr>
<tr>
<td>Composite attribute</td>
<td>Set of simple component attributes</td>
</tr>
<tr>
<td>Multivalued attribute</td>
<td>Relation and foreign key</td>
</tr>
<tr>
<td>Value set</td>
<td>Domain</td>
</tr>
<tr>
<td>Key attribute</td>
<td>Primary key</td>
</tr>
</tbody>
</table>
4.4 Guidelines and Normal Form

4.4.1.3 Null should be rare

NULL has 3 meanings, wastes space, and makes join / nested projections harder.

Example:

STUDENT(Login, ..., siblingEnrolled)

Transform into "Emergency Contact in University" relation (bonus: allow multiple contacts).

4.4.1.4 Identical Attributes in Different Tables Should Be (Primary, Foreign) Key Pairs

Example with advisorOffice and Office: if we try to write a join to obtain the phone number of a student’s advisor, we will obtain all the phone. (Not clear example, find a better one).

4.4.2 Example

MARKER(Owner, Color, OwnerOffice, Brand, BrandEmail)

TEACHER(Office, Name, Phone)

Corrected to:

MARKER(Owner, Color, Brand)

TEACHER(Office, Name, Phone)

BRAND(Name, Email)

4.4.3 Functional dependencies

Functional dependencies (FD) is a formal tool used to assess how “good” a database is, a property of the relation schema. Functional dependencies list the constraints between two sets of attributes from the database. For instance, if $X$ and $Y$ are (sets of) attributes, $X \rightarrow Y$ reads “$X$ fixes $Y$”, and implies that the value(s) of $Y$ is fixed by the value(s) of $X$.

4.4.3.1 Using semantics of attributes

“What should be.”

Let us list all the attributes of our previous example:

MARKER.Owner, MARKER.Color, MARKER.Brand, TEACHER.Office, TEACHER.Name, TEACHER.Phone, BRAND.Name, BRAND.Email

Think about their dependencies, and list them:

- $\text{TEACHER.Name} \rightarrow \text{TEACHER.Office}$
- $\text{BRAND.Name} \rightarrow \text{BRAND.Email}$
- $\text{TEACHER.Office} \rightarrow \text{TEACHER.Name}$
• TEACHER.Office → TEACHER.Phone
• MAKER.Owner and MARKER.Color → MARKER.Brand

4.4.3.2 Using relation states

“What is”, can disprove some of the assumptions made previously, but shouldn’t add new dependencies based on it (they may be by chance!).

• Maybe TEACHER.Office → TEACHER.Name does not hold, because teachers share office?
• Maybe TEACHER.Name → MARKER.Brand and MARKER.Color hold?

A particular state can’t enforce a FD, but it can negate one.

Example:

<table>
<thead>
<tr>
<th>Att. 1</th>
<th>Att. 2</th>
<th>Att. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>15</td>
<td>Boston</td>
</tr>
<tr>
<td>Bob</td>
<td>13</td>
<td>Boston</td>
</tr>
<tr>
<td>Jane</td>
<td>12</td>
<td>Augusta</td>
</tr>
<tr>
<td>Emily</td>
<td>12</td>
<td>Augusta</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>May hold</th>
<th>Won’t hold</th>
</tr>
</thead>
<tbody>
<tr>
<td>Att. 2 → Att. 3</td>
<td>Att1 → Att2</td>
</tr>
<tr>
<td>Att. 3 → Att. 2</td>
<td>Att. 3 → Att. 2</td>
</tr>
<tr>
<td>Att. 1 → Att. 3</td>
<td>Att. 2 → Att. 1</td>
</tr>
<tr>
<td>{Att. 1, Att. 2} → Att. 3</td>
<td>{Att. 3, Att. 2} → Att. 1</td>
</tr>
</tbody>
</table>

4.4.3.3 Notations

Or, more conveniently:

If an attribute is a foreign key to another, we will draw an arrow between relations:

Note that:
• $X$ and $Y$ are sets, we will write $A$ instead of $\{A\}$, but keep writing $\{A, B\}$ for $\{A, B\}$.
• $\{A_1, \ldots, A_n\} \rightarrow \{B_1, \ldots, B_m\}$ means that $A_1$ and $\ldots$ and $A_n$ fix $B_1$, and that $A_i$ and $\ldots$ and $A_n$ fix $B_m$, etc.
• FD$_1$, FD$_2$, $\ldots$, FD$_n$ for the list of functional dependencies, F for all of them.
• $A \rightarrow B$ doesn’t imply nor refute $B \rightarrow A$.
• We won’t write the FD that are implied by (this variation of) Armstrong’s axioms$^1$:
  - Reflexivity: If $Y$ is a subset of $X$, then $X \rightarrow Y$
  - Augmentation: If $X \rightarrow Y$, then $\{X, Z\} \rightarrow Y$
  - Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$ We will assume that the consequence of those axioms always hold ("closure under those rules"), but will generally not write them explicitly.

### 4.4.3.4 Definitions

Remember superkey (not minimal key), key, candidate key, secondary key? We now have a formal definition.

In one particular relation $R(A_1, \ldots, A_n)$,

• If $\{A_1, \ldots, A_n\} \rightarrow Y$ for all attribute $Y$, then $\{A_1, \ldots, A_n\}$ is a superkey.
• If $\{A_1, \ldots, A_n\} A_i$ is not a superkey anymore for all $A_i$, then $\{A_1, \ldots, A_n\}$ is a key.
• We will often discard candidate key and focus on one primary key.
• If $A_i$ is a member of some candidate key of $R$, it is a **prime attribute** of $R$. It is a **non-prime attribute** otherwise.

Given a FD $\{A_1, \ldots, A_n\} \rightarrow Y$,

• It is a **full functional dependency** if for all $A_i$, $\{A_1, \ldots, A_n\} A_i \rightarrow Y$, does not hold.
• It is a **partial dependency** otherwise.

A FD $: X \rightarrow Y$ is a **transitive dependency** if there exist a set of attribute $B$ s.t.

• $B \neq X, B \neq Y$
• $B$ is not a candidate key,
• $B$ is not a subset of any candidate key,
• $X \rightarrow B$ and $B \rightarrow Y$ hold

**Examples on lecture 17’s note to incorporate?**

### 4.4.4 Normal forms and keys

First, Second, Third, Fourth, Fifth normal form ("X"NF). Stronger than the Third, there is the Boyce-Codd NF (BCNF)

If you satisfy $N$, you satisfy $N - 1$, $N - 2$, etc. The **normal form** of a relation is the highest normal form condition that it meets.

#### 4.4.4.1 First normal form

#### 4.4.4.1.1 Definition

$^1$https://en.wikipedia.org/wiki/Armstrong%27s_axioms
The domain of all attributes must be atomic (simple, indivisible): exclude multi-valued and composite attributes.

Sometimes, additional requirement that every relation has a primary key. We will take this requirement to be part of the definition of 1NF, but some authors take a relation to be in 1NF if it has at least candidate keys (i.e., multiple possible keys, but no primary key, which makes their definition more general, cf. (Elmasri and Navathe 2015, 14.4.1)). Hence, we will always assume that a primary key is given, and it will be underlined.

### 4.4.4.2 Second normal form

#### 4.4.4.2.1 Definition

1NF + Every non-prime attribute is fully functionally dependent on the primary key.

#### 4.4.4.2.2 Normalization

For each attribute $A$ of the relation whose primary key is $A_1, ..., A_n$:

- Is it prime (i.e., is $A \in \{A_1, ..., A_n\}$)?
  - Yes → Done.
  - No → Is it partially dependent on the primary key?
    * No, it is fully dependent on the primary key → Done
    * Yes, it depends only of $\{A'_1, ..., A'_k\}$ → Do the following:
      - Create a new relation with $A$ and $A'_1, ..., A'_k$, make $\{A'_1, ..., A'_k\}$ the primary key,
      - Remove $A$ from the original relation
      - Add a foreign key from $\{A'_1, ..., A'_k\}$ to their original counterparts in the original relation.

Course (Teacher Code Credit Hours Textbook Difficulty)

becomes

Course (Teacher Code Textbook Difficulty)

CreditHours (Code Credit Hours)

Refinement: note that if more than one attribute depends of the same subset $\{A'_1, ..., A'_k\}$, we will create two relations: that is useless, we could have created just one. For instance, considering
4.4 Guidelines and Normal Form

applying the algorithm would give

whereas a more subtle algorithm would give

Note that both are in Second Normal Form, though.

Note also that if our primary key is a singleton, then there is nothing to do, we are in 2NF as soon as we are in 1NF.

4.4.4.3 Third normal form

4.4.4.3.1 Definition

2NF + no non-prime attribute is transitively dependent on the primary key.

4.4.4.3.2 Normalization

For each attribute $A$ of the relation whose primary key is $A_1, \ldots, A_n$:

- Is it prime (i.e., is $A \in \{A_1, \ldots, A_n\}$)?
  - Yes $\rightarrow$ Done.
  - No $\rightarrow$ Is it transitively dependent on the primary key?
    * No, there is no $\{A'_1, \ldots, A'_k\}$ such that $\{A_1, \ldots, A_n\} \rightarrow \{A'_1, \ldots, A'_k\} \rightarrow A$ and $\{A'_1, \ldots, A'_k\} \not\subseteq \{A_1, \ldots, A_n\}$ and $A \not\in \{A'_1, \ldots, A'_k\} \rightarrow$ Done
    * Yes, there is such a $\{A'_1, \ldots, A'_m\}$ $\rightarrow$ Do the following:
      - Create a new relation with $A$ and $\{A'_1, \ldots, A'_k\}$, make $\{A'_1, \ldots, A'_k\}$ the primary key,
      - Remove $A$ from the original relation
      - Add a foreign key from $\{A'_1, \ldots, A'_k\}$ to their original counterparts in the original relation.

ADD EXAMPLES
4.4.4.4 Notes and examples

CCL: every FD $X \rightarrow Y$ s.t. $X$ is a proper subset of the primary key, or a non-prime attribute, is problematic.

4.5 Unified Modeling Language Diagrams

4.5.1 Overview

One approach for analysis, design, implementation and deployment of databases and their applications. Databases interact with multiple softwares and users, we need a common language.

Unified Modeling Language is a standard:

- Generic
- Language-independent
- Platform-independent

Wide, powerful, but also intimidating.

You know UML from object-oriented programming language:

That’s a class diagram, there are other types of diagrams, they are not unrelated! For instance, using communication diagrams, deployment diagrams, and state chart diagrams, you can collect the requirements needed to draw a class diagram! They each offer a viewpoint on a software that will help you in making sure the various pieces will fit together: it is a tool commonly used in software engineering, and useful in database design.

\[\text{http://uml.org}\]
4.5.2 Types of diagrams

There are 14 different types of diagrams, divided between two categories: structural and behavioral.

4.5.2.1 Structural UML diagrams

They describe structural, or static, relationships between objects, softwares.

- **Class diagram** describes static structures: classes, interfaces, collaborations, dependencies, generalizations, etc. We can represent conceptual data base schema with them!
- **Object diagram**, a.k.a. instance diagram, represents the static view of a system at a particular time. You can think of a “freeze” of a program, to be able to observe the value of the variables and the objects (or instances) created.
- **Component diagram** describes the organization and the dependencies among software components (e.g., executables, files, libraries, etc.), to describe how an arbitrary large software system is split into pieces.
- **Deployment diagram** is the description of the physical deployment of artifacts (i.e., software components) on nodes (i.e., hardware). If your program runs on a local computer, fetching data from the Internet, and storing output on a server, you may describe this situation using this sort of diagram.

