

Lu Sun, and many more.

A Notebook on Linux Operating System



*To my family, friends and communities members who
have been dedicating to the presentation of this
notebook, and to all students, researchers and faculty
members who might find this notebook helpful.*



Contents

Foreword	xv
Preface	xvii
List of Figures	xix
List of Tables	xxiii
I Linux Basics	3
1 Brief Introduction to Linux	5
1.1 Linux as an Operating System	5
1.2 A Brief History of Linux	6
1.3 Linux Distributions	7
1.3.1 Red-Hat-Based Distributions	8
1.3.2 Debian-Based Distributions	9
1.4 Linux Graphical Desktop	9
1.5 Linux Installation	9
2 Shell	13
2.1 Shell as a Command Line Interface	13
2.1.1 Shell Types	14
2.1.2 Prompt	14
2.2 Basic Operations	16
2.2.1 Displaying User/Machine Information	16
2.2.2 Set Alias and Shortcuts	16
2.2.3 Check Manuals of a Command	17
2.2.4 Use Comments	17
2.2.5 Others	17
2.3 Shell Environment Variables	18
2.4 Customization of Shell	19
2.5 Shell Script Programming Basics	20
3 Text File Editing	23
3.1 General Introduction to Vim	23
3.2 Vim Modes	24
3.3 Vim Profile Configuration	27

3.3.1	Mapping Shortcuts	27
3.3.2	Syntax and Color Scheme	28
3.3.3	Other Useful Setups	28
3.3.4	Plug Tools	28
3.4	Basic Operations in <i>Vim</i>	29
3.4.1	Cut, Change, Copy and Paste	30
3.4.2	Search in the Text	32
3.4.3	Other Tips	33
3.5	Visual Modes	34
3.6	Vim Macros	35
3.7	File Explorer and Screen Splitting	36
3.8	<i>NeoVim</i>	37
3.9	Other Text Editors	37
4	File Management	41
4.1	Filesystem Hierarchy Standard	41
4.2	Commonly Used File Exploring Commands	45
4.2.1	Print Working Directory	45
4.2.2	List Information about Files and Directories	45
4.2.3	Create Files and Directories	46
4.2.4	Move, Copy-and-Paste, and Remove Files and Directories	48
4.2.5	Use of Wildcard Characters	49
4.3	Access Control List	49
4.3.1	Change Ownership and Group of a File or Directory .	50
4.3.2	Change Permissions of a File or Directory	51
4.3.3	Change Default Permissions	51
4.4	Search through the System	52
4.4.1	Look for a Command	52
4.4.2	Looking for Files by Metadata	52
4.4.3	Looking for Files by Variety of Properties	53
4.5	File Archive	55
5	Software Management	57
5.1	Tasks and Approaches	57
5.2	RPM Package	58
5.2.1	Brief Introduction	58
5.2.2	Package Management Using <code>rpm</code>	59
5.2.3	Package Management Using <code>yum</code> and <code>dnf</code>	59
5.3	DEB Package	61
5.4	Linux Kernel Management	62

<i>Contents</i>	vii
6 Process Management	65
6.1 General Introduction to Process	65
6.1.1 Process	65
6.1.2 Thread	67
6.2 Process Management in Linux	68
6.2.1 Monitor Process	69
6.2.2 Terminate Process	70
6.2.3 Switch between Foreground and Background	71
6.2.4 Change Process Priority	71
II Linux Advanced	73
7 Administration	75
7.1 Introduction to Linux Administration	75
7.2 Root Account Management	75
7.3 User Management	77
8 Storage Management	79
8.1 Partitions and Filesystems	79
8.2 Disk Partition Table Manipulation	82
8.2.1 Disk Partition	82
8.2.2 Disk Partition Table Manipulation	82
8.3 Mount, Unmount and Format a Partition	83
9 Service Control	85
9.1 Service Control	85
III Widely Used Services	87
10 Git	89
10.1 Introduction	89
10.2 Setup	90
10.2.1 Installation	90
10.2.2 Configuration	90
10.3 Local Repository Management	91
10.3.1 Initialization of a Repository	91
10.3.2 Version Tracking	92
10.3.3 Branch Management	96
10.4 Remote Repository Management	99
10.5 GitHub Actions	101
11 Database	103
11.1 Relational Database	103
11.2 Non-Relational Database	105
11.3 Structured Query Language	106
11.3.1 RDB Naming Conventions	106

11.3.2 General Introduction to SQL	107
11.3.3 General Syntax	111
11.3.4 Database Manipulation	112
11.3.5 Table Manipulation	112
11.3.6 Row Manipulation	115
11.3.7 Query	115
11.3.8 Trigger	123
11.3.9 SQL Demonstrative Example	124
11.4 General Ways to Connect to RDB	129
12 RDB Example: MySQL and MariaDB	133
12.1 Installation	134
12.2 MariaDB Basic Configuration	135
12.3 DBMS Console	135
12.4 Run SQL File	136
13 RDB Example: PostgreSQL	137
13.1 Brief Introduction	137
13.2 Installation and Authentication	138
13.2.1 Installation	138
13.2.2 Regular Authentication	139
13.2.3 Peer Authentication	139
13.2.4 Run PostgreSQL in Containers	140
13.3 PostgreSQL-Specific New Data Types	140
13.4 PostgreSQL User-Defined Data Types	141
13.5 PostgreSQL Stored Procedural and Functions	142
13.6 Manipulation and Query	144
14 Database	147
14.1 Relational Database	147
14.2 Non-Relational Database	149
14.3 Structured Query Language	150
14.3.1 RDB Naming Conventions	150
14.3.2 General Introduction to SQL	151
14.3.3 General Syntax	155
14.3.4 Database Manipulation	156
14.3.5 Table Manipulation	156
14.3.6 Row Manipulation	159
14.3.7 Query	159
14.3.8 Trigger	167
14.3.9 SQL Demonstrative Example	168
14.4 General Ways to Connect to RDB	173

15 RDB Example: MySQL and MariaDB	177
15.1 Installation	178
15.2 MariaDB Basic Configuration	179
15.3 DBMS Console	179
15.4 Run SQL File	180
16 RDB Example: PostgreSQL	181
16.1 Brief Introduction	181
16.2 Installation and Authentication	182
16.2.1 Installation	183
16.2.2 Regular Authentication	183
16.2.3 Peer Authentication	183
16.2.4 Run PostgreSQL in Containers	184
16.3 PostgreSQL-Specific New Data Types	185
16.4 PostgreSQL User-Defined Data Types	185
16.5 PostgreSQL Stored Procedural and Functions	186
16.6 Manipulation and Query	188
16.7 Non-RDB Example: MongoDB	189
16.7.1 Installation	191
16.7.2 Basic Global Operations	192
16.7.3 Data Format	193
16.7.4 Create Collection and Document	193
16.7.5 Query	195
16.7.6 Update and Remove Document	196
16.7.7 Sharding and Indexing	197
16.7.8 Other Features	197
16.8 Non-RDB Example: Redis	198
16.9 Non-RDB Example: AllegroGraph	198
16.10 Database Accessory	199
16.10.1 RDB Interface with Python	199
17 NoSQL Database Example: MongoDB	203
17.1 Features and Data Model	204
17.1.1 Features	204
17.1.2 Data Model	205
17.2 Installation	206
17.3 MongoDB Shell	207
17.3.1 Basic Operation	207
17.3.2 Create Collection and Document	208
17.3.3 Query	210
17.3.4 Update Document	212
17.3.5 Remove Document	213
17.4 MongoDB Compass	213
17.5 Advanced Query	213
17.5.1 More about <code>find()</code>	213

17.5.2	Sorting	214
17.5.3	Limiting the Number of Returns	215
17.5.4	Limiting the Fields of Returns	215
17.5.5	Counting the Number of Returns	215
17.6	Sharding and Indexing	216
17.6.1	Sharding	216
17.6.2	Indexing	216
17.6.3	Simple and Compound Index	217
17.6.4	Multi-key Index	218
17.6.5	Hide and Remove Index	218
17.6.6	Performance Evaluation with Index	219
17.7	Third-Party Connection	220
17.8	Aggregation Framework	221
17.8.1	Aggregation Pipeline Syntax	222
17.8.2	Stage: \$match	223
17.8.3	Stage: \$group	224
17.8.4	Stage: \$sort	225
17.8.5	Stage: \$limit	225
17.8.6	Stage: \$set	225
17.8.7	Stage: \$count	226
17.8.8	Stage: \$project	226
17.8.9	Stage: \$out	228
17.9	MongoDB Atlas	228
17.9.1	MongoDB Atlas Dashboard	229
17.9.2	Connection String	229
17.9.3	Atlas Search	231
17.9.4	Schema Pattern	234
18	Non-RDB Example: Redis	235
18.1	Brief Introduction to Redis	235
18.2	Installation	236
18.3	Basic Operations	237
18.3.1	Key-Value Pair Operations	237
18.3.2	Array Operations	238
18.3.3	Set Operations	238
18.3.4	Hashes	239
18.4	Redis in Practice	239
19	Virtualization and Containerization	241
19.1	Introduction	242
19.2	Virtualization and Containerization	245
19.2.1	Virtualization	245
19.2.2	Containerization	247
19.3	Docker Container Basics	248
19.3.1	Docker Engine VS Alternatives	248

<i>Contents</i>	xi
19.3.2 Docker Installation	249
19.3.3 Docker Container Management	250
19.3.4 An Example: Deploy a Containerized Application	256
19.4 Docker Volume and Bind Mount	257
19.4.1 Volume	258
19.4.2 Bind Mount	259
19.4.3 Multiple Volumes	259
19.5 Container Communication	260
19.5.1 Communication with Host Machine	260
19.5.2 Communication with Other Containers	261
19.6 Docker Image	261
19.6.1 Dockerfile Programming	262
19.6.2 Docker Image Management	268
19.6.3 Docker Image Sharing with Docker Hub	269
19.7 Multi-Container Ochestration	269
19.7.1 Protainer	270
20 Kubernetes	273
20.1 Kubernetes Basics	273
20.1.1 Infrastructure	274
20.1.2 Installation	275
20.1.3 Kubernetes Configuration Files	276
20.1.4 Cluster Deployment	277
20.1.5 Cluster Update	281
20.2 Kubernetes Advanced	281
20.2.1 Kubernetes Object: Deployment	281
20.2.2 Kubernetes Object: Service	283
20.2.3 Kubernetes Object: Persistent Volume Claim, Persistent Volume, and Volume	284
20.2.4 Kubernetes Object: Secrets	287
20.2.5 Kubernetes Environment Variables	287
20.3 Container Deployment in Production Environment	288
20.3.1 Setup Cloud Account	289
20.3.2 Configure CI/CD	290
20.3.3 Deploy Containers	290
20.3.4 Manage Secrets	291
20.3.5 Helm	291
21 HTTP Server	293
21.1 Brief Introduction to Apache HTTP Server	293
21.2 Installation of Apache HTTP Server	294
21.2.1 Apache HTTP Server Installation on Host Machine	294
21.2.2 Apache HTTP Server Execution in Container	294
21.3 Apache HTTP Server Configuration and Deployment	295
21.3.1 Virtual Host Configuration	296

21.3.2 Kerberos Authentication Configuration	297
21.4 A Brief Introduction to Website Development	297
21.5 Other Web Servers	297
IV Linux Security	299
22 Introduction to OS Security	301
22.1 Basics	301
22.1.1 Risks and Attacks	301
22.1.2 General Security Architecture	302
22.1.3 Standards and Requirements	304
22.2 Elements of Security	305
22.2.1 Security Policy	305
22.2.2 Security Mechanism	305
22.2.3 Security Assurance	307
22.2.4 Trusted Computing Base	308
22.3 Access Control	309
22.3.1 Discretionary Access Control	310
22.3.2 Mandatory Access Control	311
23 Security of Services and Applications	313
23.1 Database Security	313
23.1.1 Database Security Risks Categories	314
23.1.2 Database Access Control	315
23.1.3 Security-Enhanced DBMS Solutions in a Glance	317
23.1.4 Outsourced Database Security	318
23.1.5 Big Data Security	319
23.2 Virtualization Security	320
23.2.1 Security Concerns	320
23.2.2 VMM Security	320
A Scripts	323
A.1 <i>Vim</i> Configuration <i>vimrc</i> Used in Section 3	323
A.2 <i>Neovim</i> Configuration Files	324
B Continuous Integration and Delivery	325
B.1 Agile VS Waterfall	326
B.1.1 Waterfall	326
B.1.2 Agile	327
B.1.3 Roles in Agile-based Development	327
B.2 Continuous Integration Continuous Delivery	329
B.2.1 Pipeline	329
B.2.2 Continuous Integration	330
B.2.3 Continuous Delivery	331
B.3 <i>Github</i> Action (Part I: Basics)	333
B.3.1 Framework	334

<i>Contents</i>	xiii
B.3.2 <i>Actions</i> Triggers and Types	335
B.3.3 <i>Actions</i> Marketplace	336
B.3.4 Costs	336
B.4 <i>GitHub Actions</i> (Part II: Practice)	337
B.4.1 Trigger	337
B.4.2 Workflow	338
B.4.3 Execution	340
B.4.4 Very Complicated Actions	340
C A Brief Introduction to YAML	343
C.1 Overview	343
C.2 Syntax	344
C.2.1 Key-Value Pair	344
C.2.2 Object and Nested Object	344
C.2.3 List	345
C.2.4 Multi-line String	346
C.3 Commonly Seen YAML Use Cases	347
D Scripts	349
D.1 <i>Vim</i> Configuration <code>vimrc</code> Used in Section 3	349
D.2 <i>Neovim</i> Configuration Files	350
E Continuous Integration and Delivery	351
E.1 Agile VS Waterfall	352
E.1.1 Waterfall	352
E.1.2 Agile	353
E.1.3 Roles in Agile-based Development	353
E.2 Continuous Integration Continuous Delivery	355
E.2.1 Pipeline	355
E.2.2 Continuous Integration	356
E.2.3 Continuous Delivery	357
E.3 <i>GitHub Actions</i> (Part I: Basics)	358
E.3.1 Framework	360
E.3.2 <i>Actions</i> Triggers and Types	361
E.3.3 <i>Actions</i> Marketplace	362
E.3.4 Costs	362
E.4 <i>GitHub Actions</i> (Part II: Practice)	363
E.4.1 Trigger	363
E.4.2 Workflow	364
E.4.3 Execution	366
E.4.4 Very Complicated Actions	366

F A Brief Introduction to YAML	369
F.1 Overview	369
F.2 Syntax	370
F.2.1 Key-Value Pair	370
F.2.2 Object and Nested Object	370
F.2.3 List	371
F.2.4 Multi-line String	372
F.3 Commonly Seen YAML Use Cases	373
Bibliography	375
Index	377

Foreword

If software and e-books can be made completely open-source, why not a notebook?

This brings me back to the summer of 2009 when I started my third year as a high school student in Harbin No. 3 High School. In the end of August when the results of Gaokao (National College Entrance Examination of China, annually held in July) are released, people from photocopy shops would start selling notebooks photocopies that they claim to be from the top scorers of the exam. Much curious as I was about what these notebooks look like, never have I expected myself to actually learn anything from them, mainly for the following three reasons.

First of all, some (in fact many) of these notebooks were more difficult to understand than the textbooks. I guess we cannot blame the top scorers for being so smart that they sometimes make things extremely brief or overwhelmingly complicated.

Secondly, why would I want to adapt to notebooks of others when I had my own notebooks which in my opinion should be just as good as theirs.

And lastly, as a student in the top-tier high school myself, I knew that the top scorers of the coming year would probably be a schoolmate or a classmate. Why would I want to pay that much money to a complete stranger in a photocopy shop for my friend's notebook, rather than requesting a copy from him or her directly?

However, my mind changed after becoming an undergraduate student in 2010. There were so many modules and materials to learn for a college student, and as an unfortunate result, students were often distracted from digging deeply into a module (For those who were still able to do so, you have my highest respect). The situation became worse when I started pursuing my Ph.D. in 2014. As I had to focus on specific research areas entirely, I could hardly split much time on other irrelevant but still important and interesting contents.