In this category also exist **Composite structure diagram**, **Package diagram** and **Profile diagram**.

4.5.2.2 Behavioral UML diagrams

They describe the behavioral, or dynamic, relationship, between components.

- **Use case diagram** describes the interaction between the user and the system. Supposedly, it is the privileged tool to communicate with end-users.
• **State machine diagram**, a.k.a., state chart diagram, describes how a system react to external events. You can picture yourself a complex form of finite state automata diagram.

• **Activity diagram** is a flow of control between activities. You may have seen them already, they are supposedly easy to follow:

![Activity Diagram](image.png)

Then there is the sub-category of “Interaction diagrams”:

• **Sequence diagram** describes the interactions between objects over time, the flow of information or messages between objects. It is helpful to grasp the time ordering of the interactions.

• **Communication diagram**, a.k.a., collaboration diagram, describes the interactions between objects as a series of sequenced messages. It is helpful to grasp the structure of the objects, who is interacting with who.

This sub-category also comprise **Timing diagram** and **Interaction overview diagram**.

### 4.5.3 Zoom on classes diagrams

Looking at the “COMPANY conceptual schema in UML class diagram notation”, and comparing it with the “ER schema diagram for the COMPANY database” from the textbook, can help you in writing your own “Rosetta Stone” between ER and UML diagram. Let us introduce some UML terminology for the class diagrams.

<table>
<thead>
<tr>
<th>UML</th>
<th>ER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>Entity Type</td>
</tr>
<tr>
<td>Class Name</td>
<td>Entity Name</td>
</tr>
<tr>
<td>Attributes</td>
<td>Attributes</td>
</tr>
<tr>
<td>Operations (or Method)</td>
<td><em>Sometimes</em> Derived Attributes</td>
</tr>
<tr>
<td>Association</td>
<td>Relationship Type</td>
</tr>
<tr>
<td>Link</td>
<td>Relationship Instance</td>
</tr>
<tr>
<td>Multiplicities</td>
<td>Structural Constraint</td>
</tr>
</tbody>
</table>
As well as for ER diagram, the domain (or data type) of the attributes is optional. A composite attribute in an ER diagram can be interpreted as a structured domain in a UML diagram (think of a struct), and a multi-valued attribute requires to create a new class.

Associations are, to some extend, more expressive than relationship types:

- **As for relationship types**, they can be recursive (or reflexive), and uses role names to clarify the roles of both parties.
- **As for relationship types** they can have attributes: actually, a whole class can be connected to an association.
- **As for relationship types**, they can express a cardinality constraint on the relation between classes. They are written as \( \min \ldots \max \), with \(*\) for “no maximum”, and the following shorthands: \(*\) stands for \(0\ldots*\) and \(1\) stands for \(1\ldots1\). An association with \(1\) on one side and \(*\) on the other (resp. \(1\) and \(1, *\) and \(*\)) is sometimes called “one-to-many” (resp., “one-to-one”, “many-to-one”, “many-to-many”). The notation in partially inverted w.r.t. ER diagrams:

- **As opposed to the relationship types**, they can have a direction, indicating that the user should be able to navigate them only in one direction, or in two (which is the default). This is used for security or privacy purposes.
- **As opposed to the relationship types**, they come in various flavors:
  - You can express **aggregation**, a.k.a. “is part of” relationship, between a whole object and its component (that have their own existence).
  - You can express **composition**, which is the particular case of aggregation where the component doesn’t have an existence of their own.
  - You can express **generalization**, a.k.a. inheritance, that eliminates redundancy and makes a class a **specialization** of another one.
- **As opposed to the relationship types**, they can be **qualified**, implying that a class is not connected to the other class as a whole, but to one particular attribute, called the **qualifier**, or **discriminator**.

This last feature can be used for weak entities, but not only.
Some of those subtleties depend on your need, and are subjective, but are important tool to design properly a database, and relieving the programmer from the burden of figuring out many details.

### 4.5.4 Missing?

On Generalization

and

Example of Use Case

**Example:** cf. review session.

### Exercises

**Exercise 4.1** What could be the decomposition of an attribute used to store an email address? When could that be useful?

**Exercise 4.2** Draw the ER diagram for a “COMPUTER” entity that has one multivalued attribute “Operating_System”, a composite attribute “Devices” (decomposed into “Keyboard” and “Mouse”) and an “Id” key attribute.

**Exercise 4.3** Draw the ER diagram for a “CELLPHONE” entity that has a composite attribute “Plan” (decomposed into “Carrier” and “Price”), a “Mobile_identification_number” key attribute, and a multi-valued “App_Installed” attribute.

**Exercise 4.4** Name one difference between a primary key in the relational model, and a key attribute in the ER model.

**Exercise 4.5** What is the difference between an entity type and a weak entity type?
Exercise 4.6 What is the degree of a relationship type?

Exercise 4.7 What is a self-referencing, or recursive, relationship type? Give two examples.

Exercise 4.8 What does it mean for a binary relationship type “Owner” between entity types “Person” and “Computer” to have a cardinality ratio $M : N$?

Exercise 4.9 What are the two possible structural constraints on a relationship type?

Exercise 4.10 Under what condition(s) can an attribute of a binary relationship type be migrated to become an attribute of one of the participating entity type?

Exercise 4.11 For the following binary relationships, suggest cardinality ratios based on the common-sense meaning of the entity types.

<table>
<thead>
<tr>
<th>Entity 1</th>
<th>Cardinality Ratio</th>
<th>Entity 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUDENT</td>
<td>\</td>
<td>MAJOR</td>
</tr>
<tr>
<td>CAR</td>
<td>\</td>
<td>TAG</td>
</tr>
<tr>
<td>INSTRUCTOR</td>
<td>\</td>
<td>LECTURE</td>
</tr>
<tr>
<td>INSTRUCTOR</td>
<td>\</td>
<td>OFFICE</td>
</tr>
<tr>
<td>COMPUTER</td>
<td>\</td>
<td>OPERATING_SYSTEM</td>
</tr>
</tbody>
</table>

Exercise 4.12 Give an example of a binary relationship type of cardinality $1 : N$.

Exercise 4.13 Give an example of a binary relationship type of cardinality $N : 1$, and draw the corresponding diagram (you don’t have to include details on the participating entity types).

Exercise 4.14 Draw an ER diagram with a single entity type, with two stored attributes and one derived attribute. In your answer, it should be clear that the value for the derived attribute will always be obtained from the value(s) for the other attribute(s).

Exercise 4.15 Draw an ER diagram expressing the total participation of an entity type “BURGER” in a binary relation “CONTAINS” between “BURGER” and “INGREDIENT”. What would be the ratio of such a relation?

Exercise 4.16 Convert the following ER diagram into a relational model:
Exercise 4.17  What is a partial key?

Exercise 4.18  Why do weak entity type have a total participation constraint?

Exercise 4.19  What is insertion anomaly? Give an example.

Exercise 4.20  What is deletion anomaly? Is it a desirable feature?

Exercise 4.21  Why should we avoid attributes whose value will often be NULL? Can the usage of NULL be completely avoided?

Exercise 4.22  Consider the following relation:

\[ \text{PROF}(\text{SSN}, \text{Name}, \text{Department}, \text{Bike\_brand}) \]

Why is it a poor design to have a “Bike\_brand” attribute in such a relation? How should we store this information?

Exercise 4.23  Consider the following relation:

\[ \text{STUDENT}(\text{SSN}, \text{Name}, \ldots, \text{Sibling\_On\_Campus}) \]

Why is it a poor design to have a “Sibling\_On\_Campus” attribute in such a relation? How should we store this information?

Exercise 4.24  Consider the following relational database schema:

\[ \text{STUDENT}(\text{Logi}, \text{Name}, \ldots, \text{Major, Major\_Head}) \]
\[ \text{DEPARTMENT}(\text{Code}, \text{Name}, \text{Major\_Head}) \]

Assuming that “Major” is a foreign key referencing “DEPARTMENT.Code”, what is the problem with that schema? How could you address it?

Exercise 4.25  Consider the relation \( R(A, B, C, D, E, F) \) and the following functional dependencies:
4.5 Unified Modeling Language Diagrams

1. \( F \to \{D, C\}, D \to \{B, E\}, \{B, E\} \to A \)
2. \( \{A, B\} \to \{C, D\}, \{B, E\} \to F \)
3. \( A \to \{C, D\}, E \to F, D \to B \)

For each set of functional dependency, give a key for \( R \). We want a key, so it has to be minimal.

**Exercise 4.26** Consider the relation \( R(A, B, C, D, E, F) \) and the following functional dependencies: \[ A \to [D, E], D \to [B, F], [B, E] \to A, [A, C] \to [B, D, F], A \to F \] Answer the following:

1. How many candidate keys is there? List them.
2. How many transitive dependencies can you find? Give them and justify them.

**Exercise 4.27** Consider the relation \( R(A, B, C, D) \) and answer the following:

1. If \( \{A, B\} \) is the only key, is \( \{A, B\} \to \{C, D\}, \{B, C\} \to D \) a 2NF? List the nonprime attributes and justify.
2. If \( \{A, B, C\} \) is the only key, is \( A \to \{B, D\}, \{A, B, C\} \to D \) a 2NF? List the nonprime attributes and justify.

**Exercise 4.28** Consider the relation \( R(A, B, C, D, E, F) \) with candidate keys \( \{A, B\} \) and \( C \). Answer the following:

1. What are the prime attributes in \( R \)?
2. Is \( \{C, D\} \to E \) a fully functional dependency?
3. Write a set of functional dependencies containing at least one transitive dependency, and justify your answer.

**Exercise 4.29** Consider the relation \( R(A, B, C, D, E) \) and the following functional dependencies:

1. \( C \to D, \{C, B\} \to A, A \to \{B, C, D\}, B \to E \)
2. \( A \to \{C, D\}, C \to B, D \to E, \{E, C\} \to A \)
3. \( \{A, B\} \to D, D \to \{B, C\}, E \to C \)

For each one, give one candidate key for \( R \).

**Exercise 4.30** Consider the relation \( R(A, B, C, D, E) \) and answer the following:

1. If \( \{A, B\} \) is the primary key, is \( B \to E, C \to D \) a 2NF?% List the nonprime attributes and justify.
2. If \( \{A\} \) is the primary key, is \( B \to C, B \to D \) a 2NF?% List the nonprime attributes and justify.

**Exercise 4.31** Consider the relation \( R(A, B, C, D, E, F) \), and let \( \{B, D\} \) be the primary key, and have additionnaly the functional dependencies \( \{A, D\} \to E, C \to F \). This relation is not in 3NF, can you tell why?

**Exercise 4.32** Consider the relation \( R(A, B, C, D) \) and answer the following:

1. If \( A \) is the only key, is \( A \to \{B, C, D\}, \{A, B\} \to C, \{B, C\} \to D \) a 3NF? List the nonprime attributes and justify.
2. If \( B \) is the only key, is \( B \to \{A, C, D\}, A \to \{C, D\}, \{A, C\} \to D \) a 3NF? List the nonprime attributes and justify.

**Exercise 4.33** Consider the relation \( R(A, B, C, D, E) \) and the functional dependencies \$ \{A, B\} \to C, B \to D, C \to E \$. Answer the following:
1. $A$ by itself is not a primary key, but what is the only key that contains $A$?
2. List the non-prime attributes.
3. This relation is not in 2NF: what transformation can you operate to obtain a 2NF?
4. One of the relation you obtained at the previous step is likely not to be in 3NF. Can you normalize it? If yes, how?

**Exercise 4.34** What are the two different categories of U.M.L. diagram?

**Exercise 4.35** Can a C++ developer working on Linux and a Java developer working on MacOS use the same class diagram as a basis to write their programs? Justify your answer.

**Exercise 4.36** What kind of diagram should we use if we want to ...

1. describe the functional behavior of the system as seen by the user?
2. capture the flow of messages in a software?
3. represent the workflow of actions of an user?

**Exercise 4.37** Name two reasons why one would want to use a U.M.L. class diagram over an E.R. diagram to represent a conceptual schema.

**Exercise 4.38** Consider the following diagram:

Give the number of attributes for both classes, and suggest two operations for the class that doesn’t have any. Discuss the multiplicities: why did the designer picked those values?

**Exercise 4.39** Briefly explain the difference between an aggregation and a composition association.

**Exercise 4.40** How is generalization (or inheritance) represented in a U.M.L. class diagram? Why is such a concept useful?

**Exercise 4.41** Convert the following E.R. diagram into a U.M.L. class diagram:
Solution to Exercises

Solution 4.1 Name / extension. To have statistics about the extensions, to sort the username by length, etc.

Solution 4.2
Solution 4.3

Solution 4.4 There can be more than one key in the ER model.

Solution 4.5 The weak entity type doesn’t have a primary key.

Solution 4.6 The number of participating entity types.

Solution 4.7 A relationship type where the same entity type participates more than once. On seats, “is to the left of”, on persons, “is married to”.

Solution 4.8 That a person can own multiple computers, and that a computer can have multiple owners.

Solution 4.9 Cardinality ration and participation constraints.

Solution 4.10 When the cardinality is 1 : N, 1 : 1 or N : 1.

Solution 4.11

<table>
<thead>
<tr>
<th>Entity 1</th>
<th>Cardinality Ratio</th>
<th>Entity 2</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUDENT</td>
<td>N : 1</td>
<td>MAJOR</td>
<td>“A student has one major, but multiple students can have the same major”</td>
</tr>
<tr>
<td>CAR</td>
<td>1 : 1</td>
<td>TAG</td>
<td>“A car has exactly one tag, a tag belongs to one particular car.”</td>
</tr>
<tr>
<td>INSTRUCTOR</td>
<td>1 : N</td>
<td>LECTURE</td>
<td>“An instructor can teach multiple lecture, but a lecture is taught by only one person.”</td>
</tr>
<tr>
<td>INSTRUCTOR</td>
<td>1 : N</td>
<td>OFFICE</td>
<td>“An instructor can have multiple office, but an office belongs to only one instructor”</td>
</tr>
<tr>
<td>COMPUTER</td>
<td>M : N</td>
<td>OS</td>
<td>“A computer can have multiple operating system, the same operating system can be installed on more than one computer.”</td>
</tr>
</tbody>
</table>

Some of those choices, of course, are arguable (typically, almost any combination seems reasonable for the INSTRUCTOR : OFFICE relation).

Solution 4.12 SUPERVISION, a recursive relationship on EMPLOYEE.
Solution 4.13

Solution 4.14

Solution 4.15

Solution 4.16 To be written.

Solution 4.17 For a weak entity attribute, it is the attribute that can uniquely identify weak entities that are related to the same owner entity.

Solution 4.18 Otherwise, we couldn’t identify entities in it without owner entity.

Solution 4.19 When you have to invent a primary key or add a lot of NULL value to be able to add a tuple. I want to add a room in my DB, but the only place where rooms are listed are as an attribute on a Instructor table, so I have to "fake" an instructor to add a room.

Solution 4.20 A delete anomaly exists when certain attributes are lost because of the deletion of other attributes. It is not desirable, since it can lead to the loss of information.

Solution 4.21 Because they waste space, and because they are ambiguous (N/A, or unknown, or not communicated?). No, it is necessary sometimes.

Solution 4.22 Because it will be NULL most of the time. In a separate table.

Solution 4.23 Because it will be NULL most of the time, and because students could have more than one sibling on campus. In a separate table.

Solution 4.24 Major_Head will give update anomalies. By putting the Head of the department in the DEPARTMENT relation only, i.e., removing it from STUDENT.

Solution 4.25
1. $F$
2. $\{A, B, E\}$
3. \{A, E\}

**Solution 4.26**
1. \{A, C\},
2. \(A \rightarrow F\) by \(A \rightarrow D, D \rightarrow F\).

**Solution 4.27**
1. Yes. \(C\) and \(D\) are non prime, and they fully depend on \{\(A, B\)\}.
2. No. \(D\) is the only non prime, and it depends only on \(A\).

**Solution 4.28**
1. \(A, B, C\)
2. No, because we can remove \(D\),
3. \(A \rightarrow D, D \rightarrow E\) and \(A \rightarrow E\).

**Solution 4.29**
1. \{\(B, C\), \(A\)\},
2. \(A, \{C, E\}\),
3. \{\(A, D, E\), \{\(A, B, E\)\}\}.

**Solution 4.30**
1. No. \(C, D, E\), and \(E\) has a partial relation to \(B\)
2. Yes. Since the primary key is a singleton, it is obvious.

**Solution 4.31**
\{\(B, D\)\} \(\rightarrow C \rightarrow F\) breaks the 3NF.

**Solution 4.32**
1. No. \(B, C\) and \(D\) are non prime, \(A \rightarrow \{B, C\} \rightarrow D\) breaks the 3NF.
2. No. \(A, B\) and \(D\) are non prime, \(B \rightarrow \{A, C\} \rightarrow D\) breaks the 3NF.

**Solution 4.33**
1. \{\(A, B\)\},
2. \(C, D, E\),
3. \(R_1(A, B, C, E)\) and \(R_2(B, D)\)
4. \(R_1(A, B, C), R_2(C, E)\) and \(R_3(B, D)\).

**Solution 4.34** Behaviour and structure

**Solution 4.35** Yes, U.M.L. diagram is language-independent and platform-independent.

**Solution 4.36**
1. Use-case
2. Sequence diagram
3. Activity diagram

**Solution 4.37** To use direction for association, to have a common language with someone less knowledgeable of other diagrammatic notations. For the concept of integration.

**Solution 4.38** Flight has 5 attributes, Plane has 4. The Plane class could have the operations
\begin{verbatim}
getLastFlightNumber() : Integer and
getMaximumSpeed(MPH) : void.
\end{verbatim}

For the multiplicities: A flight could not have a plane assigned, and a plane could not be assigned to a flight. A plane can be assigned to multiple (or no) flights, but a flight must have at most one plane (and could have none).

**Solution 4.39** Aggregation: associated class can have an existence of its own.

Composition association: class doesn’t exist without the association.

**Solution 4.40** Because it avoids redundancy.
Problems

Problem 4.1 (Design for your professor) Your professor designed the following relational model at some point in his career, to help him organizing his exams and the students grades:

<table>
<thead>
<tr>
<th>Table Name and Attributes</th>
<th>Example of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAM(Number, Date, Course)</td>
<td>&lt; 1, '2018-02-14', 'CSCI3410'&gt;</td>
</tr>
<tr>
<td>PROBLEM(Statement, Points, Length, Exam)</td>
<td>&lt; 'Your professor designed...', 10, '00:10:00', 1&gt;</td>
</tr>
<tr>
<td>STUDENT_GRADE(Login, Exam, Grade)</td>
<td>&lt; 'aalyx', 1, 83&gt;</td>
</tr>
</tbody>
</table>

where EXAM.Number, PROBLEM.Statement, STUDENT_GRADE.Login and STUDENT_GRADE.Exam are primary key, and where STUDENT_GRADE.Exam and PROBLEM.Exam both refer to EXAM.Number.

The idea was to have

- The EXAM table storing information about exams,
- One entry per problem in the PROBLEM table, and to associate every problem to an exam,
- The grade of one student for one particular exam stored in the STUDENT_GRADE table.

Unfortunately, this design turned out to be terrible. Describe at least one common and interesting situation where this model would fail to fulfill its purpose, and propose a way to correct the particular problem you identified.

Problem 4.2 (Reading the MOVIES database ER schema) Consider the ER schema for the MOVIES database ((Elmasri and Navathe 2010, Figure 7.24)): 
Assume that MOVIES is a populated database. ACTOR is used as a gender-neutral term. Given the constraints shown in the ER schema, respond to the following statements with True or False. Justify each answer.

1. There are no actors in this database that have been in no movies.
2. There might be actors who have acted in more than ten movies.
3. Some actors could have done a lead role in multiple movies.
4. A movie can have only a maximum of two lead actors.
5. Every director have to have been an actor in some movie.
6. No producer has ever been an actor.
7. A producer cannot be an actor in some other movie.
8. There could be movies with more than a dozen actors.
9. Producers can be directors as well.
10. A movie can have one director and one producer.
11. A movie can have one director and several producers.
12. There could be some actors who have done a lead role, directed a movie, and produced some movie.
13. It is impossible for a director to play in the movie (s)he directed.

**Problem 4.3 (ER diagram for car insurance)** Draw the ER diagram for the following situation: A car-insurance company wants to have a database of accidents. An accident involves cars, drivers, and it has several aspects: the moment and place where it took place, the amount of damages, and a (unique) report number. A car has a license, a model, a year, and an owner. A driver has an id, an age, a name, and an address.

One of the interesting choice is: should “accident” be an entity type or a relationship type?
Problem 4.4 *(ER-to-Relation mapping for car insurance)*

Apply the ER-to-Relation mapping to your ER diagram from the previous problem.

Problem 4.5 *(ER-to-Relation mapping for Country)* Consider the following E.R. schema:

![Diagram](image)

where

- "W_IN" stands for "WRITTEN_IN", and
- "B_W_F" stands for "BORROWS_WORDS_FROM".

For this relationship, on the left-hand side is the language that borrows a word, and on the right-hand side is the language that provides the loanword.

Map that E.R. diagram to a relational database schema.

Problem 4.6 *(Normal form of a CAR_SALE relation)* Consider the following relation, and its functional dependencies:

```
CAR_SALE(Car_no, Date_sold, Salesman_no, Commission, Discount_amt)
```

Functional Dependencies:

```
{Car_no, Salesman_no} → {Date_sold, Commission, Discount_amt}
Date_sold → Discount_amt
Salesman_no → Commission
```

and let \{Car_no, Salesman_no\} be the primary key of this relation.

Based on the given primary key, is this relation in 1NF, 2NF, or 3NF? Why or why not? Normalize it completely.
Problem 4.7 (Normalizing the FLIGHT relation) Consider the following relation:

\[
\text{FLIGHT}(\text{From}, \text{To}, \text{Airline}, \text{Flight4.7}, \text{DateHour}, \text{HeadQuarter}, \text{Pilot}, \text{TZDifference})
\]

A tuple in the FLIGHT relation contains information about an airplane flight: the airports of departure and arrival, the airline carrier, the number of the flight, its time of departure, the headquarter of the company chartering the flight, the name of the pilot(s), and the time zone difference between the departure and arrival airports.