This motivated me to start reading and taking notebooks for selected books and articles, just to force myself to spent time learning new subjects out of my comfort zone. I used to take hand-written notebooks. My very first notebook was on *Numerical Analysis*, an entrance level module for engineering background graduate students. Till today I still have on my hand dozens of these notebooks. Eventually, one day it suddenly came to me: why not digitalize them, and make them accessible online and open-source, and let everyone read and edit it?

As most of the open-source software, this notebook (and it applies to the other notebooks in this series as well) does not come with any “warranty” of any kind, meaning that there is no guarantee for the statement and knowledge in this notebook to be absolutely correct as it is not peer reviewed. **Do NOT cite this notebook in your academic research paper or book!** Of course, if you find anything helpful with your research, please trace back to the origin of the citation and double confirm it yourself, then on top of that determine whether or not to use it in your research.

This notebook is suitable as:

- a quick reference guide;
- a brief introduction for beginners of the module;
- a “cheat sheet” for students to prepare for the exam (Don’t bring it to the exam unless it is allowed by your lecturer!) or for lecturers to prepare the teaching materials.

This notebook is NOT suitable as:

- a direct research reference;
- a replacement to the textbook;

because as explained the notebook is NOT peer reviewed and it is meant to be simple and easy to read. It is not necessary brief, but all the tedious explanation and derivation, if any, shall be “fold into appendix” and a reader can easily skip those things without any interruption to the reading experience.

Although this notebook is open-source, the reference materials of this notebook, including textbooks, journal papers, conference proceedings, etc., may not be open-source. Very likely many of these reference materials are licensed or copyrighted. Please legitimately access these materials and properly use them.

Some of the figures in this notebook is drawn using Excalidraw, a very interesting tool for machine to emulate hand-writing. The Excalidraw project can be found in GitHub, [excalidraw/excalidraw](https://github.com/excalidraw/excalidraw).

Preface

References of this notebook include the Linux Bible (10th edition) that I borrowed from National Library Singapore, and also many Bilibili and YouTube videos which I will cite as going through the notebook.



List of Figures

1.1	GNOME desktop environment.	10
1.2	KDE desktop environment.	10
1.3	LXDE desktop environment.	11
1.4	Xfce desktop environment.	11
3.1	Mode switching between normal mode and insert mode, and basic functions associated with the modes.	26
3.2	A flowchart for simple creating, editing and saving of a text file using <i>Vim</i>	26
3.3	A piece of text of “William Shakespeare”, for demonstration.	30
3.4	Search “he” in the piece of text of “William Shakespeare” . .	33
3.5	An example of visual mode where a block of text is selected..	34
3.6	<i>Vim</i> (with user’s profile customization as introduced in this chapter).	38
3.7	<i>Nano</i>	39
3.8	<i>Emacs</i>	39
3.9	<i>Gedit</i>	40
3.10	<i>Visual Studio Code</i>	40
4.1	An example of Linux file system hierarchy.	42
4.2	A rough categorization of commonly used directories in Linux file hierarchy standard.	43
4.3	List down information of files and subdirectories in the current working directory.	46
4.4	Change ownership and group of a file.	50
4.5	Change 9-bit permission (mode) of a file.	51
4.6	Search for files and directories using <i>locate</i>	53
6.1	A demonstration of running multiple processes on a single-core CPU.	66
6.2	Fundamental states of a process and their transferring. . . .	67
6.3	A demonstration of multiple threads in a process.	68
6.4	Execution of <code>ps -ef</code> command.	69
6.5	Execution of <code>top</code> command.	70
6.6	Priority levels in Linux.	72
8.1	A demonstration of how a hard drive is used in the OS. . . .	80

8.2 A demonstration of how partitions and filesystems relate to each other.	81
10.1 <i>Git</i> for software development management.	90
10.2 The project directory managed by <i>Git</i>	93
10.3 Two approaches of integrating branches, <code>git merge</code> VS <code>git rebase</code>	99
16.1 A demonstration of MongoDB storing data as object-of-object.	191
17.1 A demonstrative example of how MongoDB stores data as (nested) object.	205
17.2 A demonstration of MongoDB Atlas dashboard.	230
17.3 A demonstration of MongoDB Atlas dashboard where the databases and collections under “Project 0”, “Cluster0” are browsed. Database “ <code>sample_analytics</code> ”, collection “ <code>customers</code> ” is selected. There are 500 documents under this collection.	230
17.4 Atlas search set up search index using the dashboard.	232
17.5 Atlas search test.	233
18.1 A commonly seen architecture of implementing Redis.	240
19.1 System architectures of PC, VM and container.	243
19.2 PC implementation: a cook in a kitchen.	243
19.3 VM implementation: many cooks in a kitchen, each with a different cookbook.	244
19.4 Container implementation: one cook in a kitchen, handling multiple dishes, each has a cookbook and stays in its own pan.	245
19.5 An example of running <code>apline</code> container, with interactive TTY and name <code>test-apline</code>	251
19.6 List the running container <code>test-apline</code>	251
19.7 List the exited container <code>test-apline</code>	251
19.8 A demonstration of how Dockerfile, image and container link to each other.	263
19.9 A demonstration of docker image layer structure.	264
19.10 Portainer login page to create admin user.	271
19.11 Portainer dashboard overview of docker servers.	271
19.12 Portainer dashboard overview in a docker server.	272
19.13 Portainer dashboard list down of all running containers.	272
20.1 Kubernetes cluster and its key components.	274
20.2 The NodePort networking service.	279
20.3 An example of ingress service framework.	284
20.4 An example workflow of creating a production environment with Kubernetes.	289

List of Figures

xxi

21.1 Apache HTTP server test page on RHEL.	295
22.1 Layer structure of an operating system.	303
22.2 Reference Monitor Architecture.	306
22.3 Reference Monitor Architecture [1].	308
22.4 A demonstration of access control matrix.	310
B.1 Waterfall model.	326
B.2 Agile model.	328
B.3 Pipeline.	329
B.4 CI of a new feature.	330
B.5 CI and CD.	332
B.6 <i>Actions</i> framework.	335
B.7 CI and CD.	341
E.1 Waterfall model.	352
E.2 Agile model.	354
E.3 Pipeline.	355
E.4 Integration and deployment of a new feature in a conventional manner.	356
E.5 CI and CD.	359
E.6 <i>Actions</i> framework.	361
E.7 CI and CD.	367



List of Tables

2.1	Commonly used variables in prompt.	15
2.2	Commonly used shell environment variables.	19
2.3	Shell configuration files.	20
3.1	Commonly used modes in <i>Vim</i>	25
3.2	Commonly used shortcuts to switch from normal mode to insert mode.	25
3.3	Commonly used operators related to delete/cut, change, copy and paste.	31
3.4	Commonly used motions.	32
4.1	Introduction to commonly used directories in Linux file hierarchy standard.	44
4.2	Commonly used commands to navigate in the Linux file system.	45
4.3	Commonly used arguments and their effects for <i>ls</i> command.	47
4.4	Commonly used arguments and their effects for <i>mv</i> and <i>cp</i> commands.	48
4.5	Commonly used arguments and their effects for <i>rm</i> command.	49
4.6	Commonly used wildcard characters.	49
4.7	Three types of permissions.	50
4.8	Commonly used file archive tools.	56
6.1	Some attributes of a PCB.	66
9.1	Commonly seen terminologies regarding service control.	86
10.1	Different file status in a <i>Git</i> managed project.	92
11.1	An example of a relational database table.	105
11.2	A second database table in the example.	105
11.3	Widely used SQL data types.	108
11.4	Widely used SQL keywords (part 1: names).	109
11.5	Widely used SQL keywords (part 2: actions).	109
11.6	Widely used SQL keywords (part 3: queries).	110
11.7	Commonly used constraints.	113
12.1	An estimation of DBMS hardware requirements.	134

13.1 Widely used psql commands.	145
14.1 An example of a relational database table.	149
14.2 A second database table in the example.	149
14.3 Widely used SQL data types.	152
14.4 Widely used SQL keywords (part 1: names).	153
14.5 Widely used SQL keywords (part 2: actions).	153
14.6 Widely used SQL keywords (part 3: queries).	154
14.7 Commonly used constraints.	157
15.1 An estimation of DBMS hardware requirements.	178
16.1 Widely used psql commands.	189
16.2 MongoDB basic commands.	192
16.3 MongoDB basic query operators.	196
16.4 MongoDB basic update operators..	196
17.1 MongoDB basic commands.	208
17.2 MongoDB basic query operators.	211
17.3 MongoDB basic update operators..	212
17.4 MongoDB arithmetic aggregation functions.	227
19.1 Commonly used docker commands to launch a container. . .	253
19.2 Critical keywords used in a Dockerfile.	266
20.1 Commonly used Kubernetes object types.	278
23.1 DB security risks categories and associated security methods.	314
B.1 Roles in Agile model.	328
E.1 Roles in Agile model.	354



— | — | —

Part I

Linux Basics

— | — | —



1

Brief Introduction to Linux

CONTENTS

1.1	Linux as an Operating System	5
1.2	A Brief History of Linux	6
1.3	Linux Distributions	7
1.3.1	Red-Hat-Based Distributions	8
1.3.2	Debian-Based Distributions	8
1.4	Linux Graphical Desktop	9
1.5	Linux Installation	9

This chapter gives a brief introduction to Linux, including its key features, advantages and disadvantages over other operating systems (OS).

1.1 Linux as an Operating System

Linux is an OS. An OS is essentially a special piece of software running on a machine (desktop, laptop, server, mobile devices, edge device, etc.) that manages hardware resources and supports application software in the system. An OS shall be able to:

- Detect and prepare hardware
- Manage process
- Manage memory
- Manage storage and files
- Provide user and application interface, and associated authentication methods
- Provide software development kits (SDK) for developing applications

Linux has been overwhelmingly successful and adopted in many areas.

For example, Android operating system for mobile phones is developed using Linux. Google Chrome is also backed by Linux. Many websites such as Facebook are also running on Linux servers.

Some of the most favorable features of Linux, especially to large-size enterprises, are as follows.

- Clustering. It is possible to group multiple Linux machines and let them work as a whole. The group of machines appears to be a single powerful machine to the upper layer.
- Visualization. It is possible to share a server among multiple users and applications in a logically separated manner, so that each of the users thinks that he is working on a dedicated machine.
- Cloud computing. Cloud computing is an advanced usage of Linux clustering and virtualization features. Linux servers can be configured flexibly to support cloud computing functions. It is convenient to manage and audit the users and the resources they deploy.
- Real-time computing and edge computing. Embedded Linux can be implemented on micro-controllers or micro-computers for real-time edge control.

This list can go on and on.

Linux differs from Microsoft Windows and MacOS in many ways, though they are all very successful OSs. Among the three OSs, Linux is the only one that is completely open-source, in the sense that its source code can be viewed and customized by the users per requested.

1.2 A Brief History of Linux

The initial motivation of Linux is to create a UNIX-like operating system that can be freely distributed in the community.

Many modern OSs including MacOS and Linux are inspired by UNIX. UNIX operating system was created by AT&T in 1969 as a software development environment that it used internally. In 1973, UNIX was rewritten in C language, thus gaining useful features such as portability. Today, C is still the primary language used to create UNIX and Linux kernels.

AT&T, who originally owned UNIX, tried to make money from it. Back then AT&T was restricted from selling computers per required by the government. Therefore, AT&T decided to license UNIX source code to universities for a nominal fee. Researchers from universities started learning and improving UNIX, which speeded up the development of UNIX. In 1976, UNIX V6 became the first UNIX that was widely spread. UNIX V6 was developed at

UC Berkeley and was named the Berkeley Software Distribution (BSD) of UNIX.

From then on, UNIX moved towards two separated directions. While BSD remained “open”, AT&T started steering UNIX towards commercialization. By 1984 AT&T was pretty ready to start selling commercialized UNIX, namely “AT&T: UNIX System Laboratories (USL)”. USL did not sell very well. As AT&T was not allowed to sell PCs, the only thing it could do was to license the source code to other PC manufacturers. For this reason the price for the source code had to be set high as it was targeting PC manufacturers, not end users. This effectively prevented an end user from procuring UNIX source code from AT&T directly. The PC manufacturers were more profitable than AT&T just by selling UNIX-based PCs and workstations to the end users. Overall, although the community acknowledged that UNIX was useful, UNIX source code was extremely costly and was not popular among the end users.

In 1984, Richard Stallman started the GNU project as part of the Free Software Foundation. It is recursively named by phrase “GNU is Not UNIX”, intended to become a recording of the entire UNIX that could be open and freely distributed. The community started to “recreate” UNIX based on the defined interface protocols published by AT&T.

Linus Torvalds started creating his version of UNIX, i.e. Linux, in 1991. He managed to publish the first version of the Linux kernel on August 25, 1991, which only worked on a 386 processor. Later in October, Linux 0.0.2 was released with many parts of the code rewritten in C language, making it more suitable for cross-platform usage. This Linux kernel was the last and the most important piece of code to complete a UNIX-like system under GNU General Public License (GPL). It is so important that people call this operating system “Linux OS” instead of “GNU OS”, although GNU is the host of the project and Linux kernel is just a part (the most important part) of it.

1.3 Linux Distributions

As casual Linux users, people do not want to understand and compile the Linux source code to use Linux. In response to this need, different Linux distributions have emerged. They share the same Linux OS kernel but differ from each other in many ways such as software management tools and user interfaces.

Today, there are hundreds of Linux distributions in the community. The most famous two categories of distributions are as follows. The major difference is the way they manage software applications.

- Red-Hat-Based Distributions
 - Red Hat Enterprise Linux (RHEL)

- Fedora
- CentOS
- Debian-Based Distributions
 - Debian
 - Ubuntu
 - Linux Mint
 - Elementary OS
 - Raspberry Pi OS

Notice that although the source code of all the distributions above is publicly available as required by GPL license (GPL requires that any modified versions of a GPL-licensed product shall also be made open-source with a GPL license, as long as the modifications spread in the community), some of the distributions may come with a “subscription fee”. The subscription fee is not for the OS source code, but for the technical support, paid maintenance, and other add-on services that the developers of the distributions provide to the end users.

1.3.1 Red-Hat-Based Distributions

Red Hat created the Red Hat Package Manager (RPM) to manage software applications. The RPM packaging contains not only the software files but also its metadata, including version tracking, the creator, the configuration files, etc. In the OS, a local RPM database is used to track all software on the machine. Yellow Dog Updater Modified (YUM) is an open-source Linux package management application that uses RPM plus additional features for enhanced user experience. YUM is very popular among Red-Hat-based distributions.

Red Hat Enterprise Linux (RHEL) is a commercial, stable and well-supported OS that can host mission-critical applications for big business and governments. To use RHEL, customers pay for subscriptions which allow them to deploy any version of RHEL as desired. Different tiers of supports are available depending on the subscriptions. Many add-on features are available for the customers such as the cloud computing integration.

CentOS is a “recreation” version of RHEL using freely available RHEL source code. In this sense, CentOS experience should be very similar with RHEL and it is free of charge, but the users will not enjoy the professional technical support from RHEL engineers. Recently, Red Hat took over the development of CentOS project.

Fedora is a free, cutting-edge Linux distribution sponsored by Red Hat. It is less stable than RHEL, and plays as the “testbed” for Red Hat to interact with the community. From this perspective, Fedora is very similar to RHEL, just with more dynamics and uncertainties. Some functions, especially server related functions, will be tested on Fedora before implemented on RHEL.

1.3.2 Debian-Based Distributions

Different from Red-Hat-based distributions that use RPM, Debian and Debian-based distributions use Advanced Packaging Tool (APT) to manage software applications. APT simplifies the process of managing software by automating the retrieval, configuration and installation of software packages. Among all the Debian-based distributions, Ubuntu is the most successful and popular one. Ubuntu has a variety of graphical tools and focuses on full-featured desktop system while still offering popular server packages. It has a very active community to support its development.

Ubuntu has larger software pool than Fedora. Ubuntu and its associated software usually have a longer “lifespan” than Fedora because Ubuntu servers as a stable platform while Fedora is more of a “testbed”. Ubuntu is more for casual users and beginners, while Fedora more for advanced users or developers, especially developers for RHEL.