The “Pilot” attribute is multi-valued (so that between 1 and 4 pilot’s names can be stored in it). Given an airline and a flight number, one can determine the departure and arrival airports, as well as the date and hour and the pilot(s). Given the airline carrier, one can determine the headquarter. Finally, given the departure and arrival airports, one can determine their time zone difference.

Normalize the “FLIGHT” relation to its third normal form. You can indicate your steps, justify your reasoning, and indicate the foreign keys if you want to, but don’t have to.

Problem 4.8 (From business statement to dependencies) This problem asks you to convert business statements into dependencies. Consider the following relation:

\[
\text{BIKE}(\text{Serial\_no}, \text{Manufacturer}, \text{Model}, \text{Batch}, \text{Wheel\_size}, \text{Retailer})
\]

Each tuple in the relation BIKE contains information about a bike with a serial number, made by a manufacturer, with a particular model number, released in a certain batch, which has a certain wheel size, and is sold by a certain retailer.

- Write each of the following dependencies as a functional dependency (the first one is given as an example):
  1. A retailer can’t have two bikes of the same model from different batches. solution: \(\{\text{Retailer, Model}\} \rightarrow \text{Batch}\)
  2. The manufacturer and serial number uniquely identifies the bike and where it is sold.
  3. A model number is registered by a manufacturer and therefore can’t be used by another manufacturer.
  4. All bikes in a particular batch are of the same model.
  5. All bikes of a certain model have the same wheel size.

- Based on those statements, what could be a key for this relation?

- Assuming all those functional dependencies hold, and taking the primary key you identified at the previous step, what is the degree of normality of this relation? Justify your answer.

Problem 4.9 (Normalization) Consider the relations \(R\) and \(T\) below, and their functional dependencies (on top of the one induced by the primary keys):

\[
\begin{align*}
\text{R}(\text{EventId}, \text{Email}, \text{Time}, \text{Date}, \text{Location}, \text{Status}) \\
\text{T}(\text{Invno}, \text{Subtotal}, \text{Tax}, \text{Total}, \text{Email}, \text{Lname}, \text{Fname}, \text{Phone})
\end{align*}
\]
{EventId, Email} → Status
EventId → {Time, Date, Location}
Invno → {Subtotal, Tax, Total, Email}
Email → {Fname, Lname, Phone}

Normalize the relations to 2NF and 3NF. Show all relations at each stage (2NF and 3NF) of the normalization process.

Problem 4.10 (Normal form of the BOOK relation) Consider the following relation for published books:

BOOK(Book_title, Book_type, Author_name, List_price, Author_affil, Publisher)

Suppose we have the following dependencies:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book_title</td>
<td>{Publisher, Book_type}</td>
</tr>
<tr>
<td>Book_type</td>
<td>List_price</td>
</tr>
<tr>
<td>Author_name</td>
<td>Author_affil</td>
</tr>
</tbody>
</table>

What would be a suitable key for this relation?
How could this relation not be in first normal form? Explain your answer.
This relation is not in second normal form: explain why and normalize it.
Is the relations you obtained at the previous step in third normal form? Explain why, and normalize them if needed.

Problem 4.11 (From business statements to E.R. diagram – UNIVERSITY) Consider the following requirements for a UNIVERSITY database, used to keep track of students’ transcripts.

1. The university keeps track of each student’s name, student number, class (freshman, sophomore, ..., graduate), major department, minor department (if any), and degree program (B.A., B.S., ..., Ph.D.). Student number has unique values for each student.
2. Each department is described by a name and has a (unique) department code.
3. Each course has a course name, a course number, credit hours, and is offered by at least one department. The value of course number is unique for each course. A course has at least one section.
4. Each section of a course has an instructor, a semester, a year, and a section number. The section number distinguishes different sections of the same course that are taught during the same semester/year; its values are 1, 2, 3, ..., up to the number of sections taught during each semester. Students can enroll in sections and receive a letter grade, and grade point (0, 1, 2, 3, 4 for F, D, C, B, A, respectively).

Draw an E.R. diagram for that schema. Specify key attributes of each entity type and structural constraints on each relationship type. Note any unspecified requirements, and make appropriate assumptions to make the specification complete.

Problem 4.12 (From UML to relational model – DRIVER) Consider the UML diagram
below, and convert it to the relational model. Don’t forget to indicate primary and foreign keys.

**DRIVER**

- id : String
- dob : Date
- name : String
- address : Street
- City
- getAge() : Int

**COMMERCIAL DRIVER**

- Class : String

**CAR**

- vin : String
- make : String
- year : Year
- brand : String

**POSSESSION HISTORY**

- date of purchase : Date

**CAR INSURANCE**

- policy number : String
- covered amount : int
- company name : String

**IS COVERED BY**

1.1

---

**Problem 4.13 (Normal form of the CONTACT relation)**

Consider the relation

CONTACT(Phone, Call_center, Email, Zip, Brand, Website)

and the following functional dependencies:

\[
\begin{align*}
\{\text{Zip, Brand}\} & \rightarrow \{\text{Phone}\} \\
\{\text{Brand}\} & \rightarrow \{\text{Email}\} \\
\{\text{Brand}\} & \rightarrow \{\text{Website}\} \\
\{\text{Phone}\} & \rightarrow \{\text{Call_center}\}
\end{align*}
\]

Assume that \{Zip, Brand\} is the primary key. Normalize this relation to the second normal form, and then to the third normal form. Give the relations, their primary keys, and functional dependencies for both steps.

---

**Problem 4.14 (CONSULTATION relation: justification, primary key and normal form)**

Consider the relation

CONSULTATION(Doctor_no, Patient_no, Date, Diagnosis, Treatment, Charge, Insurance)

with the following functional dependencies:

\[
\begin{align*}
\{\text{Doctor_no, Patient_no, Date}\} & \rightarrow \{\text{Diagnosis}\} \\
\{\text{Doctor_no, Patient_no, Date}\} & \rightarrow \{\text{Treatment}\} \\
\{\text{Treatment, Insurance}\} & \rightarrow \{\text{Charge}\} \\
\{\text{Patient_no}\} & \rightarrow \{\text{Insurance}\}
\end{align*}
\]

1. The designer decided not to add the functional dependency \{Diagnosis\} \rightarrow \{Treatment\}. Explain what could be the designer’s justification, at the level of the mini-world.
2. Identify a primary key for this relation.
3. What is the degree of normalization of this relation? Normalize it to the third normal form.
if necessary.

**Problem 4.15 (From E.R. to relational schema and UML class diagram – CAR_INFO)**

Consider the following E.R. schema for the CAR_INFO database:

Note that a car can have at most one driver, $N$ passengers, $N$ insurances, and that car insurances exist only if they are “tied up” to a car (i.e., they are weak entities, and their identifying relationship is called “Insured”).

1. Find the key attribute for “Car”, and the partial key for “Car Insurance”. If you can’t think of any, add a dummy attribute and make it be the key.
2. Convert that E.R. diagram to a relational database schema.
3. Convert the E.R. diagram to a U.M.L. class diagram. Comparing Figure 7.16 with Figure 7.2 from your textbook should guide you.

**Problem 4.16 (Discovering MySQL Workbench)** In this exercise, we will install and explore the basic functionalities of MySQL Workbench, which is a cross-platform, open-source, and free graphical interface for database design.

1. Install MySQL Workbench: use your package manager, or download the binaries from https://dev.mysql.com/downloads/workbench/.
2. Once installed, execute the software. Under the panel “MySQL Connections”, you should see your local installation listed as “Local instance 3306”. Click on the top-right corner of that box, and then on “Edit Connections”. Alternatively, click on ”Database”, on ”Manage Connections”, and then on ”Local instance 3306”.
3. Check that all the parameters are correct. Normally, you only have to change the name of the user to “testuser”, and leave the rest as it is. Click on ”Test the con-
4.5 Unified Modeling Language Diagrams

4. Now, click on the box “Local instance 3306”, and enter your password. A new tab appears, you can see the list of schemas in the bottom part of the left panel.

5. Click on “Database”, and then on “Reverse Engineering” (or hit \texttt{ctrl} + \texttt{r}), click on “next”, enter your password, and click on “next”. You should see the list of the schemas stored in your database. Select one (any one, we are just exploring the functionalities at that point), click on “next”, and then click on “execute”, “next”, and “close”.

6. You’re back on the previous view, but you should now see “E.E.R. diagram” on the top of the middle panel. Click on “E.E.R. diagram” twice, scroll down if needed, and you should see the E.E.R. diagram.

7. This diagram isn’t exactly an E.R. diagram, and it is not a U.M.L. diagram either. Yet, you should still be able to understand parts of it, and should try to modify it. Make some relations mandatory, change their name, add an attribute, change the name of another, insert a couple of elements in an entity, add a row in a table, etc. Make sure you understand the meaning of the lines between the entities.

8. Once you’re done, try to “Forward Engineer” by hitting “Ctrl” + “G”. Click on “next” twice, enter your password, click on lick on “next” once more, and you should see the SQL code needed to produce the table you just designed using the graphical tool.

---

**Problem 4.17 (Using MySQL Workbench’s reverse engineering)** This problem requires you to have successfully completed Pb 4.16 and Pb 4.12.

Using the relational database schema you obtained in Pb 4.12, write the SQL implementation of that database. Then, using MySQL Workbench, use the “Reverse Engineering” function to obtain a E.E.R. diagram of your database, and compare it with the U.M.L. diagram from Pb 4.12. Apart from the difference inherent to the nature of the diagram (i.e., U.M.L. Vs E.E.R.), how are they the same? How do they differ? Is the automated tool as efficient and accurate as you are?

---

**Problem 4.18 (From business statements to dependencies – KEYBOARD)** This exercise asks you to convert business statements into dependencies. Consider the following relation:

\[
\text{KEYBOARD(Manufacturer, Model, Layout, Retail\_Store, Price)}
\]

A tuple in the KEYBOARD relation contains information about a computer keyboard: its manufacturer, its model, its layout (AZERTY, QWERTY, etc.), the place where it is sold, and its price.

1. Write each of the following business statement as a functional dependency:
   a) A model has a fixed layout.
   b) A retail store can’t have two different models produced by the same manufacturer.

2. Based on those statements, what could be a key for this relation?

3. Assuming all those functional dependencies hold, and taking the primary key you identified at the previous step, what is the degree of normality of this relation? Justify
Problem 4.19  (Reverse engineering by hand)  Look at the following relational model, and "reverse-engineer" it to obtain an E.R. diagram:

Solution to Selected Problems

Solution to Problem 4.2 (Reading the MOVIES database ER schema)  
1. true  
2. true  
3. true  
4. true  
5. false  
6. false  
7. false  
8. true  
9. true  
10. true  
11. true  
12. true  
13. false
Solution to Problem 4.3 (ER diagram for car insurance)

OR

Solution to Problem 4.6 (Normal form of a CAR_SALE relation) The CAR_SALE relation is in 1st normal form, since it has a primary key, and by assuming that all the attributes are atomic. This relation is not in 2nd Normal Form: since Date_sold → Discount_amount and Salesman_no → Commission, then some attributes (namely Discount_amount and Commission) are not fully functional dependent on the primary key. Hence, this relation can’t be in 3rd normal form either.

To normalize,

2NF:
Car_Sale1(Car_no, Date_sold, Discount_amt)
Car_Sale2(Car_no, Salesman_no)
Car_Sale3(Salesman_no, Commission)
3NF:
Car_Sale1-1(Car_no, Date_sold)
Car_Sale1-2(Date_sold, Discount_amt)
Car_Sale2(Car_no, Salesman_no)
Car_Sale3(Salesman_no, Commission)

bogue: indicate primary key.

Solution to Problem 4.8 (From business statement to dependencies)
1. \{ Manufacturer, Serial_no \} → \{ Model, Batch, Wheel_size, Retailer \}
2. Model → Manufacturer
3. Batch → Model
4. \{ Model, Manufacturer \} → Wheel_size

- \{ Manufacturer, Serial_no \}
- If every attribute is atomic, it is in second nf. \{ Manufacturer, Serial_no \} → Batch → Model breaks the 3NF.

Solution to Problem 4.14 (CONSULTATION relation: justification, primary key and normal form)
Because there are no partial dependencies, the given relation is in 2NF already. This however is not 3NF because the Charge is a nonkey attribute that is determined by another nonkey attribute, Treatment. We must decompose further:

R (Doctor_no, Patient_no, Date, Diagnosis, Treatment)
R1 (Treatment, Charge)

We could further infer that the treatment for a given diagnosis is functionally dependant, but we should be sure to allow the doctor to have some flexibility when prescribing cures.

Solution to Problem 4.10 (Normal form of the BOOK relation)
- \{ Book Title, Author Name \}
  - If an attribute is composite or multi-valued.
  - Because of \{ Book_title \} → \{ Publisher, Book_type \}. We can normalize it as (Book Title, Publisher, Book Type, List Price), (Author Name, Author Affiliation), (Author Name, Book Title).
  - Because of \{ Book_title \} → \{ Book_type \} → \{ List_price \}

Solution to Problem 4.15 (From E.R. to relational schema and UML class diagram – CAR_INFO)
For "Car", we need to create an attribute, like “vin”. For "Car Insurance", “Policy Number” is perfect.
Note that, during the conversion, we had to make “Insured Car” part of the primary key of `CAR INSURANCE`.

Solution to Problem 4.17 (Using MySQL Workbench’s reverse engineering) We give first the code, then the drawing.

```sql
/* code/sql/PERSON.sql */
/* code/sql/PERSON.sql */
-- DROP SCHEMA HW_Queries;
CREATE SCHEMA HW_6;
USE HW_6;
CREATE TABLE PERSON(
    id VARCHAR(25) PRIMARY KEY,
```
```
4.5 Unified Modeling Language Diagrams

CREATE TABLE CAR(
    vin VARCHAR(25) PRIMARY KEY,
    make VARCHAR(25),
    model VARCHAR(25),
    year DATE,
    driver VARCHAR(25),
    FOREIGN KEY (driver) REFERENCES PERSON(id)
    ON UPDATE CASCADE
);

ALTER TABLE person ADD FOREIGN KEY (seat) REFERENCES car(vin);

CREATE TABLE CAR_INSURANCE(
    policy_number VARCHAR(25) PRIMARY KEY,
    company_name VARCHAR(25),
    insured_car VARCHAR(25),
    FOREIGN KEY (insured_car) REFERENCES CAR(vin)
);

CREATE TABLE PHONE(
    id VARCHAR(25),
    number VARCHAR(25),
    FOREIGN KEY (id) REFERENCES PERSON(id),
    PRIMARY KEY (id,number)
);
```
5 Databases Applications

Resources

- http://spots.augusta.edu/caubert/teaching/general/java/
- If you experience troubles, https://www.ntu.edu.sg/home/ehchua/programming/howto/ErrorMessages.html#JDBCErrors might be a good read.
- (Elmasri and Navathe 2010, 13.3.2) or (Elmasri and Navathe 2015, Chapter 10) is a condensed, but good read.
- Many textbook on Java includes a part on Databases, cf. for instance [Gaddis2014, Chapter 16].

5.1 Overview

Two options to interact with a database:

- Interactive interface (C.L.I.), what we used so far
- Application program / Database application
  1. Embed SQL commands in your program: a pre-compiler scans the code, extract the SQL commands, execute them on the DBMS.
  2. Use a library, or Application Programming Interface for accessing the database from application programs.
  3. Create a new language that extends SQL (PL/SQL¹)

We will consider option 2. Every database application follows the same routine:

1. Establish / open the connection
2. Interact (Update, Query, Delete, Insert)
3. Terminate / close the connection

<table>
<thead>
<tr>
<th>Python</th>
<th>Python Database API</th>
<th><a href="https://www.python.org/dev/peps/pep-0249/">https://www.python.org/dev/peps/pep-0249/</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
<td>MySQL Connector/Net</td>
<td><a href="https://dev.mysql.com/downloads/connector/net/6.10.html">https://dev.mysql.com/downloads/connector/net/6.10.html</a></td>
</tr>
<tr>
<td>Java</td>
<td>Java DataBase Connectivity</td>
<td><a href="https://docs.oracle.com/javase/9/docs/api/java/sql/package-summary.html">https://docs.oracle.com/javase/9/docs/api/java/sql/package-summary.html</a></td>
</tr>
</tbody>
</table>

¹http://www.oracle.com/technetwork/database/features/plsql/index.html
5.2 Java’s Way

Java actually uses

- A **protocol** (the API, a class library), Java DataBase Connectivity (JDBC), common to all DBMS. Essentially, a collection of classes to send SQL statements, retrieve and update the results of a query, handle exceptions, etc.
- A **subprotocol** (the driver, connector), Connector/J for MySQL.

And the routine is a bit more complex:

1. Import library
2. Load driver (can also be done at execution time)
3. Open connection (create Connection and Statement objects)
4. Interact with DB (use Statement object)
5. Close connection

5.3 Flash Intro to Java

### 5.4 A First Program

#### 5.4.1 The Java code

```java
// javac FirstProg.java
// java -cp
  .:mysql-connector-java-5.1.44/mysql-connector-java-5.1.44-bin.jar
  FirstProg
import java.sql.*;

public class FirstProg {
    public static void main(String[] args) {
        try {
            Connection conn =
                DriverManager.getConnection("jdbc:mysql://localhost:3306/HW_EBOOKSHOP",
                "testuser","password");
            Statement stmt = conn.createStatement();

            String strSelect = "SELECT title, price, qty FROM books WHERE qty
                                > 40";
            System.out.print("The SQL query is: "+ strSelect + "\n");
            ResultSet rset = stmt.executeQuery(strSelect);

            System.out.println("The records selected are:");
            int rowCount = 0;
            String title;
            double price;
            int qty;

            while(rset.next()) {
                title = rset.getString("title");
                price = rset.getDouble("price");
                qty = rset.getInt("qty");
                System.out.println(title + ", " + price + ", " + qty);
                rowCount++;
            }

            System.out.println("Total number of records = "+ rowCount);
            conn.close();
        }
        catch(SQLException ex) {
            ex.printStackTrace();
        }
    }
}
```

A couple of comments:
- `java.sql.*` contains the classes `Connection`, `Statement`, `ResultSet`, `ResultSetMetadata`.
- In the string "jdbc:mysql://localhost:3306/HW_EBOOKSHOP", `jdbc` is the protocol, `mysql` is the subprotocol, `localhost` is the url of the database, 3306 is the port, and
HW_EBOOKSHOP is the schema (that needs to already exist in this case).

- Note that strSelect doesn’t end with ; (it could, but doesn’t have to).
- next() returns true if there is something left in the set of result, and move to the next line if it is the case. It is close to the code we would use to read from a file.
- We could use 1, 2, and 3 instead of "title", "price" and "qty" in the while loop: getString, getDouble and getInt also take integers, corresponding to the position of the attribute in the result set.

### 5.4.2 The database

```sql
/* code/sql/HW_EBOOKSHOP.sql */

CREATE DATABASE HW_EBOOKSHOP;

USE HW_eBOOKShop;

CREATE TABLE BOOKS (
    id int PRIMARY KEY,
    title varchar(50),
    author varchar(50),
    price float,
    qty int
);

INSERT INTO BOOKS VALUES (1001, 'Java for dummies', 'Tan Ah Teck', 11.11, 11);
INSERT INTO BOOKS VALUES (1002, 'More Java for dummies', 'Tan Ah Teck', 22.22, 22);
INSERT INTO BOOKS VALUES (1003, 'More Java for more dummies', 'Mohammad Ali', 33.33, 33);
INSERT INTO BOOKS VALUES (1004, 'A Cup of Java', 'Kumar', 44.44, 44);
INSERT INTO BOOKS VALUES (1005, 'A Teaspoon of Java', 'Kevin Jones', 55.55, 55);

SELECT * FROM BOOKS;
```

```
MariaDB [HW_EBOOKSHOP]> SELECT * FROM books;
+------+----------------------------+--------------+-------+------+
<table>
<thead>
<tr>
<th>id</th>
<th>title</th>
<th>author</th>
<th>price</th>
<th>qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Java for dummies</td>
<td>Tan Ah Teck</td>
<td>11.11</td>
<td>11</td>
</tr>
<tr>
<td>1002</td>
<td>More Java for dummies</td>
<td>Tan Ah Teck</td>
<td>22.22</td>
<td>22</td>
</tr>
<tr>
<td>1003</td>
<td>More Java for more dummies</td>
<td>Mohammad Ali</td>
<td>33.33</td>
<td>33</td>
</tr>
<tr>
<td>1004</td>
<td>A Cup of Java</td>
<td>Kumar</td>
<td>44.44</td>
<td>44</td>
</tr>
<tr>
<td>1005</td>
<td>A Teaspoon of Java</td>
<td>Kevin Jones</td>
<td>55.55</td>
<td>55</td>
</tr>
</tbody>
</table>
+------+----------------------------+--------------+-------+-----+
5 rows in set (0.00 sec)
```
5.4.3 The result

If you store the program in FirstProg.java, compile it, with

```
javac FirstProg.java
```

and then execute it, with

```
java -cp .:mysql-connector-java-5.1.44/mysql-connector-java-5.1.44-bin.jar
   FirstProg
```

then you would obtain:

```
The SQL query is: select title, price, qty from books where qty > 40
The records selected are:
A Cup of Java, 44.44, 44
A Teaspoon of Java, 55.55, 55
Total number of records = 2
```

5.4.4 A variation

If you were to replace the body of try in the previous program with

```java
String strSelect = "SELECT * FROM books";
ResultSet rset = stmt.executeQuery(strSelect);
System.out.println("The records selected are:");
ResultSetMetaData rsmd = rset.getMetaData();
int columnsNumber = rsmd.getColumnCount();
String columnValue;
while (rset.next()) {
    for (int i = 1; i <= columnsNumber; i++) {
        if (i > 1) System.out.print(" ", "");
        columnValue = rset.getString(i);
        System.out.print(columnValue + " "+ rsmd.getColumnName(i));
    }
    System.out.println();
}
conn.close();
```

You would obtain:

```
The records selected are:
1001 id, Java for dummies title, Tan Ah Teck author, 11.11 price, 11 qty
1002 id, More Java for dummies title, Tan Ah Teck author, 22.22 price, 22 qty
1003 id, More Java for more dummies title, Mohammad Ali author, 33.33 price, 33 qty
1004 id, A Cup of Java title, Kumar author, 44.44 price, 44 qty
1005 id, A Teaspoon of Java title, Kevin Jones author, 55.55 price, 55 qty
```
5.5 Mapping Datatypes

<table>
<thead>
<tr>
<th>SQL</th>
<th>JAVA</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>CHARACTER(n)</td>
<td>String</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>String</td>
</tr>
<tr>
<td>REAL</td>
<td>float</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>DECIMAL(t,d)</td>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>DATE</td>
<td>java.sql.Date</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>boolean</td>
</tr>
<tr>
<td>BIT(1)</td>
<td>byte</td>
</tr>
</tbody>
</table>

We can’t always have that correspondance: what would correspond to a reference variable? To a private attribute? This series of problems is called “object-relational impedance mismatch”, it can be overcome, but at a cost.