1.4 Linux Graphical Desktop

Graphical user interface is not necessary to run Linux OS. Yet, many Linux distributions support graphical desktops to convene the end users. When installing these distributions, the user can choose whether or not to install a graphical desktop environment along with the OS. The most popular desktop environment is GNOME. There are other choices such as KDE, LXDE and Xfce desktops. GNOME and KDE are more for regular PCs while LXDE and Xfce, being light in size, more for low-power-demanding systems.

Figures 1.1, 1.2, 1.3 and 1.4 give the flavors of each desktop environment mentioned above. From the figures we can see that GNOME adopts a more Linux/MacOS style desktop environment, while KDE “Windows 7” style. LXDE and Xfce are more simple in graphics presentations and they are more for embedded systems.

It is possible to install multiple desktop environments on one machine, in which case the user can choose which desktop environment to use each time the computer is started.

1.5 Linux Installation

Linux can be installed both on a fixed hard drive or on a mobile storage such as a thumb drive. The installation of different distributions may differ. Thanks to the graphical installation tools for the popular distributions, the installations can be done fairly easily.



FIGURE 1.1
GNOME desktop environment.

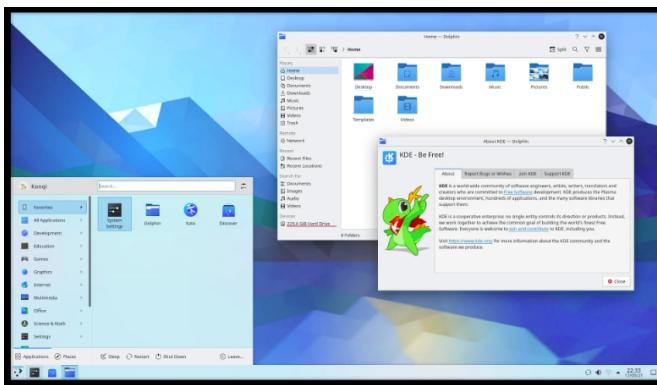


FIGURE 1.2
KDE desktop environment.

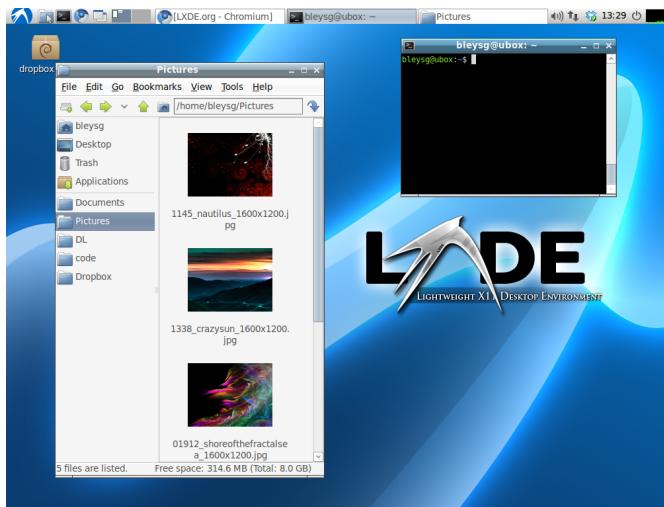


FIGURE 1.3
LXDE desktop environment.

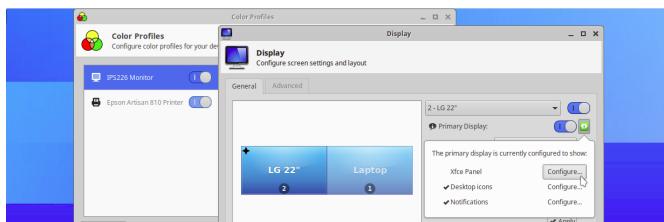


FIGURE 1.4
Xfce desktop environment.

Instructions of installing Ubuntu is given by <https://ubuntu.com>. Instructions of installing Fedora is given by <https://getfedora.org>. For the use of RHEL, consult with Red Hat at <https://www.redhat.com>. Red Hat provides different types of RHEL licenses for different using purpose, including developer license, which is cheaper than a standard enterprise-level license and serves well for learning purpose.

2

Shell

CONTENTS

2.1	Shell as a Command Line Interface	13
2.1.1	Shell Types	13
2.1.2	Prompt	14
2.2	Basic Operations	15
2.2.1	Displaying User/Machine Information	16
2.2.2	Set Alias and Shortcuts	16
2.2.3	Check Manuals of a Command	17
2.2.4	Use Comments	17
2.2.5	Others	17
2.3	Shell Environment Variables	18
2.4	Customization of Shell	19
2.5	Shell Script Programming Basics	19

Linux command line interface (CLI), usually known as the “shell” (also known as “terminal”), is the most available, flexible and powerful tool for the users and programs to interact with the OS and perform certain actions.

Notice that the use of the shell is not compulsory for casual users when the graphical desktop is present. Still, it is strongly recommended that the users shall understand at least the basics of the shell since it is more flexible and powerful than the graphical desktop and hence can become handy time to time when configuring certain software.

Linux shell will be used rapidly in the remaining of this notebook.

2.1 Shell as a Command Line Interface

Linux’s default CLI, usually known as the “shell”, was invented before the graphical tools, and it has been more powerful and flexible than the graphical tools from the first day. On those machines where no graphical desktops are installed, the use of shell is critical.

2.1.1 Shell Types

There are different types of shells for Linux. The most commonly used shell is the “bash shell” which stands for “Bourne Again Shell”, derived from the “Bourne Shell” used in UNIX. Unless otherwise mentioned, the shell we refer to in the remaining of the notebook is bash shell. An example of a shell that calculates Fibonacci series is given below. Notice that in practice shell is mainly used for interaction with the OS and not for software development, data processing or numerical calculation. The example is only for demonstration.

```
#!/usr/bin/bash
n=10
function fib
{
    x=1; y=1
    i=2
    echo "$x"
    echo "$y"
    while [ $i -lt $n ]
    do
        i='expr $i + 1 '
        z='expr $x + $y '
        echo "$z"
        x=$y
        y=$z
    done
}
r='fib $n'
echo "$r"
```

Some other shells such as “C Shell” and “Korn Shell” are also popular among certain users or certain Linux distributions. For example, C Shell supports C-like shell programming, which is sometimes more convenient than the shell. In case where the Linux distribution does not have these shells pre-installed, the user can install and use these shells just like installing any other software.

2.1.2 Prompt

With a newly started shell, a string (usually containing username, hostname, current working directory, etc.) followed by either a \$ or # should appear. Following the string, the user can key in the shell commands. The string may look different on different machines. An example is given below.

```
<username>@<hostname>:~$
```

The above string is called a *prompt*, indicating the start of a user command. By default, for regular users the ending of the prompt is \$, while for

TABLE 2.1

Commonly used variables in prompt.

Variable	Description
\!	Command number of the current command in the command history.
\#	Command number of the current command in the active shell.
\\$	User prompt “\$” for a regular user, or “#” for the root user.
\w	Current working directory entire path.
\W	Current working directory base name.
\h	Host name.
\d	Current date.
\t	Current time.
\u	Username.
\s	Shell name, for example “bash”.

the root user the ending is #. The prompt can be customized by changing the environment variable PS1. See Sections 2.3 and 2.4 for details about environment variables and shell customization methods respectively. Commonly used variables in PS1 are summarized in Table 2.1.

For the term “root user”, we are referring to a special user with username and user ID (UID) “root” and 0 respectively. This UID gives him the administration privilege over the machine, such as adding/removing users, changing ownership of files, etc. To avoid fatal damage to the entire system by human error, root user shall not be used unless absolutely necessary. For this reason, the root user’s authentication is often deactivated by default (by setting its login password to invalid).

Notice that the root user is different from a “sudoer”, later of which is basically a regular user equipped with *sudo privilege*. A sudoer can temporarily switch to root user by using **sudo su** as follows.

```
<username>@<hostname>:$ sudo su
[sudo] password for <username>:
root@<hostname>:/home/<username>#
```

More about sudo privilege, **sudo** and **su** commands are introduced in Chapter 7.

Key in a command and press **Enter** to execute the command. A Linux shell command looks like the following in general

```
$ <command> [<option>] [<input>]
```

The shell comes with built-in commands supported by Linux. Different Linux distributions may support different commands. Shell can also trigger the execution of applications installed in the system. More about commonly used commands and applications will be introduced in the remaining of the notebook.

2.2 Basic Operations

Some basic operations of the shell such as setting alias to commands, adding comments, running commands in the background, etc., are introduced here. More are to be covered in the remaining chapters of this notebook.

2.2.1 Displaying User/Machine Information

When login to a new system, the first step is often to check the basic system information, such as hardware configuration, OS version, username, hostname, etc. While there are many ways to do them, each with some features different than others, some handy commands are introduced here.

The following commands show basic information of a user.

```
$ whoami
<username>
$ grep <username> /etc/passwd
<username>:x:<uid>:<gid>:<gecos>:<home-directory>:<shell>
```

In the above, `whoami` is used to display the current login user's username. Command `grep` is used to search content from files or folders, in this case the user name from `/etc/passwd` file which stores user information. This should return the username, the password (for encrypted password, an "x" is returned), UID, group id (GID), user id info (GECOS), home directory and default shell location of the user. Another command `id` also returns the UID and GID information of the current user. More about commands such as `grep` will be introduced in later chapters.

The following commands show the date and hostname of the machine.

```
$ date
<date, time and timezone>
$ hostname
<hostname>
```

The following command `lshw` lists down hardware information in details. Sudo privilege is recommended when using this command, to give more detailed and accurate information of the system. Sometimes the displayed information can be too detailed. Use `-short` argument with the command to shorten the output.

```
$ sudo lshw
```

2.2.2 Set Alias and Shortcuts

Command `alias` is used to create short-cut keys for commands and associated options, which makes it more convenient for the system operators to work on

the shell. Some alias has already been created automatically when the shell is started. Use `alias` to check the existing alias in the shell. An example is given below.

```
$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
```

A temporary alias can be added to the shell by using

```
$ alias <shortcut command>=<original command and options>
```

for example

```
$ alias pwd='pwd; ls -CF'
```

To permanently add alias to the shell, the alias needs to be added to the shell start scripts such as `~/.bashrc`.

2.2.3 Check Manuals of a Command

Many commands can be used flexibly and it is impossible to illustrate all their details. Consider use the following two methods to check the detailed manual about a command.

```
$ man <command>
$ <command> --help
```

2.2.4 Use Comments

Use `#` to lead a single-line comment. For multi-line comment, use `: ' ... '`. Examples are given below.

```
# this is a single-line comment
: '
this is a multi-line comment
this is a multi-line comment
this is a multi-line comment
,'
```

2.2.5 Others

Use `history` to check history commands. Use `!<history command index>` to repeat a history command, or use `!!` to repeat the latest previous command. It is possible to disable history recording function for privacy purpose.

Use & after a command to run it in the background.

Use ; to write two commands in a single row, and execute them sequentially with one click of Enter.

Use \ to break a single-line command into multiple rows when the command is very long.

2.3 Shell Environment Variables

To execute a command by its name, the OS needs to know where the command is located at. Commonly used commands shall be included in the PATH environment of the shell, so that they can be executed anytime from any working directory. The PATH environment is a collection of directories in the system, and it is initialized automatically when the shell is started. Check the PATH environment by

```
$ echo $PATH
<directory-1>:<directory-2>:<directory-3>: ...
```

where echo displays a line of text, and \$PATH is a built-in environment variable that records the PATH environment of the current shell.

The default PATH environment often contains the following directories.

- /bin, /usr/bin: commonly used Linux built-in commands
- /sbin, /usr/sbin: commonly used Linux built-in commands for administrators
- /home/<username>/bin: commands defined by a user

To determine the location of a particular command, use type if the command can be found in the PATH environment. Alternatively, use locate, mlocate to search all the accessible files in the system to find a command. The syntax is demonstrated below.

```
$ type <command>
<command-location>
```

In addition to PATH, there are other shell environment variables for the user to monitor and control the OS. Table 2.2 summarizes commonly used shell environment variables. Command echo \${<variable-name>} can be used to check the values of these variables. Use command env to check a list of environment variables in the shell.

The environmental variables can be created or updated as follows.

```
<variable-name> = <variable-value> ; export <variable-name>
```

For example,

TABLE 2.2

Commonly used shell environment variables.

Variable	Description
BASH	Full pathname of the <code>bash</code> command.
BASH_VERSION	Current version of the <code>bash</code> command.
EUID	Effective user ID number of the current user, which is assigned when the shell starts, based on the user's entry in <code>/etc/passwd</code> .
HISTFILE	Location of the history file.
HISTFILESIZE	Maximum number of history entries.
HISTCMD	The number index of the current command.
HOME	Home directory of the current user.
PATH	Path to available commands.
PWD	Current directory.
OLDPWD	Previous directory.
SECONDS	Number of seconds since the shell starts.
RANDOM	Generating a random number between 0 and 99999.

```
PATH = $PATH:/getstuff/bin ; export PATH
```

adds a new directory `/getstuff/bin` to the `PATH` environmental variable. This allows temporary change to the `PATH` environment variable. Notice that each time the shell is restarted, the environmental variables are also reset.

To permanently change the value of an environmental variable, consider customizing the shell. Shell customization is introduced in Section 2.4.

2.4 Customization of Shell

Shell configuration files are loaded each time a new shell starts. User-defined permanent configurations (such as useful alias) can be put into these files so that the configurations can be implemented automatically. Some useful files are summarized in Table 2.3.

To customize the shell behavior for a user, the most commonly method is to edit his `~/.bashrc` which is executed automatically each time he starts a shell.

TABLE 2.3

Shell configuration files.

File pathname	Description
<code>/etc/profile</code>	The environment information for every user, which executes upon any user logs in. Root privilege is required to edit this file.
<code>/etc/bashrc</code>	Bash configuration for every user, which executes upon any user starts a shell. Root privilege is required to edit this file.
<code>~/.bash_profile</code>	The environment information for current user, which executes upon the user logs in.
<code>~/.bashrc</code>	Bash configuration for current user, which executes upon the user starts a shell.
<code>~/.bash_logout</code>	Bash log out configuration for current user, which executes upon the user logs out or exit the last bash shell.

2.5 Shell Script Programming Basics

A truly power feature of the shell is its ability to redirect inputs/outputs of commands, thus to chain the commands together. Meta-characters pipe (!), ampersand (&), semicolon (;), dollar (\$), parenthesis (()), square bracket ([]), less than sign (<), greater than sign (>), double greater than sign (>>), error greater than sign (2>) and a few more more, are used for this feature. Details are given below.

The pipe (!) connects the output of the first command to the input of the second command. The following example searches keyword “function” in `calculate_fib.sh` which was given previously.

```
$ cat calculate_fib.sh | grep function
function fib
```

where `cat` concatenates files and print on the standard output, and `grep` prints lines that match patterns in each file.

The semicolon (;) allows inputting multiple commands in the same line in the script. The commands are then executed one after another from left to right.

The ampersand (&) can be put in the end of a line so that the command on that line will run in the background. The commands or process running in the background does not occupy the shell standard display, and the users can continue working on other commands in parallel. This is particularly useful when a task is going to take a long time to be executed. To manage the tasks running in the background, check more details in Chapter 6.

Use the dollar sign \$ (not the prompt) to indicate a command expansion.

The command in \$(<command>) will be executed as a whole, then treated as a single argument. The content in () is sometimes called sub-shell. For example, to display the function defined in `calculate_fib.sh` previously,

```
$ echo Display functions: $(cat calculate_fib.sh | grep function)
Display functions: function fib
```

Below is another example to count the number of files/folders in the current directory.

```
$ echo There are $(ls -a | wc -w) files in this directory.
There are 69 files in this directory.
```

where `wc` counts the number of lines, words or bytes in a file.

Use \$[<arithmetic expression>] for simple calculations, such as

```
$ echo 1+1=$[1+1]
1+1=2
```

The dollar sign \$ is also used to expand the value of a variable, either environmental variable or self-defined variable, as explained previously in 2.3. An example is \$PATH which returns the PATH environment.