5.6 Differences Between executeQuery, executeUpdate and execute

<table>
<thead>
<tr>
<th>Name</th>
<th>executeQuery</th>
<th>executeUpdate</th>
<th>execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used for</td>
<td>SELECT</td>
<td>INSERT, UPDATE, DELETE</td>
<td>Any type</td>
</tr>
<tr>
<td>Input Type</td>
<td>string</td>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>Return Type</td>
<td>ResultSet</td>
<td>int, the number of rows affected by the query</td>
<td>boolean, true if the query returned a ResultSet, false if the query returned an int or nothing</td>
</tr>
</tbody>
</table>

To retrieve the ResultSet obtained by an execute statement, you need to use getResultSet or getUpdateCount. For more details, consult https://docs.oracle.com/javase/7/docs/api/java/sql/Statement.html.

5.7 A Second Program

5.7.1 Passing options

We can pass options when connecting to the database:

```java
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/HW_DBPROG" + "?user=testuser" + "&password=password")
```
createDatabaseIfNotExist is about schema, actually.

### 5.7.2 Creating a table

Use `stmt.executeUpdate` and a create statement. You can use the `getMetaData()` of the `DatabaseMetaData` to obtain information about the tables.

### 5.7.3 Inserting values

Use `stmt.executeUpdate` (multiple insertion possible if `allowMultiQueries` was set to true). Another way of batch processing statements:

```java
stmt.addBatch(insert3);
stmt.addBatch(insert4);
stmt.executeBatch();
```

### 5.7.4 Prepared statements

“A query with a slot”: parsed and stored on the database, but not executed. When the program gives values, it is executed.

Compared to executing SQL statements directly, prepared statements have three main advantages:

- Reduces parsing time (one time VS as many time as values)
- Minimize bandwidth (send only the parameters, and not the whole query)
- Protect against SQL injections

### 5.7.5 Advanced Statements objects

You can pass options when creating Statement objects to be able to read it both ways, and to be able to update rows.

COPY HOMEWORK HERE

## Exercises

**Exercise 5.1** What are the technologies that makes it possible for a Java application to communicate with a DBMS?

**Exercise 5.2** What JDBC method do you call to get a connection to a database?

**Exercise 5.3** Briefly explain what the `next()` method from the `ResultSet` class does, and give its return type.

**Exercise 5.4** How do you submit a `SELECT` statement to the DBMS?

**Exercise 5.5** Where is a `ResultSet` object’s cursor initially pointing? How do you move the cursor forward in the result set?
**Exercise 5.6** Give three navigation methods provided by ResultSet.

**Exercise 5.7**

Explain this JDBC URL format:
```java
jdbc:mysql://localhost:3306/HW_NewDB?createDatabaseIfNotExist=true&useSSL=true
```

**Exercise 5.8** In what class is the getColumnName method?

**Exercise 5.9** What is a prepared statement?

**Exercise 5.10** In the code below, there are five errors between line 13 and line 32. They are not subtle Java errors (like misspelling a key word) and do not come from the DBMS (so you should assume that the password is correct, that the database exists, etc.). For each error, highlight it precisely and give a short explanation.

```java
/* code/java/ProgWithErrors.java */
import java.sql.*;

public class ProgWithErrors{
    public static void main(String[] args) {
        try {
            Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/
+i"HW_TestDB?user=testuser&password=password");
            Statement stmt = conn.createStatement();
        }

        String strSelect = "SELECT title FROM DISKS WHERE qty > 40;";
        ResultSet rset = stmt.executeUpdate(strSelect);

        System.out.println("The records selected are: (listed last first): ");
        rset.last();

        while(rset.previous()) {
            String title = rset.getDouble("title");
            System.out.println(title + "\n");
        }

        String sss = "SELECT title FROM DISKS WHERE Price <= ?";
        PreparedStatement ps = conn.prepareStatement(sss);
        ResultSet result = ps.executeQuery();

        conn.close();
    }
}
/* Errors before this point.*/
```
5.7 A Second Program

Exercise 5.11 Write a small program that determine whenever the null value from Java is equal to the NULL value in your DBMS.

Solution to Exercises

Solution 5.1 API + driver

Solution 5.2 DriverManager.getConnection()

Solution 5.3 It checks if there is data to read, and if move the cursor reads it. It returns a Boolean.

Solution 5.4 Using .executeQuery(strSelect)

Solution 5.5 Before the first line. Using the next method.

Solution 5.6 first, last, next, previous, relative, absolute methods.

Solution 5.7 Connect to localhost:3306 and create a new database if needed, and use secure connection.

Solution 5.8 ResultSetMetaData

Solution 5.9 A prepared statement is a feature used to execute the same (or similar) SQL statements repeatedly with high efficiency.

Solution 5.10 The errors are the following:

- l. 13: The query is incorrect (WHERE should come before FROM),
- TO be written

Solution 5.11 Here it is:

```java
import java.sql.*;

public class NullProg {
    public static void main(String[] args) {
        try {
            Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/HW_DBPROG?user=testuser&password=password&createDatabaseIfNotExist=true");
            Statement stmt = conn.createStatement();
            stmt.executeUpdate("CREATE TABLE Test (" +
                             "A CHAR(25), " +
                             "B INTEGER, " +
                             "C DOUBLE)");
            String strAdd = "INSERT INTO Test VALUES (NULL, NULL, NULL);";
        }
    }
}
```
```java
int number_of_row_changed = stmt.executeUpdate(strAdd);
System.out.println("This last query changed " +
   number_of_row_changed + " row(s).");

ResultSet result = stmt.executeQuery("SELECT * FROM Test");
if (result.next()) {
   System.out.print(result.getString(1) + " " +
   result.getDouble(2) + " " + result.getInt(3));
   if (result.getString(1) == null) System.out.print("
And null for CHAR in SQL is null for String in Java."");
}
conn.close();
} catch (SQLException ex) {
   ex.printStackTrace();
}
```

5.8 Problems

**Problem 5.1 (MySQL’s batch mode and HW_EBOOKSHOP.sql)** In the archive, navigate to code/sql/ and open HW_EBOOKSHOP.sql. Then, open a terminal (or command-line interpreter), navigate to the folder where you stored that file (using cd), and type

```bash
mysql -u testuser -p < HW_EBOOKSHOP.sql
```

for linux, or (something like)

```bash
"C:\Program Files\MySQL\MySQL Server 5.7\bin\mysql.exe" -u testuser -p
< HW_EBOOKSHOP.sql
```

for Windows.

You just discovered MySQL’s batch mode, that perform series of instructions from a file. You can easily make sure that the database and the table were indeed created, and the values inserted.

**Problem 5.2 (First Database Application)** This exercise supposes you successfully completed Pb 5.1. We will compile and execute your first database application, using Java and MySQL.

- I will assume that you have MySQL installed and set-up as indicated in Homeworks 5.21 and 5.22.
- I will assume that you have Java installed. If not, please refer to http://spots.augusta.edu/caubert/teaching/general/java/ for a simple program and the instructions to compile and execute it.
- We need to set up the driver (or connector) to make the java sql API and MySQL communicate. To do so,
  - Go to https://dev.mysql.com/downloads/connector/j/
5.8 Problems

- Click on “Download” in front of “Platform Independent (Architecture Independent), ZIP Archive”
- Look for the (somewhat hidden) “No thanks, just start my download.”
- You will download a file named "mysql-connector-java-***.zip", where *** is the version number.
- Upon completion of the download, unzip the file, and locate the "mysql-connector-java-***-bin.jar” file.
- Copy that file in code/java/.

- Open a terminal in that same folder, and compile FirstProg.java, using javac FirstProg.java (or an equivalent command for windows). Normally, nothing will be printed, but a FirstProg.class file will be created.
- Now, execute that program, using ```{bash} java -cp .:mysql-connector-java-***-bin.jar FirstProg ```

in Linux, or

```{bash} java -cp .;mysql-connector-java-***-bin.jar FirstProg ```

in Windows. The -cp option lists the places where java should look for the class used in the program: we are explicitly asking java to use the mysql-connector-java-***-bin.jar executable to execute our FirstProg executable. Try to execute FirstProg without that flag, and see what happens.

**Problem +.## (Advanced Java Programming)** Read, execute, break, edit, compile, patch, hack and (most importantly) understand the following program:

```sql
/* code/java/AdvancedProg.sql */
/
/*
This is a long program, introducing:
I. How to pass options when connecting to the database,
II. How to create a table
III. How to insert values
IV. How to use prepared statements
V. How to read backward and write in ResultSets

If you want to run this program multiple times, you have to either:
1. Comment first statement of II. Creating a table
2. Change the name of the schema, from HW_DBPROG to whatever you want
3. Drop the DVD table: connect to your database, and then enter
   USE HW_DBPROG;
   DROP TABLE DVD;
   Or do it from within your program!

If you use option 1, you will keep inserting tuples in your table:
   cleaning it with
   DELETE FROM DVD;
   can help. You can do it from within the program!
```
```java
import java.sql.*;

public class AdvancedProg {
    public static void main(String[] args) {
        try {
            /*
             * I. Passing options to the database
             */

            Connection conn =
                DriverManager.getConnection("jdbc:mysql://localhost:3306/HW_DBPROG"
                        +
                        "?user=testuser"
                        +
                        "&password=password"
                        +
                        "&allowMultiQueries=true"
                        +
                        "&createDatabaseIfNotExist=true"
                        +
                        "&useSSL=true");

            // Read about other options at
            //
            //
            https://jdbc.postgresql.org/documentation/head/connect.html

            Statement stmt = conn.createStatement();
        } {

            /*
             * II. Creating a table
             */

            stmt.execute("CREATE TABLE DVD (" +
                        "Title CHAR(25) PRIMARY KEY, " +
                        "Minutes INTEGER, " +
                        "Price DOUBLE");

            /* If we were to execute
            * SHOW TABLES
            * directly in the MySQL interpreter, this would display
            * at the screen
            * +--------------------------+
            * | Tables_in_Hw_NewDataBase |
            */
```

**5.8 Problems**
* +--------------------------+
* | DVD               |
* +--------------------------+

But here, to access this information, we will use the connection's metadata.

```
DatabaseMetaData md = conn.getMetaData();
// DatabaseMetaData is a class used to get information about the database: the driver, the user, the versions, etc.

ResultSet rs = md.getTables(null, null, "%", null);
```

```java
while (rs.next()) {
    System.out.println(rs.getString(3));  // In the ResultSet returned by getTables, 3 is the TABLE_NAME.
}
```

<table>
<thead>
<tr>
<th>III. Inserting values</th>
</tr>
</thead>
</table>

String sqlStatement = "INSERT INTO DVD VALUES ('Gone With The Wind', 221, 3);"
int rowsAffected = stmt.executeUpdate(sqlStatement);
System.out.print(sqlStatement + " changed " + rowsAffected + " row(s).\n")

<table>
<thead>
<tr>
<th>* Batch Insertion</th>
</tr>
</thead>
</table>
String insert1 = "INSERT INTO DVD VALUES ('Aa', 129, 0.2)";
String insert2 = "INSERT INTO DVD VALUES ('Bb', 129, 0.2)";
String insert3 = "INSERT INTO DVD VALUES ('Cc', 129, 0.2)";
String insert4 = "INSERT INTO DVD VALUES ('DD', 129, 0.2)";

// Method 1: Using executeUpdate, if the option allowMultiQueries=true was passed in the url given to getConnection and your DBMS supports it.
stmt.executeUpdate(insert1 + ";");
stmt.executeUpdate(insert2);

// Method 2: Using the addBatch and executeBatch methods
stmt.addBatch(insert3);
stmt.addBatch(insert4);
stmt.executeBatch();

/*
 * IV. Prepared Statements
*/

// Example 1
sqlStatement = "SELECT title FROM DVD WHERE Price <= ?";
// We have a string with an empty slot, represented by "?".
PreparedStatement ps = conn.prepareStatement(sqlStatement); // We create a PreparedStatement object, using that string with an empty slot.
float maxprice = 0.5f;
ps.setFloat(1, maxprice); // This statement says "Fill the first slot with the value of maxprice".
ResultSet result = ps.executeQuery(); // And then we can execute the query, and display the results:

System.out.printf("For %.2f you can get:
", maxprice);

while(result.next()){
    System.out.printf("%s
", result.getString(1));
}

// Example 2
sqlStatement = "INSERT INTO DVD VALUES (?, ?, ?)"; // Now, our string has 3 empty slots, and it is an INSERT statement.
PreparedStatement preparedStatement = conn.prepareStatement(sqlStatement);

preparedStatement.setString(1, "The Great Dictator");
preparedStatement.setInt(2, 124);
```java
preparedStatement.setFloat(3, 5.4f);
rowsAffected = preparedStatement.executeUpdate(); // You can check "by hand" that this statement was correctly executed.
System.out.println(preparedStatement.toString() + " changed " + rowsAffected + " row(s).\n
// If we try to mess things up, i.e., provide wrong datatypes:
preparedStatement.setString(1, "The Great Dictator");
preparedStatement.setString(2, "The Great Dictator");
preparedStatement.setString(3, "The Great Dictator");

// Java compiler will be ok, but we'll have an error at execution time when executing the query. You can uncomment the line below to see for yourself.
//rowsAffected = preparedStatement.executeUpdate();

// Of course, we can use prepared statement inside loops.
for (int i = 1; i < 5; i++) {
    preparedStatement.setString(1, "Saw " + i);
    preparedStatement.setInt(2, 100);
    preparedStatement.setFloat(3, .5f);
    preparedStatement.executeUpdate();
}

/*
 * V. Reading backward and writing in ResultSets
 */

// To read backward and write in ResultSets, you need to have a statement with certain options:
Statement stmtNew = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                        ResultSet.CONCUR_UPDATABLE);

/*
 * Those options change two things about the ResultSet we obtain using this statement
 *
 * The first argument is the scrolling level:
 * TYPE_FORWARD_ONLY = default.
 * TYPE_SCROLL_INSENSITIVE = can scroll, but updates don't impact result set.
 * TYPE_SCROLL_SENSITIVE = can scroll, update impact result set.
 *
 * The second argument is the concurrency level:
 */
```
* CONCUR_READ_ONLY: default.
  * CONCUR_UPDATABLE: we can change the database without issuing SQL statement.

```
/*
 * Reading backward
 */

sqlStatement = "SELECT title FROM DVD WHERE Price < 1;"
result = stmtNew.executeQuery(sqlStatement);
System.out.println("For $1, you can get:");
if (result.last()) { // We can jump to the end of the ResultSet
    System.out.print(result.getString("Title") + " ");
}
System.out.println("and also, (in reverse order)");
while (result.previous()) { // Now we can scroll back!
    System.out.print(result.getString("Title") + " ");
}

/*
 * Other methods to navigate in ResultSet:
 * first()
 * last()
 * next()
 * previous()
 * relative(x): move cursor x times (positive = forward, negative = backward)
 * absolute(x): move to the row number x. 1 is the first.
 */

/*
 * Changing the values
 */

System.out.print("\n\nLet us apply a 50% discount.\nCurrently, the prices are:\n");

sqlStatement = "SELECT title, price FROM DVD;"
result = stmtNew.executeQuery(sqlStatement);
while (result.next()) {
    System.out.printf("%20s \t $%3.2f\n", result.getString("title") , result.getDouble("price"));
}
result.absolute(0); // We need to scroll back!

while (result.next()) {
    double current = result.getDouble("price");
    result.updateDouble("price", (current * 0.5));
    result.updateRow();
}
System.out.print("\n\nAfter update, the prices are:\n");

result.absolute(0); // We need to scroll back!

while (result.next()) {
    System.out.printf("%20s %7.2f\n", result.getString("title") , result.getDouble("price"));
    conn.close();
} catch (SQLException ex) {
    ex.printStackTrace();
}
}

Solution to Selected Problems

Solution to Pb 5.2  Without the flag, we obtain:

$ java FirstProg
java.sql.SQLException: No suitable driver found for jdbc:mysql://localhost:3306/HW_EBOOKSHOP
    at java.sql.DriverManager.getConnection(DriverManager.java:689)
    at java.sql.DriverManager.getConnection(DriverManager.java:247)
    at FirstProg.main(FirstProg.java:9)


6 A Bit About Security

Resources

6.1 Usual Aspects

6.1.1 Threat model

- Who is threatening you?
- What are the risks?
  1. Loss of integrity (improper modification)
  2. Loss of availability
  3. Loss of confidentiality (unauthorized disclosure)
- “You are as strong as your weakest link.”
- Never trust the user or their computer.

6.1.2 Control measures

- Access control (user account, passwords, restrictions)
- Inference control (can’t access information about a particular “case”)
- Flow control (prevent indirect access)
- Encryption (salting + encrypting, can be a legal obligation): password + salt -> hashed.

Insert short intro. to salting, cryptography.

6.2 Specificities Of Databases

6.2.1 Attack

Attacks: buffer overflow, denial of service, weak authentication, privilege escalation, SQL injections.

“Mixing the instructions with the data”: a judge asking “what is your name”, and you answer “Bill, you are now free to go”.