The less than sign < and greater than sign > are used to map the input/output of a command to a file instead of the standard input and output. An example using command `sort` together with input direction < is given as follows. Considering sorting characters “a”, “c”, “b”, “g”, “e”, “f”, “d” using `sort` command. The letters are input from the console as follows. Use `ctrl+D` to quit the input, and the output after sorting will be displayed in the console as follows.

```
$ sort
a
c
b
g
e
f
d
a
b
c
d
e
f
g
```

For demonstration purpose, create a file `before_sort` in the current working directory. Inside `before_sort` are letters “a”, “c”, “b”, “g”, “e”, “f”, “d”, each occupying a separate row. There are several ways to prepare the file which will be explained shortly. For now, just assume that the file already exists. Use `cat` to quickly check its content as follows.

```
$ cat before_sort
a
c
b
g
e
f
d
```

Use **sort** to sort **before_sort** as follows. In this case, the input to **sort** becomes a file, rather than the standard input from the keyboard. Notice that in this example, **sort before_sort** also works, as **sort** will by default take its first argument as the location of the file to be sorted.

```
$ sort < before_sort
a
b
c
d
e
f
g
```

Use **>** to redirect the output of a command to a file as given in the following example.

```
$ sort < before_sort > after_sort
$ cat after_sort
a
b
c
d
e
f
g
```

where **sort** does not output the result to the console, but instead saves the result in a file named **after_sort**. The double greater sign **>>** works similarly with **>** except that **>>** will append the output to an existing file, while **>** overwrites the existing file.

With the above been said, it is possible to use the following to create the **before_sort** file that has been used in the example.

```
$ echo -e "a\nc\nb\nng\nne\nf\nnd\n" > before_sort
```

The error greater sign **2>**, **2>>** works similarly with **>**, **>>** except that instead of redirecting standard output messages, it redirects the error messages.

3

Text File Editing

CONTENTS

3.1	General Introduction to Vim	23
3.2	Vim Modes	24
3.3	Vim Profile Configuration	25
3.3.1	Mapping Shortcuts	27
3.3.2	Syntax and Color Scheme	28
3.3.3	Other Useful Setups	28
3.3.4	Plug Tools	28
3.4	Basic Operations in <i>Vim</i>	29
3.4.1	Cut, Change, Copy and Paste	30
3.4.2	Search in the Text	32
3.4.3	Other Tips	33
3.5	Visual Modes	34
3.6	Vim Macros	34
3.7	File Explorer and Screen Splitting	36
3.8	<i>NeoVim</i>	37
3.9	Other Text Editors	37

Many text editors are supported in Linux, to name a few, *Vim*, *Emacs*, *gedit*, *Visual Studio Code*. These text editors come with different features. Some of the text editors even provide integrated development environment (IDE) features.

Among the vast number of choices, *Vim* is probably the most popular one. It works perfectly in a shell environment without relying on graphical desktop, thus is adopted by many Linux distributions as the built-in text editor. *Vim* is introduced in this chapter, followed by a brief review of some other commonly used text editors.

3.1 General Introduction to Vim

Vim, which stands for “*Vi IMproved*”, is a free and open-source software developed by Bram Moolenaar et al. It is an expansion to the *Vi* text editor to

include features such as syntax highlighting, etc., and has become the default text editor of many Unix/Linux based operating systems.

Some people claim *Vim* to be the most powerful text file editor and IDE for programming on a Linux machine (and potentially on all computers and servers). The main reasons are as follows.

- *Vim* is usually built-in to Linux during the operating system installation, making it the most available and cost-effective text editor.
- *Vim* can work on machines where graphical desktop is not supported.
- *Vim* is light in size and is suitable to run even on an embedded system.
- *Vim* operations are done mostly via mode switch and shortcut keys; as a result, **the brain does not need to halt and wait for the hand to grab and move the mouse**, thus speeding up the text editing.
- *Vim* is highly flexible and can be customized according to the user's habit (for example, through `~/.vimrc`), and it allows the users to define shortcut keys.
- *Vim* can automate repetitive operations by defining macros.
- *Vim* can be integrated with third-party tools which boost its features to a higher level.

Vim can become very powerful and convenient for the user if he is very used to it. However, *Vim* is not as intuitive as other text editors such as *gedit*, and there might be a learning curve for the beginners.

Bram Moolenaar, the original inventor of *Vim*, passed away on August 3, 2023 at the age of 62. His project *Vim* has been taken over by the contributors from community.

3.2 Vim Modes

Unlike other text editors, *Vim* defines different “modes” during the operation, each mode has some unique features. For example, in the *insert* mode, *Vim* maps keyboard inputs with the text file just like an conventional text editor. In the *normal* mode (this is the default mode when opening *Vim*), *Vim* uses useful and customizable shortcut keys to quickly navigate the document and perform operations such as cut, copy, paste, replace, search, and macro functions. In the *virtual* mode, *Vim* allows the user to select a block of content for further editing. In the *cmdline* mode, *Vim* takes order from command lines and interact with Linux to perform tasks such as save and quit.

The following Table 3.1 summarizes the commonly used modes in Vim.

TABLE 3.1Commonly used modes in *Vim*.

Mode	Description
Normal	Default mode. It is used to navigate the cursor in the text, search and replace text pieces, and run basic text operations such as undo, redo, cut (delete), copy and paste.
Insert	It is used to insert keyboard inputs into the text, just like commonly used text editors today.
Visual	It is similar to normal mode but areas of text can be highlighted. Normal mode commands can be used on the highlighted text.
Cmdline	It takes in a single line command input and perform actions accordingly, such as save and quit.

TABLE 3.2

Commonly used shortcuts to switch from normal mode to insert mode.

Operator	Description
i	Insert before the character at the cursor.
I	Insert at the beginning of the row at the cursor.
a	Insert after the character at the cursor.
A	Insert at the end of the row at the cursor.
o	Create a new row below the cursor and switch to insert mode.
O	Create a new row above the cursor and switch to insert mode.

As a start, the following basic commands can be used to quickly create, edit and save a text file using vim. In home directory, start a shell and key in

```
$ vim testvim
```

to create a file named “testvim” and open the file using *Vim*. Notice that in some Linux versions, *vi* might be aliased to *vim* by default.

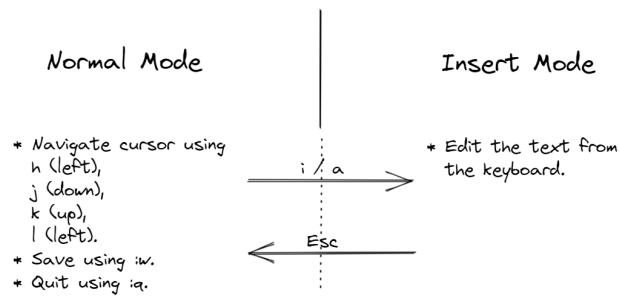
In the opened file, use **Esc** and **i/a** (insert/append) to switch between normal mode and insert mode. Notice that in the normal mode, the cursor is on a character. When using the insert command, the insertion cursor is put to the left of that character, whereas when using the append command, the insertion cursor is put to the right of that character.

In the normal mode, use **h, j, k, l** to navigate the position of the cursor.

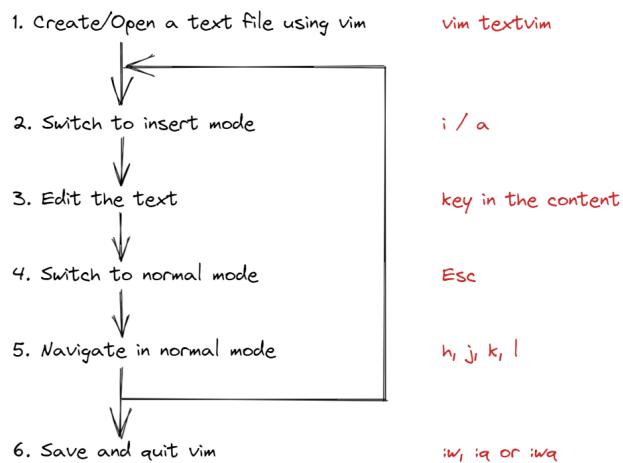
Finally, in the normal mode, use **:w** to save the file, and **:q** to quit *vim*, or use **:wq** to save and quit *Vim*.

The above basic commands and their relationships are summarized in Fig. 3.1. A flowchart to create/open, edit, save, and quit a text file using the aforementioned commands are given in Fig. 3.2.

There are other shortcuts to switch from normal mode to insert mode. Some of them are summarized in Table 3.2.

**FIGURE 3.1**

Mode switching between normal mode and insert mode, and basic functions associated with the modes.

**FIGURE 3.2**

A flowchart for simple creating, editing and saving of a text file using *Vim*.

3.3 Vim Profile Configuration

With the basic operations introduced in Section 3.2, we are able to create and edit a text file as we want to, just like using any other text editor. Though at this point the advantages of using *Vim* over other text editors are not obvious yet, the *Vim* editor is at least useable.

Before introducing more advanced features of *Vim*, for better user experience we can now customize the user profile to suit our individual habit. Notice that the customization is completely optional and personal. This section only introduces the idea and basic methods such as re-mapping keys and creating user-defined shortcuts. Everything introduced here are merely examples and it is completely up to the user how to design and implement his own profile.

In Linux, navigate to home directory. Create the following path and file `~/.vim/vimrc` or `~/.vimrc`. Open the *vimrc* file as a blank file using *Vim*. The individual user profile can be customized here.

There are many readily available *vimrc* profiles shared in the community. Feel free to use them as a reference when creating a new profile.

3.3.1 Mapping Shortcuts

It is desirable to re-map some keys to speed up the text editing. For example, by mapping `jj` to `Esc` in insert mode, one can switch from insert mode to normal mode more quickly (notice that consequent “`jj`” is rarely used in English). Another example could be mapping `j`, `k`, `i` to `h`, `j`, `k` respectively in normal and visual modes, making the navigation more intuitive. In that case, another key needs to be mapped to `i` because insert command is very useful and we do not want to lose it.

The mapping of keys and keys combinations can be done as follows in *vimrc*.

```
inoremap jj <Esc>
noremap j h
noremap k j
noremap i k
noremap h i
```

where `inoremap` is used to map keys (combinations) in insert mode, and `noremap` in normal and visual modes.

The upper case letter `S` and lower case letter `s` in normal mode are originally used to delete and substitute texts, and they are rarely used due to the more powerful shortcut `c` which does similar tasks. We can re-map `S` to saving, and disable `s`. Similarly, upper case letter `Q` is mapped to quitting *Vim*.

```
noremap s <nop>
map S :w<CR>
map Q :q<CR>
```

where `<nop>` stands for “no operation” and `CR` stands for the “enter” key on the keyboard. The keyword `map` differs from `noremap` in the sense that `map` is for recursive mapping.

3.3.2 Syntax and Color Scheme

By default *Vim* displays white colored contents on a black background. Use the following command in *vimrc* to enable syntax highlighting or change color schemes. Use `:colorscheme` in cmdline mode in *Vim* to check for available color schemes.

```
syntax on
colorscheme default
```

The following setups in *vimrc* displays the row index and cursor line (a underline at cursor position) of the text, which can become handy during the programming. Furthermore, it sets auto-wrap of text when a single row is longer than the displaying screen.

```
set number
set cursorline
set wrap
```

The following command opens a “menu” when using cmdline mode, making it easier to key in commands.

```
set wildmenu
```

3.3.3 Other Useful Setups

Use `scrolloff` to make sure that when scrolling in *Vim*, there are always margins lines in the top and bottom of the screen, so that the cursor is always close to the centre of the screen.

```
set scrolloff=3
```

Enable spell check in *Vim* as follows.

```
map sc :set spell!<CR>
```

where `sc` can be used to quickly turn on and off the spell check function. In addition, when the cursor is put on the wrongly spelled word, use `z=` to open a list of possible corrections.

3.3.4 Plug Tools

In the Linux community, many plug tools have been created to add useful features for *Vim*. As a demonstration, in this section *vim-plug*, a light-size vim plugin management tool created on GitHub, is used to install selected *Vim*

plugins. Details about *vim-plug* can be found at GitHub under *junegunn/vim-plug*.

Following the instructions given by GitHub under *junegunn/vim-plug*, to use *vim-plug* on Linux, the very first step is to use *cURL*, a command-line tool for transferring data specified with URL syntax, to download *vim-plug*. To install *cURL* if it has not been installed, use

```
$ sudo yum install curl # red-hat-based distributions  
$ sudo apt install curl # debian-based distributions
```

With *cURL* installed, use the following in the shell to install *vim-plug*

```
$ curl -fLo ~/.vim/autoload/plug.vim --create-dirs \  
      https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim
```

In the very beginning of *vimrc*, add the following to specify the plugins to be installed. As an example, *vim-airline/vim-airline* and *joshdick/onedark.vim* are to be installed, the first of which adds a status line at the bottom of the *Vim* window, and the second adds a popular color scheme “onedark”.

```
call plug#begin()  
Plug 'vim-airline/vim-airline'  
Plug 'joshdick/onedark.vim'  
call plug#end()
```

Finally, reload *vimrc*, then run `:PlugInstall` in cmdline mode to install the plugins. Use `colorscheme onedark` instead of `colorscheme default` in *vimrc* to enjoy the onedark color scheme.

Notice that instead of setting up configurations permanently in *vimrc*, the user can also apply a setup in cmdline mode for temporary use in an open session. A full list of *vimrc* configurations used in this chapter can be found in the appendix.

3.4 Basic Operations in Vim

In normal mode, the most frequently used operation is probably `u` for undo. Other commonly used operations, such as delete, cut, copy, paste, replace and search, are mostly done in normal mode through shortcut keys. For example, `dd` deletes (cuts) the entire row at the cursor and `p` pastes the content in the clipboard to the cursor position. For beginners, remembering shortcut keys can be difficult. Therefore, it is suggested looking for the consistent patterns of the different commands, rather than brute-force remembering all the operations.

Many *Vim* shortcut keys in normal mode has the following pattern: an operator command directly followed by a motion command without space in between.

```
<operator><motion>
```

The operator command tells *Vim* what to do (say, copy), and the motion command tells the applicable range of the operation (say, a row, or a word, or a character). Some operator commands may work alone without motion commands.

3.4.1 Cut, Change, Copy and Paste

The following lines taken from Wikipedia under “William Shakespeare” is used as an example to demonstrate delete/cut, change, copy and paste functions. In the text file, each sentence takes a separate row as shown in Figure 3.3.

William Shakespeare (bapt. 26 April 1564 – 23 April 1616) was an English playwright, poet and actor, widely regarded as the greatest writer in the English language and the world's greatest dramatist.
 He is often called England's national poet and the “Bard of Avon” (or simply “the Bard”).

1 William Shakespeare (bapt. 26 April 1564 – 23 April 1616) was an English playwright, poet and actor, widely regarded as the greatest writer in the English language and the world's greatest dramatist.
2 He is often called England's national poet and the "Bard of Avon" (or simply "the Bard").

FIGURE 3.3

A piece of text of “William Shakespeare”, for demonstration.

Use either **x** or **X** to delete the character at the cursor or previous to the cursor, respectively. To delete multiple characters, one way is to press **x** or **X** multiple times (or hold the keys). Alternatively, it is possible for *Vim* to automatically repeat the procedure. For example, **20x** tells *Vim* to perform **x** for 20 times. The same applies for other operators or motions commands. For example, **10l** executes **l** for 10 times, moving the cursor to the right for 10 characters.

Operator **d**, similar with **dd**, deletes the contents of the text, but it requires a motion command and can be used more flexibly. The motion shall tell *Vim* the applicable range to delete/cut.

For example, **d1** deletes one character to the right, i.e. deletes the character at the cursor just like **x**. Likewise, **dh** deletes one character to the left just like **X**. Similarly, **d20l** deletes 20 characters to the right, where “**20l**” as a whole plays as the motion of “20 characters to the right”. A combination by using things like **5d4l** also works and leads to the same result as $20 = 5 \times 4$.

Command **d** can be used even more flexibly. For example, by using word-related motions, **d** can delete/cut by words instead of by characters. Move the cursor to the beginning of a word, (for example, “S” in “Shakespeare”), use **dw** to delete the word. The word motion **w** is similar with **l**, except that **l**

TABLE 3.3

Commonly used operators related to delete/cut, change, copy and paste.