Example with ASP, Active Server Pages, a server-side scripting language:

```java
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

1. `105; DROP TABLE Suppliers;` Execute remote command
2. `105 or 1 = 1` Exploit, bypass login screen
3. `admin|--` Line comment, privilege escalation

Can also be used for DBMS fingerprinting.
6.2.2 Protections

1. Backups:
   
   ```
   mysqldump --all-databases -u testuser -p password -h localhost > dump.sql
   ```

2. Prepared Statements (a.k.a. stored procedures)

3. White list input validation

4. Escaping

5. Be up-to-date, desactivate the options you are not using, read newsfeeds,
7 Presentation of NoSQL

Resources

7.1 A Bit of History

Inspired from (Sadalage and Fowler 2012, Chap. 1)

7.1.1 DB applications and application DB

When you write a DB application, you have two options:

1. One database for many softwares
2. One database for each softwares

Option a. can cause severe impacts on the efficiency of your database: since maintaining the integrity of the database is a requirement, a lot of synchronization is needed. With option b., you develop an “application database”, and you have more freedom of choice: since only a program interact with a database, you can chose whatever data management you want.

But people were attached to SQL and kept using it.

7.1.2 Clusters, clusters...

Increase in everything (traffic, size of data, number of clients, etc.) meant "up or out", and there was two ways to increase the resources:

1. Bigger machines
2. More machines

Option b. was generally less expensive, but came with two drawbacks w.r.t. databases:

1. Cost of licences,
2. Force to perform “unnatural acts”: relational model are really not made to be distributed
7.1 A Bit of History

7.1.3 A first shift

- Google Big Table\(^1\), 2004 (made public in ... 2015!) (Chang et al. 2006)
- Amazon DynamoDB\(^2\), 2004 (used in Simple Storage Service (S3) in 2007)
- Facebook’s Cassandra is sometimes mentioned, but it came later on, around 2009 (Lakshman and Malik 2009).

Particular, big company, with specific needs, but people interested in solving some of their problems. Now, people started to think that there could be other ways.

One goal was to get rid of “impedance mismatch”: mapping classes or objects to database tables defined by a relational schema is complex and cumbersome.

Some issues:

- No absolute notion of “private” and “public” in RDBMS (relative to needs)
- Data-type differences (no pointer, weird way of defining string, etc.)
- Value in a relational structure have to be simple (no complex datatype, no structure)

“Impedance mismatch” is that annoying need for a translation.

Also, the data is now

- moving
- growing
- too diverse

for traditional relational DBMS.

7.1.4 Gathering forces

Multiple attempts, going in multiple directions. A meetup to discuss them coined the term “NoSQL” in an attempt to have a “twittable” hashtag, and it stayed (even it is as specific as describing a dog with “no-cat”). The original meet-up asked for “open-source, distributed, nonrelational database”. Today, no official definition, but NoSQL often implies the following:

- No relational model
- Not using SQL. Some still have a query language, and it resembles SQL (to minimize learning cost), for instance Cassandra’s CQL.
- Run well on clusters
- Schemaless: you can add records without having to define a change in the structure first.
- Open source

Most importantly: polyglot persistence, “using different data storage technologies to handle varying data storage needs.”

7.1.5 The future?

A lot of enthusiasm, also because it “frees the data” (and, actually, the metadata, cf. application/ld+json, JavaScript Object Notation for Linked Data, schema.org, etc.). Some of it will last for sure: polyglot persistency, the possibility of being schema-less, being “distributed first”, the possibility of sacrificing consistency for greater good, etc. Doesn’t mean SQL (“OldSQL”) and

\(^1\)https://cloud.google.com/bigtable/
\(^2\)https://aws.amazon.com/dynamodb/
relational database are over: still useful in many scenario, and the powerful query language is great (writing your own every time is a nightmare...).

Starting ~ 2010, one reaction was to develop “NewSQL”, which would combine aspects of both approaches. MongoDB announced that it would have more and more of the ACID properties! https://www.mongodb.com/blog/post/multi-document-transactions-in-mongodb

Also, a really great use of NoSQL is to adopt it at an early stage of the development, when it isn’t clear what the schemas should be. When the schemas are final, then you can shift to relational DBMS!

### 7.2 Comparison

#### 7.2.1 Overview

« Comparaison n’est pas raison »

- Semi-structured data (no schema)
- High performance
- Availability
- Data Replication (improves availability and performance)
- Scalability (horizontal scalability (add nodes) instead of vertical (add memory))
- Eventual Consistency
- Natively versionning

Vs

- Immediate data consistency
- Powerful query language (join is missing from SQL, has to be implemented on the application-side)
- Structured data storage (can be too restrictive)

#### 7.2.2 ACID Vs CAP

ACID is the guarantee of validity even in the event of errors, power failures, etc.

- Atomicity → Transactions are all or nothing
- Consistency → Transactions maintains validity
- Isolation → Executing two transactions in parallel or one after the other would have the same result
- Durability → Once a transaction has been committed, it is stored in non-volatile memory.

CAP (a.k.a. Brewer’s theorem): Roughly, “In a distributed system, one has to choose between consistency (every read receives the most recent write or an error) and availability (every request receives a (non-error) response, without guarantee that it contains the most recent write)” (the P. standing for “Partition tolerance”).

### 7.3 Categories of NoSQL Systems
### 7.4 MongoDB

**7.4.1 Resources**

- [https://oss.sonatype.org/content/repositories/releases/org/mongodb/mongo-java-driver/3.7.0-rc0/](https://oss.sonatype.org/content/repositories/releases/org/mongodb/mongo-java-driver/3.7.0-rc0/)
- [https://jsonlint.com/](https://jsonlint.com/)
- [https://mongodb.github.io/mongo-java-driver/3.4/driver/getting-started/quick-start/](https://mongodb.github.io/mongo-java-driver/3.4/driver/getting-started/quick-start/)
- [https://docs.mongodb.com/manual/administration/security-checklist/](https://docs.mongodb.com/manual/administration/security-checklist/)
- [https://docs.mongodb.com/getting-started/shell/](https://docs.mongodb.com/getting-started/shell/)
- [https://university.mongodb.com](https://university.mongodb.com)
- [https://en.wikipedia.org/wiki/MongoDB](https://en.wikipedia.org/wiki/MongoDB)
- [https://www.w3schools.com/xml/schema_example.asp](https://www.w3schools.com/xml/schema_example.asp)
- [https://couchdb.apache.org/](https://couchdb.apache.org/)
- [https://www.mongodb.com/](https://www.mongodb.com/)
- [https://www.couchbase.com/](https://www.couchbase.com/)
- [https://redis.io/](https://redis.io/)
- [https://cassandra.apache.org/](https://cassandra.apache.org/)
- [https://hbase.apache.org/](https://hbase.apache.org/)
- [https://neo4j.com/](https://neo4j.com/)
- [https://ignite.apache.org/](https://ignite.apache.org/)
- [https://www.arangodb.com/](https://www.arangodb.com/)
7.4 MongoDB

7.4.2 Introduction

MongoDB is

- free (business model: training, support, DB as service, they actually developed MongoDB because they wanted a good solution for a cloud solution!),
- open-source
- cross-platform
- document-oriented (JSON-like documents with schemas).

And there are drivers for C, C++, C#, Hadoop Connector, Haskell, Java, node.js, PHP, Perl, Python, Ruby, Scala (Casbah).

The mongo shell is an interactive JavaScript interface to MongoDB.

7.4.3 Document-oriented database

Document-oriented database (document store) contains semi-structured data, it is a subclass of the key-value store:

- Relational databases (RDB) pre-define the data structure in the database (fields + data type).
- Key-value (KV) treats the data as a single opaque collection, which may have any number (incl. 0) fields for every record.
- Document-oriented (DO) system relies on internal structure in the data to extract metadata.

RDB is excellent for optimization, but waste space (placeholders for optional values). KV doesn’t allow any optimization, but flexibility and more closely follows modern programming concepts. DO has the flexibility of KV, and allow for some optimization.

One important difference: in RDB, data is stored in separate tables, and a single object (entity) may be spread across several tables. In DO, one object = one instance, and every stored object can be different from every other.

Pro:

- Mapping objects to a DB simpler
- Change “in place”
- Increase speed of deployment

Document:

- Implementations differ on the details of the definition, but always the central notion. MongoDB has its own implementation, but there are ~ 45 others. MongoDB is the most popular one (next: Amazon DynamoDB, Couchbase, CouchDB)
- Documents encapsulate and encode data (Self-Describing Data)
- Do not need to adhere a standard schema.
- One program can have many different types of objects, and those objects often have many optional fields

12https://docs.mongodb.com/ecosystem/drivers/
• Formats: XML, YAML, JSON, PDF, etc.

MongoDB uses JSON to BSON (portmanteau of the words “binary” and “JSON”), and actually extend JSON. Think of BSON as a binary representation of JSON (JavaScript Object Notation) documents.

An example of XML (Extensible Markup Language) document (you can actually convert from XML to JSON):

```xml
<shiporder orderid="889923">
    <orderperson>John Smith</orderperson>
    <shipto>
        <name>Ola Nordmann</name>
        <address>Langgt 23</address>
        <city>4000 Stavanger</city>
        <country>Norway</country>
    </shipto>
    <item>
        <title>Empire Burlesque</title>
        <note>Special Edition</note>
        <quantity>1</quantity>
        <price>10.90</price>
    </item>
    <item>
        <title>Hide your heart</title>
        <quantity>1</quantity>
        <price>9.90</price>
    </item>
</shiporder>
```

• Invalid document exists!
• Human and computer-readable
• No predefined tags
• Extensible

### 7.4.4 General organization of MongoDB databases

<table>
<thead>
<tr>
<th>RDBMS</th>
<th>MongoDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>database instance</td>
<td>MongoDB instance</td>
</tr>
<tr>
<td>schema</td>
<td>database</td>
</tr>
<tr>
<td>table</td>
<td>collection</td>
</tr>
<tr>
<td>row</td>
<td>document</td>
</tr>
</tbody>
</table>

Each MongoDB instance has multiple databases, each database can have multiple collections.

Two documents (delimited by [...], used to delimit an arry of document):

```json
[
    {
        "firstname": "Martin",
        "likes": [ "Biking",
```
Use the same example all along!

- addresses is a document embedded in a document!
- Some attributes are common, some are not: that’s fine, every document can have its own schema.

A collection should be on “related” entities (do not store server logs, store customers and list of employee in the same collection!), and not too abstract ones (no “Server stuff”). Also, if you store document that are too different, your performances will take a big hit. Bottom line: think about your usage, and the kind of queries you will perform.

"Schema-less" does not mean “organization-less”!

### 7.4.5 First elements of syntax

```javascript
db.<name of the collection>.<command>
```

(db is not the name of the database, it is just the prefix).

```javascript
db.book.insert({"title": "Mother Night", "author": "Kurt Blabal"})
```

MongoDB will add a unique identifier (_id) if you do not provide one. You can think of that as a primary key.

```javascript
db.books.remove({"title": "Mother Night"})
```

```javascript
db.books.update({"title": "Mother Night"}, {\$set {"quantity" : 10}})
```

Other function, such as $inc, to increment.

```javascript
db.books.find() is like SELECT * FROM Books;
```

```javascript
db.books.find({"title": "Mother Night"})
```

```javascript
db.books.find({"title": "Mother Night"}, {"author":1, "quantity":1})
```

```javascript
db.books.find({"title": "Mother Night"}, {"author":0, "quantity":0}): everything but the author and the quantity.
```
db.books.find({"quantity":{"$gte": 10, "$lt": 50}}): greater than equal to 10, less than 50.

Possibility to mimic some features (unique attributes), but no referential key integrity, for instance.

Most insert / update / delete will return success as soon as one node received your command, but you may tweak them so that success is returned only once the operation has been performed on the majority of the nodes.

### 7.4.6 Set up

Install mongodb (non-official version, not maintained by MongoDB) and download https://mongodb.github.io/mongo-java-driver/.

Command-line: mongo

Not a lot of features, need to write a lot on the program side. But there are tons of API ("package manager" approach), cf. for instance an API over mongo-java-driver: http://jongo.org/ (support some form of prepared statement)

The documentation is nicely written and well-organized: we’ll follow parts of it, please refer to it if needed.

### 7.4.7 Example


Compile and execute with

```
javac -cp .:mongo-java-driver-3.7.0-rc0.jar MongoTest2.java
java -cp .:mongo-java-driver-3.7.0-rc0.jar MongoTest2
```

After various import statement, and the usual header:

```java
MongoClientURI connectionString = new
    MongoClientURI("mongodb://localhost:27017");
MongoClient mongoClient = new MongoClient(connectionString);
```

Or, more compact:

```java
MongoClient mongoClient = new MongoClient();
```

Get a collection:

```java
MongoDatabase database = mongoClient.getDatabase("mydb");
MongoCollection<Document> collection = database.getCollection("test");
```

Assume we want to create the following document:

```java
{
    "name" : "MongoDB",
    "type" : "database",
    "count" : 1,
    "versions": [ "v3.2", "v3.0", "v2.6" ]
}
Then we can use the Document class, and then insert it:

```java
document = new Document("name", "MongoDB");
document.append("type", "database");
document.append("count", 1);
document.append("versions", Arrays.asList("v3.2", "v3.0", "v2.6"));
document.append("info", new Document("level", "easy").append("used", "yes"));
```

We can “chain” the append: document.append("type", "database").append("count", 1);

And then insert:

`collection.insertOne(document);`

Only at this point would the database and collection being created. To make sure everything went right:

```bash
mongo show dbs
use mydb
show collections
db.collection.find()
{ "_id" : ObjectId("5ae08a7252cbeb2717712b9f"), "name" : "MongoDB" ... }
```

We can construct lists of documents and insert them:

```java
List<Document> documents = new ArrayList<Document>();
for (int i = 0; i < 10; i++) {
documents.add(new Document("i", i));
}
collection.insertMany(documents);
```

### 7.5 Principles

- “Schemaless means more responsibility”
- Some denormalization, sometimes: duplicate the information, to have it all in one place. Example: table for phone number, for employee, for emergency contact. You can duplicate that information, no big deal. Less join (resources expensive), but need more storage, more functions, to substitute.
- NoSQL injection: your application should accept only strings from your users (never allow objects by design) and sanitize the inputs before using them (mongo-sanitize is a good module for this).

### Exercises

**Exercise 7.1** What is polyglot persistence? Is it useful?

**Exercise 7.2** What does it mean to be “schemaless”? What does it imply?
Exercise 7.3  What is denormalization? When could that be useful?

Exercise 7.4  What is the (object-relational) impedance mismatch? Is it an issue that can’t be overcome?

Solution to Exercises

Solution 7.1  It is the task of picking the right DBMS for the task, and to involve multiple DBMS in a single application. Yes, it is useful. Per wikipedia[^13], “Polyglot persistence is the concept of using different data storage technologies to handle different data storage needs within a given software application.”.

Solution 7.2  That a table can contain documents, or tuples, with different attributes. It implies more responsibilities.

Solution 7.3  To duplicate data about other entities in some entities. It is useful when joining is expensive.

Solution 7.4  Data-base and object-oriented principles are different and it requires work to make them work together. This correspondance, or matching, can be implemented in the application, or lead to the design of new DBMS.

[^13]: en.wikipedia.org/wiki/Polyglot_persistence
References


Sullivan, Dan. 2015. NoSQL for Mere Mortals. Addison-Wesley Professional.