Operator	Description
x	Delete/Cut the character at cursor.
X	Delete/Cut the character before cursor.
dd	Delete/Cut the entire row.
d	Delete/Cut selected text according to the motion command.
cc	Change the entire row.
c	Change selected text according to the motion command.
yy	Copy the entire row.
y	Copy selected text according to the motion command.
p	Paste clipboard to the cursor.

directs to the next character, while **w** directs to the beginning of next word. Similarly, **b** directs to the beginning of the current/previous word. Thus, **db** can be used to delete word to the left. Examples **d10b**, **10db**, **d20w**, **5d4w** can be used to delete multiple words at a time. Motions **w** and **b** can also be used to navigate in the text just like **l** and **h**.

When in the middle of a word, **dw** will delete the characters from the cursor to the beginning character of the next word. For example, if the cursor is currently at “k” in “Shakespeare”, **dw** will delete “kespeare ” (notice that the space between “Shakespeare” and “(bapt.” will also be deleted). To delete from the beginning of the word instead, you can use **b** first to navigate back to the beginning of the word, then apply **dw**. Alternatively, use “inner-word” motion **iw** to indicate that the whole word of the cursor shall be deleted, i.e., **diw** to delete “Shakespeare”. The space after “Shakespeare” will stay. To remove that space together with the word “Shakespeare”, use **daw** instead. More about motions **aw** will be introduced shortly.

In addition to character-related motions **h**, **l** and word-related motions **b**, **w**, there are similar motions for sentence ((previous),) (next) and paragraph { (previous), } (next). There are also inner-sentence motion **is**, inner-paragraph motion **ip**, inner-quotation motion **i'**, **i"**, **i‘** and inner-block motion **i(**, **i<**, **i{**, and many more. For example, when the cursor is at “A” of “26 April 1564”, **di(** will delete everything inside “()”, i.e. deleting “bapt. 26 April 1564 - 23 April 1616”.

The operators and motions introduced so far are summarized in Tables 3.3 and 3.4. Notice that motions **aw**, **as**, **ap** are also given in the table. They are similar with their corresponding **iw**, **is**, **ip** except that when deleting, the sequential blank space (for word and sentence) or blank row (for paragraph) will also be deleted.

To change contents, use operator **c**, which works the same way as **d** but it automatically switch to insert mode after removing the content. To copy a piece of text to clipboard, use **y** (stands for “yank”) followed by its associated

TABLE 3.4

Commonly used motions.

Motion	Description
h, l	One character to the left or right.
j, k	One row to the up or down.
b	First character of the current word, or fist character of the previous word.
e	Last character of the current word.
w	First character of the next word.
(,)	One sentence to the previous or next.
{, }	One paragraph to the previous or next.
iw, is, ip	Inner-word, inner-sentence, inner-paragraph.
aw, as, ap	A word, a sentence, a paragraph (including the end blank).
i', i", i'	Inner-quotation for different types of quotations.
i(, i<, i[,]	inner-block for different types of brackets.
0 (zero)	Beginning of the row.
\$	Ending of the row.
gg	Beginning of the text.
G	Beginning of the last row of the text.

motion to indicate the range of text. The motions also follow Table 3.4. To paste the text in the clipboard to the cursor, use p. No motion is required.

In addition to Table 3.4, another commonly used type of motion is to “find by character”. For example, consider the following row of text. The cursor is currently at letter “A”.

```
ABCDEFG;HIJKLMNOP;OPQ;RST;UVW;XYZ
```

In normal mode, using f followed by a character will navigate the cursor to the nearest corresponding character that appears in the text. For example, fG will move the cursor to letter “G”. Similarly, f; will move the cursor to the “;” between “G” and “H”. Key in f; again and the cursor will move to “;” between “N” and “O”. From here key in 2f; and the cursor will go to “;” between “T” and “U”, as it is equivalent to executing f; twice. If df; is used when the cursor is at letter “A”, “ABCDEFG;” will be deleted.

3.4.2 Search in the Text

In normal mode, use /<content> to search a keyword or a phrase. The following customization in *vimrc* shall give a better searching experience.

When searching in the text for a particular word or phrase (searching in the text will be covered in a later of the chapter), to make the searching result highlighted, add the following line to the user profile *vimrc*.

```
set hlsearch
```

```
exec "nohlsearch"
set incsearch
noremap <Space> :nohlsearch<CR>
set ignorecase
noremap = nzz
noremap - Nzz
```

where `hlsearch` enables highlighting all matching results in the text, and `incsearch` enables highlighting texts along with typing the keyword. *Vim* remembers the keyword from the previous search and may automatically highlight them in the text on a new session. This can be confusing sometimes. To tackle the issue, use command `exec "nohlsearch"` (`exec` in *vimrc* executes a command when starting a new session) after `set hlsearch` to force *Vim* to clear its searching memory on a new session. Finally, to quit searching, use `:nohlsearch` in cmdline mode and the highlights shall be gone. For convenience, consider mapping it with a customized shortcut key as well, for example to `Space`. As a bonus, set `ignorecase` to ignore case-sensitive during the searching.

Keys `n` and `N` is used to navigate through the searching result, and `zz` is used to pin cursor position in the central of the screen. They are mapped to `-` and `=` in *vimrc*. Notice that they can also be used as motion together with delete/cut, change and copy as given in Table 3.3.

With the above setup, searching for “he” using `/he` leads to the following result given in Fig. 3.4. From Fig. 3.4, it can be seen that all appearances of

The screenshot shows a terminal window with the text of William Shakespeare's biography. The word 'he' is highlighted in yellow across several instances. The cursor is at the end of the first 'he' in the sentence 'the greatest writer'. The status bar at the bottom shows '/he'.

```
1 William Shakespeare (bapt. 26 April 1564 – 23 April 1616) was an English pla
2 ywright, poet and actor, widely regarded as the greatest writer in the English language and the world's greatest dramatist.
3 He is often called England's national poet and the "Bard of Avon" (or simply "the Bard").
```

FIGURE 3.4

Search “he” in the piece of text of “William Shakespeare”.

“he” (case-insensitive) is highlighted, and the cursor is automatically moved to its first appearance, i.e. “he” in “the greatest writer”. Click `Enter` to enable free move of the cursor.

3.4.3 Other Tips

Use `Ctrl+o`, `Ctrl+i` to “undo” and “redo” cursor positions, respectively. They only move the cursor position and don’t change the actual texts.

To save as a file, use `:w <new path>` in cmdline mode.

In cmdline mode, use `!` to interact with the shell. For example, consider a case where a read-only file needs to be edited and saved by a sudoer who forgot to start *Vim* using `sudo`. The common `:w` will be rejected. In this case, use `:w !sudo tee %` to perform the save, where `tee` is a Linux command

that takes standard input and writes to a file, and % stands for the current file. In another example where an existing file's content is to be insert into the current text, navigate the cursor to the place to insert the text, and use :r !cat <filename>.

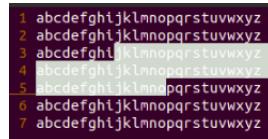
To convert and save the current text into an *html* file, use :%TOhtml. From there, the file can be further converted into a PDF file.

3.5 Visual Modes

The use of a mouse makes selecting a block of text very intuitive. In most text editors, the selected text will be highlighted, as if the cursor expands from one character to the entire block of text. Sequentially, operations such as delete and copy can be performed on the selected text.

The three visual modes of *Vim*, namely “visual”, “visual-line” and “visual-block”, provide similar experience where the user can select and highlight a block of text.

Use v to enter the visual mode, then navigate the cursor to select a block of text. This allows the user to select text between any two characters. An example is given by Fig. 3.5. Alternatively, use V to enter the visual-line mode where multiple lines can be easily selected, and use <**ctrl**>+v to enter visual-block mode to select a rectangular block of text.



A screenshot of a terminal window showing Vim in visual mode. The text consists of seven lines, each starting with a number from 1 to 7. Lines 3 through 6 are highlighted in yellow, indicating they are selected. The text content is:

```

1 abcdefghijklmnopqrstuvwxyz
2 abcdefghijklmnopqrstuvwxyz
3 abcdefghiijklmnopqrstuvwxyz
4 abcdefghiijklmnopqrstuvwxyz
5 abcdefghiijklmnopqrstuvwxyz
6 abcdefghiijklmnopqrstuvwxyz
7 abcdefghiijklmnopqrstuvwxyz

```

FIGURE 3.5

An example of visual mode where a block of text is selected.

In any of the above visual mode, use :normal + <operation> to execute operation(s) form the normal mode for each line. This allows convenient editing of multiple lines of text all together. For example, use V to select a few lines of contents, followed by :normal Oiprefix-. This will insert **prefix-** to all the lines, as if Oiprefix- is executed in normal mode to all the lines separately. Notice that if i has been mapped by another character in normal mode, use that character instead.

Alternatively, in visual-block mode after selecting a block of content, use I to enter insert mode and insert content in the first row of the selected block. When exiting the insert mode using <Esc>, the changes will apply to all selected rows. Similar effect can be achieved.

3.6 Vim Macros

Vim macros is a convenient and power tool for repetitive works. Use `q` in normal mode to start and end a macro recording. The syntax follows

```
q<macro-name><operations>q
```

where `<macro-name>` is a single character that labels the macro.

Consider the following example where there is a text file containing the following content

```
apple.jpg  
pear.jpg  
orange.jpg  
banana.jpg  
peach.jpg
```

and we would like to change the content to

```
image: 'apple.jpg'  
ttl: 5  
image: 'pear.jpg'  
ttl: 5  
image: 'orange.jpg'  
ttl: 5  
image: 'banana.jpg'  
ttl: 5  
image: 'peach.jpg'  
ttl: 5
```

In this example, repetitive work is involved and it is time consuming to do it manually if there are thousands of items in the file, and it is better to record a macro to automate the procedure. Navigate the cursor to the first row of the text, and type the following sequence of characters.

```
q<macro name>Oimage: '<Esc>A'<Enter>ttl: 5<Esc>jq
```

where `<macro name>` can be any character, for example `s`. The string in the middle `Oimage: '<Esc>A'<Enter>ttl: 5<Esc>j` is the necessary procedure to perform the revision for one row. If everything is done correctly, the file should appear as

```
image: 'apple.jpg'  
ttl: 5  
pear.jpg  
orange.jpg  
banana.jpg  
peach.jpg
```

and the cursor should be at row `pear.jpg`.

To repeat the recorded procedures, use `@<macro-name>`. In this example, just key in `@s` in the normal mode, and the file should appear as

```
image: 'apple.jpg'
ttl: 5
image: 'pear.jpg'
ttl: 5
orange.jpg
banana.jpg
peach.jpg
```

Repetitively using @s proceeds with the revision. In the case the procedure needs to be repeated for many times, use <number>@<macro-name>, in this example 3@s, to complete the remaining task.

3.7 File Explorer and Screen Splitting

Many IDEs come with project folder navigation and screen splitting features. In these IDEs, there is often a built-in file explorer, from where the user can navigate in the file system, select a file to edit, and the IDE will split the window for the selected file. *Vim* has similar features that supports file explorer and screen splitting, either via built-in functions or third-party support tools.

In *Vim*, use :Explore, :Sexplore or :Vexplore (or :Ex, :Sex, :Vex for short) in cmdline mode to open a file explorer, from where the user can navigate the cursor to select a file. *Vim* will then open the file in a split window that allows the user to further editing the file.

There are many third-party plugin tools that enable convenient file explorer functions. For example, github.com/scrooloose/nerdtree. More details can be found in the associated repository on *GitHub*.

Use :split and :vsplit for horizontal and vertical screen splitting, respectively. A second split window would show up with the same text file opened. For simplicity, these commands can be mapped in *vimrc* as follows.

```
noremap sj :set nosplitright<CR>:vsplit<CR>
noremap sl :set splitright<CR>:vsplit<CR>
noremap si :set nosplitbelow<CR>:split<CR>
noremap sk :set splitbelow<CR>:split<CR>
```

where **splitright** and **splitbelow** is used to setup the default cursor position after splitting the screen.

In a split window, open a new file using :e <path>. To navigate the cursor across different split windows, use **Ctrl+w** followed by **h**, **j**, **k** and **l**. For simplicity, they can be mapped as follows.

```
noremap <C-j> <C-w>h
noremap <C-l> <C-w>l
noremap <C-i> <C-w>k
noremap <C-k> <C-w>j
```

where $<C->$ stands for $\text{Ctrl}+$.

Resize the selected split window using `:res+<number>`, `:res-<number>`, `:vertical resize+<number>`, `:vertical resize-<number>`. For simplicity, map these commands as follows.

```
noremap J :vertical resize-2<CR>
noremap L :vertical resize+2<CR>
noremap I :res+2<CR>
noremap K :res-2<CR>
```

3.8 NeoVim

Vim being an open-source project allows other to fork and build new projects on top of it. *NeoVim* is one of those projects. Comparing with *Vim* whose code is almost all from Bram Moolenaar, *NeoVim* is more of a community-driven project with diversified contributors.

A potential problem with project codes coming from a single contributor is that the code is often a bit more messy than if it were written and cross-checked by multiple contributors, and that has been the main criticism *Vim* has received. *NeoVim*, on the other hand, has cleaner code base.

It seems that *NeoVim* is a more cutting-edge version of *Vim* in the sense that new features usually come sooner in *NeoVim* than in *Vim*. *NeoVim* also has a better and more native support for Lua language. It seems that the entire configuration file for *NeoVim*, i.e., the counterpart of *vimrc*, can be built by Lua files. *Vim* is initially a single-thread application, which is fine when it is used as a text editor. However, it limits the ability of *Vim* calling other shell services. The user needs to wait for the shell services to end before he can start editing the document again. *NeoVim*, on the other hand, uses multi-thread framework, hence does not pose this issue. Later on *Vim* added the support to allow plug-ins to trigger multi-thread processes.

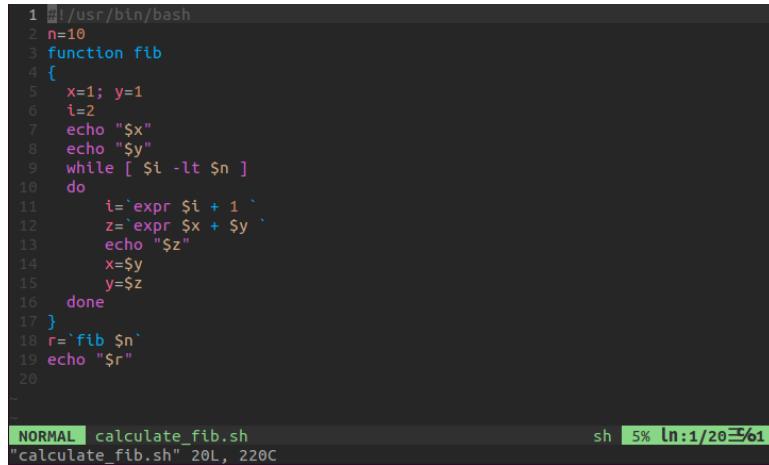
The drawback of *NeoVim* is that it being a younger and more collaborative project seems to be less stable than *Vim*.

Both *Vim* and *NeoVim* are in-development projects, and new commits are being update every day.

3.9 Other Text Editors

Apart from *Vim*, many other text editors are also widely used in Linux, each with different features. For demonstration purpose, *Vim* and other text editors

are used to open a shell script that calculates the first 10 elements of Fibonacci series.

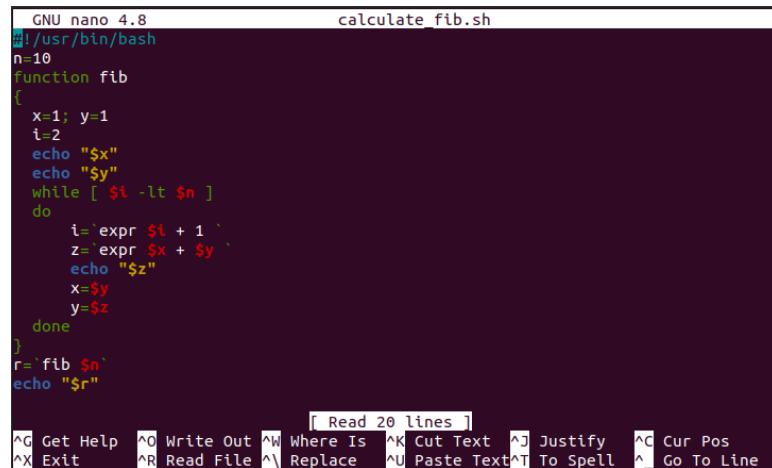


The screenshot shows the Vim editor with a shell script named `calculate_fib.sh`. The script defines a function `fib` that calculates the nth Fibonacci number using a loop and `expr` command. The status bar at the bottom indicates the file name, current mode (NORMAL), line number (ln:1), and column number (20C).

```
1 #!/usr/bin/bash
2 n=10
3 function fib
4 {
5     x=1; y=1
6     i=2
7     echo "$x"
8     echo "$y"
9     while [ $i -lt $n ]
10    do
11        i=`expr $i + 1 `
12        z=`expr $x + $y `
13        echo "$z"
14        x=$y
15        y=$z
16    done
17 }
18 r=`fib $n`
19 echo "$r"
20
```

FIGURE 3.6

Vim (with user's profile customization as introduced in this chapter).

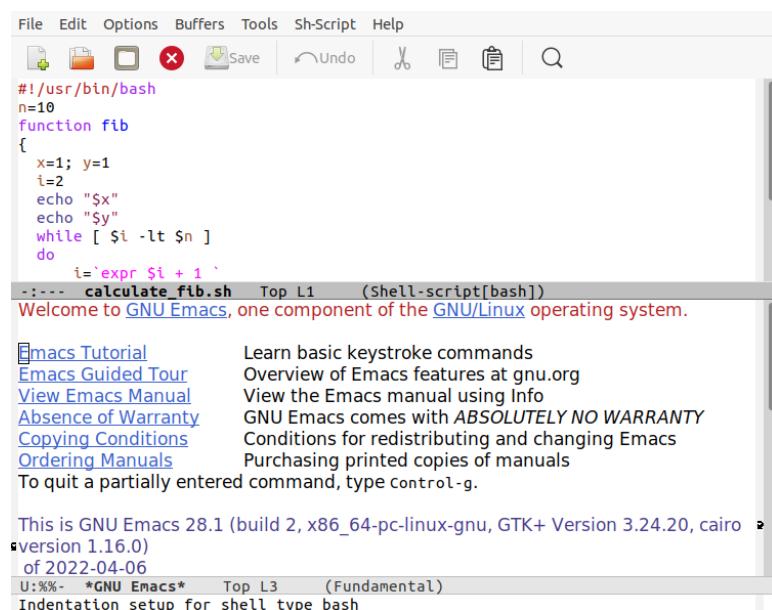


```

GNU nano 4.8                               calculate_fib.sh
#!/usr/bin/bash
n=10
function fib
{
    x=1; y=1
    i=2
    echo "$x"
    echo "$y"
    while [ $i -lt $n ]
    do
        i=`expr $i + 1 `
        z=`expr $x + $y `
        echo "$z"
        x=$y
        y=$z
    done
}
r=`fib $n`
echo "$r"

[ Read 20 lines ]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit      ^R Read File ^L Replace   ^U Paste Text ^T To Spell ^ ^ Go To Line

```

FIGURE 3.7*Nano.*


File Edit Options Buffers Tools Sh-Script Help

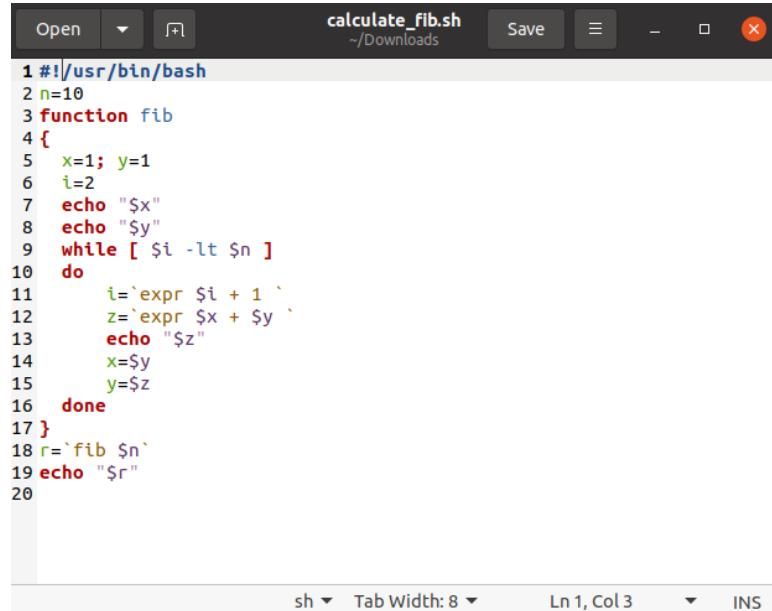
#! /usr/bin/bash
n=10
function fib
{
 x=1; y=1
 i=2
 echo "\$x"
 echo "\$y"
 while [\$i -lt \$n]
 do
 i=`expr \$i + 1 `
 ... calculate_fib.sh Top L1 (Shell-script[bash])
Welcome to [GNU Emacs](#), one component of the [GNU/Linux](#) operating system.

[Emacs Tutorial](#) Learn basic keystroke commands
[Emacs Guided Tour](#) Overview of Emacs features at gnu.org
[View Emacs Manual](#) View the Emacs manual using Info
[Absence of Warranty](#) GNU Emacs comes with ABSOLUTELY NO WARRANTY
[Copying Conditions](#) Conditions for redistributing and changing Emacs
[Ordering Manuals](#) Purchasing printed copies of manuals

To quit a partially entered command, type control-g.

This is GNU Emacs 28.1 (build 2, x86_64-pc-linux-gnu, GTK+ Version 3.24.20, cairo 1.16.0) of 2022-04-06
U:%%- *GNU Emacs* Top L3 (Fundamental)
Indentation setup for shell type bash

FIGURE 3.8*Emacs.*

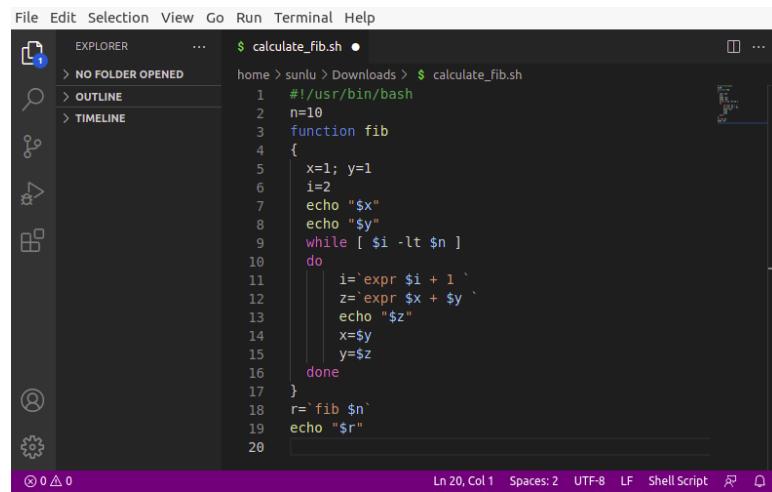


```

1 #!/usr/bin/bash
2 n=10
3 function fib
4 {
5   x=1; y=1
6   i=2
7   echo "$x"
8   echo "$y"
9   while [ $i -lt $n ]
10  do
11    i=`expr $i + 1 `
12    z=`expr $x + $y `
13    echo "$z"
14    x=$y
15    y=$z
16  done
17 }
18 r=`fib $n`
19 echo "$r"
20

```

The screenshot shows a Gedit window with a shell script named "calculate_fib.sh" open. The script calculates the nth Fibonacci number using a loop and arithmetic expansion. The code is color-coded for readability.

FIGURE 3.9*Gedit.*


```

File Edit Selection View Go Run Terminal Help
EXPLORER ... $ calculate_fib.sh •
NO FOLDER OPENED
OUTLINE
TIMELINE
$ calculate_fib.sh
1 #!/usr/bin/bash
2 n=10
3 function fib
4 {
5   x=1; y=1
6   i=2
7   echo "$x"
8   echo "$y"
9   while [ $i -lt $n ]
10  do
11    i=`expr $i + 1 `
12    z=`expr $x + $y `
13    echo "$z"
14    x=$y
15    y=$z
16  done
17 }
18 r=`fib $n`
19 echo "$r"
20

```

The screenshot shows the same shell script in Visual Studio Code. The interface includes a sidebar with icons for Explorer, Search, and Timeline. The status bar at the bottom indicates the file is a "Shell Script".

FIGURE 3.10*Visual Studio Code.*

4

File Management

CONTENTS

4.1	Filesystem Hierarchy Standard	41
4.2	Commonly Used File Exploring Commands	43
4.2.1	Print Working Directory	45
4.2.2	List Information about Files and Directories	45
4.2.3	Create Files and Directories	46
4.2.4	Move, Copy-and-Paste, and Remove Files and Directories	47
4.2.5	Use of Wildcard Characters	48
4.3	Access Control List	49
4.3.1	Change Ownership and Group of a File or Directory ...	50
4.3.2	Change Permissions of a File or Directory	50
4.3.3	Change Default Permissions	51
4.4	Search through the System	52
4.4.1	Look for a Command	52
4.4.2	Looking for Files by Metadata	52
4.4.3	Looking for Files by Variety of Properties	53
4.5	File Archive	55

File management is a big portion of OS. In Linux, each device (such as a printer) is treated and managed as a file, and Linux uses a tree hierarchy to manage devices and files. This chapter introduces the filesystem hierarchy and commonly used file management commands.

4.1 Filesystem Hierarchy Standard

The root directory is denoted by a single forward slash “/”. All sub directories or files can be located by its full path, which looks like the following

```
/<directory>/<subdirectory>/.../<directory-name>  
/<directory>/<subdirectory>/.../<file-name>
```

where the first `/` in each row represents the root directory, and sequential `/` represents entering a subdirectory.

Upon Linux installation, a file hierarchy is created. A user can create new files under this hierarchy framework, but should not change the framework itself. The hierarchy is given in Fig. 4.1. Notice that different Linux distributions may differ slightly on how the architecture looks like. The “`/`” in the figure, as introduced, stands for the root directory, and “`root`” in the figure is a subdirectory under `/` whose directory name is “`root`” and it is used store root user related documents. They are two different directories.



FIGURE 4.1

An example of Linux file system hierarchy.

A regular user’s home directory is often located at `/home/<user name>`. When logging in as a regular user, his home directory is stored in the `HOME` environment and can be retrieved by `$HOME`. A shortcut to `$HOME` is given by the tilde `~` for convenience. Hence, for example `ls ~` lists down the files and directories under his home directory.

As can be seen from Fig. 4.1, the hierarchy contains quite a few pre-

determined subdirectories, each serving a different purpose. For the ease of illustration, these subdirectories are roughly categorized by functionalities and accessibility shown in Fig. 4.2.

	"Used by the OS"		"Used by all users"		"Used by a specific user"
	Administration (System) Level		All-Users Level		Individual User Level
Executable	/bin	/sbin	/usr/bin	/usr/sbin	/home/<user name>
Library	/lib		/usr/lib		/home/<user name>
Data / Program Storage	/opt	/var	/usr/local /usr/src	/usr/share	/home/<user name>
Device	/dev	/media	/mnt		
Configuration	/etc				/home/<user name>
System	/boot	/proc	/sys		
Other	/tmp	/usr	/root	/usr/tmp	

FIGURE 4.2

A rough categorization of commonly used directories in Linux file hierarchy standard.

A brief introduction to the directories are summarized in Table 4.1.

Linux file hierarchy standard differs from MS-DOS and Windows in several ways. Firstly, Linux stores all files (regardless of their physical location) under the root directory, while Windows uses drive letters such as C:\, D:\ to distinguish different hard drives. Secondly, Linux uses slash (/) to separate directory names, e.g. /home/username while Windows uses back slash (\), e.g. C:\Users\username. Lastly, Linux uses “magic numbers” to indicate file types, while Windows often uses suffixes to tell file types.

Magic numbers of a file refer to the first few bytes of a file that are unique to a particular file type, for example, PNG file is hex 89 50 4e 47. Linux compare the magic numbers of a file with an internal database to decide the file types and features. Distinguishing file types using magic numbers can be more reliable than using suffixes, though a bit less intuitive.

TABLE 4.1

Introduction to commonly used directories in Linux file hierarchy standard.

Directory	Description
/bin, /sbin	Executables used by the OS, the administrator, and the regular users.
/lib	Libraries to support /bin and /sbin.
/usr/bin, /usr/sbin	Executables used by the administrator and the regular users.
/usr/lib	Libraries to support /usr/bin and /usr/sbin.
/opt	Application software installed by OS and administrator for all users.
/var	Directories of data used by applications.
/usr/local	Application software installed by administrator for all users.
/usr/share	Architecture-independent sharable text files for applications.
/usr/src	Source files or packages managed by software manager.
/dev	Files representation of devices, such as CPU, RAM, hard disks.
/media	System mounts of removable media.
/mnt	Manual mounts of devices.
/etc	Configuration files for OS, users, and applications.
/boot	Linux bootable kernel and initial setups.
/proc	System resources information.
/sys	Linux kernel information, including a mirror of the kernel data structure.
/tmp, usr/tmp	Temporary files.
/root	Root user's home directory.
/home/<user name>	A regular user's home directory, containing executables, configurations and files specifically belong to this user.

TABLE 4.2

Commonly used commands to navigate in the Linux file system.

Command	Description
<code>pwd</code>	Print working directory.
<code>ls</code>	List the subdirectories and files (and their detail information) in a given directory.
<code>touch</code>	Create an empty file.
<code>mkdir</code>	Create an empty subdirectory.
<code>mv</code>	Move (cut-and-paste) a directory or a file; change name of a directory or a file.
<code>cp</code>	Copy-and-paste a directory or a file.
<code>rm, rmdir</code>	Remove a directory or a file (not to Trash, but just gone).
<code>chmod</code>	Change permission.
<code>chown</code>	Change ownership.

4.2 Commonly Used File Exploring Commands

Some of the most widely used file exploring and managing commands are summarized in Table 4.2. Notice that `chmod` and `chown` are administration related commands that change the accessibility of a directory or a file, and will be introduced in a later sections together with the Linux permission system. The rest commands are categorized and introduced in the following subsections.

4.2.1 Print Working Directory

As given in Table 2.2, `$PWD` is the environmental variables to store the current working directory of the shell. Therefore, to print the current working directory in the console, use command

```
$ echo $PWD
```

Alternatively, use `pwd` as follows which has the same effect.

```
$ pwd
```

4.2.2 List Information about Files and Directories

As one of the most frequently used commands, `ls` lists down information about the files and subdirectories in the selected directory, and by default sort the entries alphabetically. The syntax is given below.

```
$ ls [<option>] [<path>]
```

An example is given in Fig. 4.3. As shown in the example, command `ls`

alone shows only the name of files and subdirectories excluding hidden items and details of each item. With the additional arguments in the option field, the returns can be customized with more details displayed. For example in Fig. 4.3, the `-l` argument displays the information in long listing form, which includes the owner and access control list information. More details about files and directories access control list are given in later part of this section.

```

sunlu@SUNLu-Laptop:~$ ls
anaconda3  Documents  Dropbox  eclipse-workspace  gurobi.log  octave-workspace  Public  snap  texmf
Desktop   Downloads  eclipse  gurobi      Music       Pictures        R      Templates  Videos
sunlu@SUNLu-Laptop:~$ ls -l
total 72
drwxrwxr-x 27 sunlu sunlu 4096 Dec 21 01:54 anaconda3
drwxr-xr-x  2 sunlu sunlu 4096 Apr 30  2021 Desktop
drwxr-xr-x  3 sunlu sunlu 4096 Jun 15 10:42 Documents
drwxr-xr-x  2 sunlu sunlu 4096 Jun 15 14:37 Downloads
drwx----- 13 sunlu sunlu 4096 Jun 15 10:38 Dropbox
drwxrwxr-x  3 sunlu sunlu 4096 Dec 21 12:34 eclipse
drwxrwxr-x  3 sunlu sunlu 4096 Feb  3  2021 eclipse-workspace
drwxrwxr-x  3 sunlu sunlu 4096 Mar  1  2021 gurobi
-rw-rw-r--  1 sunlu sunlu 488 Mar  3  2021 gurobi.log
drwxr-xr-x  2 sunlu sunlu 4096 Jan 26  2021 Music
-rw-rw-r--  1 sunlu sunlu 43 May  5 16:48 octave-workspace
drwxr-xr-x  2 sunlu sunlu 4096 Mar 29 08:58 Pictures
drwxr-xr-x  2 sunlu sunlu 4096 Jan 26  2021 Public
drwxrwxr-x  3 sunlu sunlu 4096 Feb  7  2021 R
drwx----- 9 sunlu sunlu 4096 Apr  8 14:03 snap
drwxr-xr-x  2 sunlu sunlu 4096 Feb  3  2021 Templates
drwxrwxr-x  4 sunlu sunlu 4096 Feb 23  2021 texmf
drwxr-xr-x  2 sunlu sunlu 4096 Jan 26  2021 Videos
sunlu@SUNLu-Laptop:~$ ls Do*
Documents:
MATLAB

Downloads:
calculate_fib.sh

```

FIGURE 4.3

List down information of files and subdirectories in the current working directory.

More information can be found in the `ls` command manual which is accessible via `ls --help`. Some commonly used `ls` arguments are summarized in Table 4.3. It is also possible to combine the options. For example, `ls -al` aggregates the effects of using `ls -a` and `ls -l`.

Notice that some Linux distributions may come by default an alias about `ls`, which usually helps to displays the information in a clearer manner. For example, when `ls='ls --color=auto'` is used, the displayed content will be colored based on the type of the files and subdirectories.

4.2.3 Create Files and Directories

The `touch` command is used to update timestamps of a file. If the file does not exist, `touch` will create that file with empty content. Hence, it is often used to create empty files. To do so, simply use `touch` followed by the full path to the file as follows.

```
$ touch [<option>] <path>
```

TABLE 4.3Commonly used arguments and their effects for *ls* command.

Directory	Description
-a, --all	Include hidden files and subdirectories in the display, including current directory “.” and parent directory “..” in the list.
-A, --almost-all	Include hidden files and subdirectories in the display, excluding “.” and “..”.
-C, --color [=WHEN]	Colorize the output.
-l	Use a long listing format.
-s, --size	Print the allocated size of each file, in blocks.
-S	Sort the displayed content.
-t	Sort by modification time.

For example,

```
$ touch ~/test
```

will create an empty file “**test**” under the user’s home directory. If only the name of the file is given, it will by default create the file under the current working directory. Notice that if a file name starts with “.”, it will be treated as a hidden file or automatically.

touch can also be used to create multiple files in a single-line command. For example,

```
$ touch test1 test2
```

creates both **test1** and **test2** in the current working directory.

To create a file containing a single line of string, consider using **echo** command with **>** as follows. It is more convenient than using *Vim* for the same task, although also possible.

```
$ echo '<content>' > <path>
```

For example,

```
$ echo '<html><body><h1>Hello world!</h1></body></html>' > ~/test.html
```

creates a simple static HTML web page that says “Hello world!” in the home directory.

Similar with **touch**, use **mkdir** followed by the path of the directory (including directory name) to create a directory as follows.

```
$ mkdir [OPTION] <path>
```

Specifically, **-p** option of **mkdir** allows it to create nested directories along the given path if the directories do not exist.

TABLE 4.4Commonly used arguments and their effects for `mv` and `cp` commands.

Directory	Description
<code>-b</code>	Make a backup before overwrite.
<code>-u</code>	Overwrite only when source target item is newer than the target path item.
<code>-i</code>	Prompt before overwrite.
<code>-f</code>	Do not prompt before overwrite.

4.2.4 Move, Copy-and-Paste, and Remove Files and Directories

To move a file or a directory from an existing directory to another, simply use `mv` command as follows.

```
$ mv [<option>] <source> <target>
```

Different from the conventional cut-and-paste, while moving the item, it is possible to also rename the item simultaneously. For example,

```
$ mv ~/dog.png ~/Pictures/puppy.png
```

will not only move the file `dog.png` in the home directory to the subdirectory `Pictures`, but also change the file name to `puppy.png`. For this reason, `mv` can also be used to rename an item rather than moving the item, just by “move” it to the same directory but with a different name.

Some commonly used arguments of `mv` is summarized in Table 4.4, many of which concerns about the case where there is already an existing item with the identical name in the target path.

The copy-and-paste command `cp` works similar with the move command `mv`, except that it will not remove the item from the source path. Similar syntax applies to `cp` as follows, and arguments in Table 4.4 also apply to `cp`.

```
$ cp [<option>] <source> <target>
```

To permanently delete an item, use `rm` command as follows.

```
$ rm [<option>] <path>
```

For safety, when using `rm` the OS will keep prompting messages asking user to confirm whether to permanently delete an item or not. In some OS setups, it is by default forbidden to delete a directory unless it is empty. The following arguments in Table 4.5 can be used to change the setup.

It is possible though, that removed items using `rm` be recovered by expertise. For greater assurance that the deleted contents are truly unrecoverable, consider using `shred` which can physically overwrite the portion of hardware drive where the item is located. More details of `shred` can be found by

```
$ shred --help
```

TABLE 4.5Commonly used arguments and their effects for `rm` command.

Directory	Description
<code>-f</code>	Ignore nonexistent files and arguments and do not prompt.
<code>-r</code>	Remove directories and their contents recursively.
<code>-i</code>	Prompt before every removal.
<code>-d</code>	Remove empty directories.

TABLE 4.6

Commonly used wildcard characters.

Directory	Description
<code>*</code>	Matches any number of characters.
<code>?</code>	Matches one character.
<code>[...]</code>	Matches characters given in the square bracket, which can include a hyphen-separated range of characters.

4.2.5 Use of Wildcard Characters

When performing actions such as listing, moving, copying, removing, wildcard characters can be used in the path. For example, `ls a*` lists all items in the current directory that starts with letter “a”. Commonly used meta-characters are summarized in Table 4.6.

4.3 Access Control List

Each file or directory in the Linux OS is assigned with an owner and a permission list known as the Access Control List (ACL). ACL prevents unauthorized entities to access an item. The ACL of a file can be viewed using `ls -l`. An example has been given in Fig. 4.3 in the earlier section.

The first column of the output in Fig. 4.3 gives the type and permission of the item. The leading `d` and `-` indicate subdirectory and regular file respectively. Other commonly seen indicators are `l` for a symbolic link, `b` for a block device, `c` for a character device, `s` for a socket and `p` for a named pipe.

Following the item type indicator is the ACL of the item in the form of 9-bit permission that looks like `rwxrwxrwx`. The characters `r`, `w` and `x` stand for three types of permissions “read”, “write” and “execute” respectively. An explanation to these permissions is summarized in Table 4.7 and more details can be found in the `ls` command manual accessible using `ls --help`. The 9-bit permission of an item indicates the permissions of 3 types of users to the item, the first 3 bits the file owner, the middle 3 bits the file group, and

TABLE 4.7

Three types of permissions.

Directory	Description
r	View what is in the file or directory.
w	Change file contents; rename file; delete file. Add or remove files or subdirectories in a directory.
x	Run a file as a program. Change to the directory as the current directory; search through the directory; access metadata (file size, etc.) of files in the directory.

the last 3 bits other users. If any bit in the 9-bit permission is overwritten by a dash -, it means that the associated permission for the associated users is banned.

Commands `chown` and `chmod` can be used to change the ownership and ACL of an item respectively. Details are given in the following subsections.

4.3.1 Change Ownership and Group of a File or Directory

Administrative privilege is required to run `chown` command to change the ownership and group of a file or a directory as follows.

```
# chown [<option>] <new_owner>[:<new_group>] <path>
```

For example, in Fig. 4.4,

```
$ sudo chown root:root calculate_fib.sh
```

is used to change the ownership and group of file `calculate_fib.sh` from `sunlu` to `root`. Notice that elevated privilege is required to change its ownership, otherwise the request will be rejected as shown in Fig. 4.4.

```
sunlu@SUNLU-Laptop:~/Downloads$ ls -la
total 8
-rw-rw-r-- 1 sunlu sunlu 221 Jun 23 13:36 calculate_fib.sh
-rw-rw-r-- 1 sunlu sunlu 48 Jun 15 16:24 test.html
sunlu@SUNLU-Laptop:~/Downloads$ chown root:root calculate_fib.sh
chown: changing ownership of 'calculate_fib.sh': Operation not permitted
sunlu@SUNLU-Laptop:~/Downloads$ sudo chown root:root calculate_fib.sh
sunlu@SUNLU-Laptop:~/Downloads$ ls -la
total 8
-rw-rw-r-- 1 root root 221 Jun 23 13:36 calculate_fib.sh
-rw-rw-r-- 1 sunlu sunlu 48 Jun 15 16:24 test.html
```

FIGURE 4.4

Change ownership and group of a file.

4.3.2 Change Permissions of a File or Directory

Both the owner and the users with administrative privilege can change the ACL of a file or directory using `chmod` as follows. The ACL, in this context, is called the mode of the item.

```
$ chmod [<option>] <new_mode> <path>
```

For example, in Fig. 4.5, `g-w` is used to subtract “writing” permission from “group”, and `go+w` is used to add “writing” permission to ‘group’ and “other”, respectively. Here, `u`, `g` and `o` represents “user” (owner), “group” and “other”, and `r`, `w` and `x`, “read”, “write” and “execute”, respectively. Alternatively, 3-

```
sunlu@SUNLu-Laptop:~/Downloads$ ls -l
total 8
-rw-rw-r-- 1 sunlu sunlu 221 Jun 23 13:36 calculate_fib.sh
-rw-rw-r-- 1 sunlu sunlu 48 Jun 15 16:24 test.html
sunlu@SUNLu-Laptop:~/Downloads$ chmod g-w calculate_fib.sh
sunlu@SUNLu-Laptop:~/Downloads$ ls -l
total 8
-rw-r--r-- 1 sunlu sunlu 221 Jun 23 13:36 calculate_fib.sh
-rw-rw-r-- 1 sunlu sunlu 48 Jun 15 16:24 test.html
sunlu@SUNLu-Laptop:~/Downloads$ chmod go+w calculate_fib.sh
sunlu@SUNLu-Laptop:~/Downloads$ ls -l
total 8
-rw-rw-rw- 1 sunlu sunlu 221 Jun 23 13:36 calculate_fib.sh
-rw-rw-r-- 1 sunlu sunlu 48 Jun 15 16:24 test.html
sunlu@SUNLu-Laptop:~/Downloads$ chmod 664 calculate_fib.sh
sunlu@SUNLu-Laptop:~/Downloads$ ls -l
total 8
-rw-rw-r-- 1 sunlu sunlu 221 Jun 23 13:36 calculate_fib.sh
-rw-rw-r-- 1 sunlu sunlu 48 Jun 15 16:24 test.html
```

FIGURE 4.5

Change 9-bit permission (mode) of a file.

digit numbers such as 664 as shown in Fig. 4.5 can also represent a permission. The first digit is associated with the permission given to the “user”, where in this example 6 (110B) represents `rw-`. Each “1” in the binary number is associated with a permission. The second and third digits are associated with the permission given to “group” and “other” respectively. Hence, 664 would assign `rw-rw-r--` to the file.

4.3.3 Change Default Permissions

The default ACL for a newly created file or directory is defined by `umask`. Check its value by

```
$ umask
```

And change its value temporarily in the opening shell by

```
$ umask <new value>
```

The value of `umask`, after converting to binary, represents the bits in the 9-bit permission system that is disabled. For example, a `umask` value of 002 represents `rwxrwxr-x` in the 9-bit permission system, because the binary form of 002, 000000010, blocks the 8-th bit in the permission.

To permanently change the value of `umask`, configure it in `.bashrc` as introduced in Section 2.4.

4.4 Search through the System

The most frequently used 3 searching actions are as follows.

- Look for the location of a command using its name
- Look for the location of a file using its name (and other metadata such as size, permission, etc.)
- Look for the location of a file using a portion its content

Many approaches can be used to achieve the goals, some of which are introduced as follows.

4.4.1 Look for a Command

Use `type` to look for a command as follows.

```
$ type <command>
```

For example

```
$ type cd
cd is a shell builtin
$ type python
python is /usr/bin/python
$ type ls
ls is aliased to 'ls --color=auto'
```

4.4.2 Looking for Files by Metadata

Many Linux distributions come with built-in command `locate` that can be used to quickly locate a file by (a fraction of) its path as follows. Notice that as long as a file or directory's full path contains the searched content, there is

a chance that it will appear in the result. As a result, if the name of a directory is used for searching, all the items in that directory will likely to appear in the result (as their full paths contain the name of the directory).

```
$ locate <file or path fraction>
```

The mechanism of `locate` is explained as follows. The OS runs `updatedb` in the background usually once a day to update an internal database that gathers the names of files, and `locate` searches that database for the file. Notice that `locate` may fail to find recently added files if it has not been added to the database by `updatedb`. Besides, not all files' name are stored in that database by default, and a configuration file at `/etc/updatedb.conf` determines which files' name to be included. It is also worth mentioning that it will take some time to run `updatedb` for the first time, as it has a lot of things to add to the database during its initial run.

One may get confused by commands `locate` and `mlocate`. The two commands are very similar and they differ only beyond basic usage. Sometimes `locate` is aliased to `mlocate` when both functions are installed. In the scope of our discussion, we do not distinguish `locate` and `mlocate`.

An example of using `locate/mlocate` is given in Fig. 4.6. It can be seen from this example that the searching is done globally and does not rely on the current working directory.

```
sunlu@SUNLU-Laptop:~$ ls
anaconda3  Documents  Dropbox  eclipse-workspace  gurobl.log  octave-workspace  Public  snap      texmf
Desktop    Downloads  eclipse  gurobl          Music       Pictures        R       Templates  Videos
sunlu@SUNLU-Laptop:~$ locate Downloads
/home/sunlu/Downloads
/home/sunlu/.config/google-chrome/Downloads
/home/sunlu/.local/share/applications/_home_sunlu_Downloads_eclipse-installer_eclipse-inst-jre-linux64_eclipse-installer_.desktop
/home/sunlu/Downloads/calculate_fib.sh
/home/sunlu/Downloads/test.html
sunlu@SUNLU-Laptop:~$ cd Music
sunlu@SUNLU-Laptop:~/Music$ locate Downloads
/home/sunlu/Downloads
/home/sunlu/.config/google-chrome/Downloads
/home/sunlu/.local/share/applications/_home_sunlu_Downloads_eclipse-installer_eclipse-inst-jre-linux64_eclipse-installer_.desktop
/home/sunlu/Downloads/calculate_fib.sh
/home/sunlu/Downloads/test.html
```

FIGURE 4.6

Search for files and directories using `locate`.

4.4.3 Looking for Files by Variety of Properties

It is worth mentioning that for safety and privacy reasons, `mlocate` only shows the items that the user would be able to detect manually using `cd` and `ls` in the first place. Therefore, a regular user cannot locate any file under `/root` or other users' home directory using this method.

A more common and widely accepted way of looking for a file by its variety of attributes is using `find` as follows.

```
$ find [<options>] [<path>] <expression>
```

The `<options>` argument can be used to configure how the symbolic links

should be handled. Symbolic links are like shortcut in Windows PC. It is a link that points to other directories or files. If **-P** (default option) is used, the symbolic links are treated as the links themselves, but not the files they are linking to, whereas **-L** does the opposite. Debugging modes and query optimization can also be enabled from **<options>**. The **<path>** argument specifies the directory from where the query is conducted. The **<expression>** argument specifies the keyword and the query method.

The following are examples of using **find** in different scenarios. The examples are taken from RedHat at [2].

List everything in a directory and its subdirectories

```
$ find ~/Documents -ls
3554235 0 drwxr-xr-x [...] 05:36 /home/seth/Documents/
3554224 0 -rw-rw-r-- [...] 05:36 /home/seth/Documents/Foo
3766411 0 -rw-rw-r-- [...] 05:36 /home/seth/Documents/Foo/foo.txt
```

Find files by name

Case sensitive: **-name**

```
$ find / -name "*Foo*txt" 2>/dev/null
/home/seth/Documents/Foo.txt
```

Case insensitive: **-iname**

```
$ find / -iname "*Foo*txt" 2>/dev/null
/home/seth/Documents/Foo.txt
/home/seth/Documents/foo.txt
/home/seth/Documents/foobar.txt
```

Wildcards such as ***** can be used. Commonly used wildcards are given in Table 4.6.

Find files by the content

```
$ find ~/Documents/ -name "*txt" -exec grep -Hi penguin {} \;
/home/seth/Documents/Foo.txt:I like penguins.
/home/seth/Documents/foo.txt:Penguins are fun.
```

where in this example, **-exec** allows executing a command to the findings. Notice that **grep** by itself can also be used to search files by contents.

Find files by type

To find all files,

```
$ find ~ -type f
/home/seth/.bash_logout
/home/seth/.bash_profile
/home/seth/.bashrc
```

```
/home/seth/.emacs  
/home/seth/.local/share/keyrings/login.keyring  
/home/seth/.local/share/keyrings/user.keystore  
/home/seth/.local/share/gnome-shell/gnome-overrides-migrated
```

Specifically, to find empty files,

```
$ find ~ -type f -empty  
random.idea.txt
```

To find all subdirectories,

```
$ find ~/Public -type d  
find ~/Public/ -type d  
/home/seth/Public/  
/home/seth/Public/example.com  
/home/seth/Public/example.com/www  
/home/seth/Public/example.com/www/img  
/home/seth/Public/example.com/www/font  
/home/seth/Public/example.com/www/style
```

Use `-maxdepth` to set the searching depth as follows.

```
$ find ~/Public/ -maxdepth 1 -type d  
/home/seth/Public/  
/home/seth/Public/example.com
```

Find files by the timestamp it was accessed / changed / metadata changed

To find files whose latest modified time is at least 30 days ago (30 days ago or earlier), use

```
find /var/log -mtime +30
```

where `-atime`, `-ctime` and `-mtime` search based on the number of days since each file was accessed, changed or had its metadata changed respectively. The alternatives `-amin`, `-cmin` and `-mmin` work similarly in minutes. The `+` indicates “at least” whereas `-` indicates “not more than”.

4.5 File Archive

The `tar`, `gzip` and `zip` commands can all be used in files archive, but with different features, as given in Table 4.8.

Only `tar` command is introduced here, as it is the most commonly used one among the three, and it can satisfy most use cases. A common way of using `tar` is as follows. Use

```
$ tar -cvzf <archive-file> <file1> <file2> <file3> ...
```

TABLE 4.8

Commonly used file archive tools.

Directory	Description
tar	Save many files together into a single tape or disk archive, and can restore individual files from the archive. By default, it does not compress the files. However, -z option can be used in combination of the command to add compression feature.
gzip	Compress or restore files.
zip	Compress multiple files one-by-one and integrate them together into a single file.

to archive and zip files, and

```
$ tar -xvzf <archive-file>
```

to restore files from the archive file. The commonly used archive file name, in this scenario, is **<filename>.tgz**.

The detailed explanation to all available options for **tar** can be found using **tar --help**. The most commonly used options are **-c**, **-x**, **-z**, **-f** and **-v**, standing for creating compress tape, extracting (restoring) file, adding compressing feature, using file archive, and listing processed files in the console, respectively.

5

Software Management

CONTENTS

5.1	Tasks and Approaches	57
5.2	RPM Package	58
5.2.1	Brief Introduction	58
5.2.2	Package Management Using <code>rpm</code>	59
5.2.3	Package Management Using <code>yum</code> and <code>dnf</code>	59
5.3	DEB Package	61
5.4	Linux Kernel Management	62

Linux as an OS monitors and maintains the software installed on the system. Different Linux distributions may use different tools to manage the software.

5.1 Tasks and Approaches

The OS should support at least the following tasks.

- Install software.
 - Install software from the installation package.
 - If possible, automatically find and download the installation package.
 - If possible, automatically find, download and install all the dependencies.
- Query software.
 - Query the software already installed on the system.
 - If possible, query software online.
- Remove software.
- Update software.

In Linux, software information (dependencies, version, instruction to installation, etc.) are stored in special data structures known as packages. When installing a software, the OS locates the packages and install them according to the instructions that comes with the packages. The OS also traces the software it has installed, monitors their behaviors, performs updates or uninstalls per requested by the user. Different Linux distributions may use different package formats and package management tools. The most well-known ones include:

- Red Hat Package Manager (RPM), also recursively named as RPM Package Manager
- Debian Package Manager (DEB)

Note that RPM and DEB can refer to both the package formats and the package managing tools. There are proponents on both sides of RPM versus DEB, and both of them are very popular tools in different Linux distributions.

Linux kernel is also a software and needs to be monitored and updated from time to time. Linux kernel management is also briefly introduced in this chapter. Notice that advanced kernel management such as kernel customization goes beyond the scope of this notebook, hence are not included in this chapter.

5.2 RPM Package

RPM (.rpm) is the preferred package format for Red-Hat-based distributions such as RHEL, Fedora, CentOS and Oracle Linux. It uses `rpm` command to mange RPMs. In a later stage, other tools such as `yum` and `dnf` have been added to the library for better user experience and enhanced RPM facility.

5.2.1 Brief Introduction

RPM contains rich information, including not only the payload of the software such as commands, configuration files, libraries and documentation, but also metadata such as the source of the package, dependencies, etc.

The name of an RPM should follow the protocol and by itself contains important information about the software. An example is given below. Use `rpm -q` followed by the software name `python3` to query the package as follows. If the package is installed, its full name should be returned.

```
$ rpm -q python3  
python3-3.9.18-3.el9_4.1.x86_64
```

where in this example the name of the software is `python3`, the version `3.9.18`, the release (build) `3.el9_4.1` with release number `4`, distribution

`e19_4` and patch number 1, and finally architecture `x86_64`. When the package is stored on the machine, it should also come with the suffix `.rpm`.

To retrieve more details, use `rpm -qi` instead and

```
$ rpm -qi python3
Name      : python3
Version   : 3.9.18
Release   : 3.e19_4.1
Architecture: x86_64
Install Date: Tue 09 Jul 2024 12:48:28 PM +08
Group     : Unspecified
Size      : 33021
License    : Python
Signature  : RSA/SHA256, Tue 21 May 2024 06:30:22 AM +08, Key ID 199
             e2f91fd431d51
Source RPM : python3.9-3.9.18-3.e19_4.1.src.rpm
Build Date : Fri 17 May 2024 10:49:06 PM +08
Build Host : x86-64-03.build.eng.rdu2.redhat.com
Packager   : Red Hat, Inc. <http://bugzilla.redhat.com/bugzilla>
Vendor    : Red Hat, Inc.
URL       : https://www.python.org/
Summary   : Python 3.9 interpreter
Description :
Python 3.9 is an accessible, high-level, dynamically typed, interpreted
programming language, designed with an emphasis on code readability.
It includes an extensive standard library, and has a vast ecosystem of
third-party libraries.
...
```

can be retrieved.

5.2.2 Package Management Using `rpm`

An RPM package is required to install the software. An RPM package comes from the upstream software provider, who collects source codes, binary files, and other data, builds the software, and signs the package.

The command `rpm` can be used to install the software from an RPM package. Note that installing software using `rpm` requires the precise location of the RPM package and all dependencies to be installed in advance, which can be inconvenient. The `rpm` command can also be used to validate, query, update, and remove software.

When a package is installed on the machine, its information is saved in the RPM database managed by the OS.

5.2.3 Package Management Using `yum` and `dnf`

Due to the drawbacks of `rpm` mentioned earlier, other tools have been developed to handle package upstream repositories and dependencies more con-

veniently. These tools are Yellowdog Updater Modified (YUM) and its next generation, Dandified YUM (DNF), with their corresponding commands `yum` and `dnf` respectively.

YUM and DNF introduce the concept of repositories, allowing RPM packages to be part of a larger software “package tree”. It becomes the publisher’s responsibility to ensure that all the dependencies of the RPM can be traced from the repositories. If a dependency is not present, YUM and DNF can search the repositories to find the dependencies and install them together. When the user wants to install an RPM that has not been downloaded onto the machine, YUM and DNF can search the repositories for the RPM, download, and install it automatically. As a result, the user does not need to manually install all the dependencies or find and download the RPM themselves. In this sense, YUM and DNF can be taken as a wrapper of `rpm` that come with additional features such as repository resolution and package retrieval. They run `rpm` internally once the packages have been downloaded to the machine.

DNF is a further improvement of YUM with better dependency resolution and less memory usage. In many occasions, command `yum` can be replaced by `dnf` seamlessly. In some Linux distributions such as RHEL 9, `yum` is used as an alias for `dnf`. In the remaining part of the section, only DNF is discussed.

The basic syntax of DNF is

```
$ dnf <operation> <package>
```

where `<operation>` include

- `install`: install a package
- `remove`: remove a package
- `search`: search the repositories for a package
- `autoremove`: remove dependencies packages irrelevant to currently installed program
- `check-update`: check updates of packages from the repositories without downloading or installing the updates
- `downgrade`: revert to a previous version
- `info`: provide the basic information of a package
- `reinstall`: reinstall a package
- `upgrade`: upgrade packages
- `exclude`: exclude a package from transaction

Each time when `dnf` starts, it checks its configuration file which is located at `/etc/dnf/dnf.conf`. Users can edit the configuration file to change the behavior of DNF, for example, to enable / disable GNU Privacy Guard (GPG)

key check, the maximum number of different versions of the same software the system can keep, whether to delete dependencies when removing software, where to keep cache and log files, etc.

DNF keeps a list of repositories at `/etc/yum.repos.d` directory as its upstream software provider. The stored repositories are essentially text files named with suffix `.repo`, inside which are the name of the repository, the URL, the GPG key, etc. To install software whose repository is not stored by default in this folder (this could happen when the software provider is a third-party and its software is not registered in the default repositories), users may need to add the repository and GPG key manually. By adding the repository configuration file and importing the GPG key, the user enables DNF to use the new repository to install and update software packages.

Other commonly used commands include

- `dnf list --installed [<package>, ...]`: list installed packages
- `dnf list --available [<package>, ...]`: list available packages
- `dnf list --upgrades [<package>, ...]`: list upgrades available for the installed packages
- `dnf list --autoremove`: list dependencies not used by any software
- `dnf -q`: query repository for information; this is a powerful and rich command whose details are too many to be covered here
- `dnf deplist <package>`: query dependencies of a package
- `dnf history`: check historical transactions
- `dnf clean all`: clean all the files from cache

A full list of `dnf` commands and their explanations can be found elsewhere at [3].

5.3 DEB Package

DEB (`.deb`) package is developed by Debian GNU/Linux project, and it is used by Debian and other Debian-based distributions such as Ubuntu and Linux Mint. The basic tool to install, remove and query DEB is `dpkg`, but the end users often use `apt*` commands to perform package management, rather than using `dpkg` directly.

Like RPM, DEB contains both the metadata of the software known as the control files as well as the payload of the software. A control file may look like the following (example from [4])

```

Package: hello
Version: 2.9-2+deb8u1
Architecture: amd64
Maintainer: Santiago Vila <sanvila@debian.org>
Installed-Size: 145
Depends: libc6 (>= 2.14)
Conflicts: hello-traditional
Breaks: hello-debhelper (<< 2.9)
Replaces: hello-debhelper (<< 2.9), hello-traditional
Section: devel
Priority: optional
Homepage: https://www.gnu.org/software/hello/
Description: example package based on GNU hello
The GNU hello program produces a familiar, friendly greeting. It
allows non-programmers to use a classic computer science tool which
would otherwise be unavailable to them.

.
Seriously, though: this is an example of how to do a Debian package.
It is the Debian version of the GNU Project's 'hello world' program
(which is itself an example for the GNU Project).

```

which is very similar to RPM package information.

More details about DEB as well as the use of `apt*` can be found at [4].

5.4 Linux Kernel Management

Different Linux distributions, though essentially using the same OS kernel, may differ when it comes to how they update and maintain the kernel. In the scope of this notebook, only RHEL is introduced in a very brief manner. A more detailed explanation to how RHEL manages kernel can be found at [5], which is not only a handbook of how RHEL manages Linux kernel, but also a good learning material for OS kernels in general.

Notice that RHEL kernel is a modified version of the upstream Linux kernel by Red Hat engineers. Therefore, it may differ from kernels used in other Linux distributions.

Package `kernel*` such as `kernel-core` are the main package of Linux kernel. Just like other packages, `dnf` can be used to query and update that package. It is also possible to install multiple kernels with different versions on the same physical machine, in which case there will be a boot entry for each kernel and the user can select which kernel to boot upon restarting the system.

To query, install and update the kernel, use

```

$ sudo dnf -q kernel-core
$ sudo dnf install kernel-<version>

```

```
$ sudo dnf update kernel
```

respectively.

Kernel modules are scripts used to extend OS functionality. The user can develop and deploy their own modules to customize the OS.

To list existing modules and to query information of these modules, with **kmod** installed, use

```
$ lsmod  
$ modinfo <module name>
```

respectively.

With **kernel-devel**, **gcc** and **elfutils-libelf-devel** installed, the user can develop, test and deploy customized modules.

Linux kernel parameters are tunable values relevant to OS behavior. They can be changed in run-time or when system reboots. It is possible to address the kernel parameters via the following approach.

- Use **sysctl** command
- Edit **/proc/sys/** to change kernel parameters temporarily
- Edit configuration files in **/etc/sysctl.d/**

For example,

```
$ sysctl -a
```

displays all kernel parameters values.



6

Process Management

CONTENTS

6.1	General Introduction to Process	65
6.1.1	Process	65
6.1.2	Thread	67
6.2	Process Management in Linux	68
6.2.1	Monitor Process	68
6.2.2	Terminate Process	69
6.2.3	Switch between Foreground and Background	70
6.2.4	Change Process Priority	71

A process refers to an instance of a computer program that is running in the system. Managing processes is one of the essential tasks of an OS. In a Windows system, the user can use the task manager, a graphical tool, to check and manage all the running processes. In a Linux system, the user can manage process in the prompt console using bash commands.

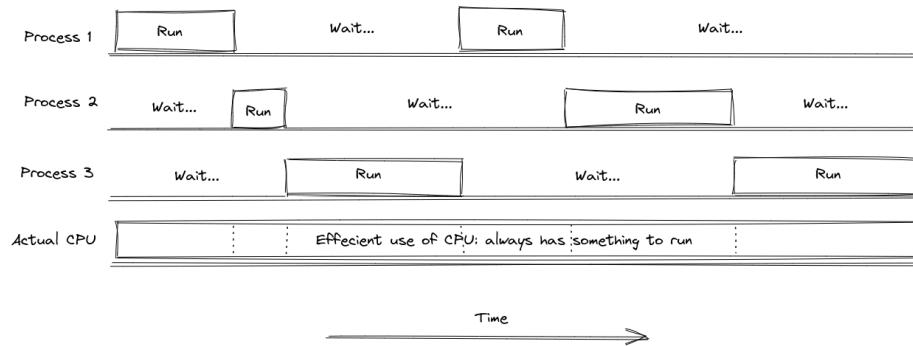
6.1 General Introduction to Process

A process is the fundamental unit for the OS to manage the resources used by a running program.

6.1.1 Process

To improve the efficiency of the CPU, the OS allows multiple processes (also called tasks, jobs) to share the computational capability and memory of the system, each thinking that it is exclusively using the all machine resources, as shown in Fig. 6.1.

The status of a process is stored in its *Process Control Block (PCB)*. The PCB is a special data structure used to describe the dynamic of a process. The OS manages the PCBs and control the processes accordingly. Some of the attributes of a PCB are summarized in Table 6.1.

**FIGURE 6.1**

A demonstration of running multiple processes on a single-core CPU.

TABLE 6.1
Some attributes of a PCB.

Name	Description
Identifier	The unique ID of the process.
State	The state of the process, for example, running, suspended, terminated.
Priority	Priority level in comparison with other processes.
Program Counter	A pointer to the next line of program to be executed.
Memory regions	A pointer to the RAM where the code and data of the process is stored.
Accounting Information	Time limits, clock time used, etc.

A process shall at least have the following states. The fundamental states of a process and their transferring are introduced in Fig 6.2, where “blocked” indicates that the process is waiting for other inputs to carry out the remaining part of the program, and “suspended” indicates that the process is hold for some reasons. When a process is offloaded from the CPU, its context is moved from the CPU registers to the PCB of the progress.

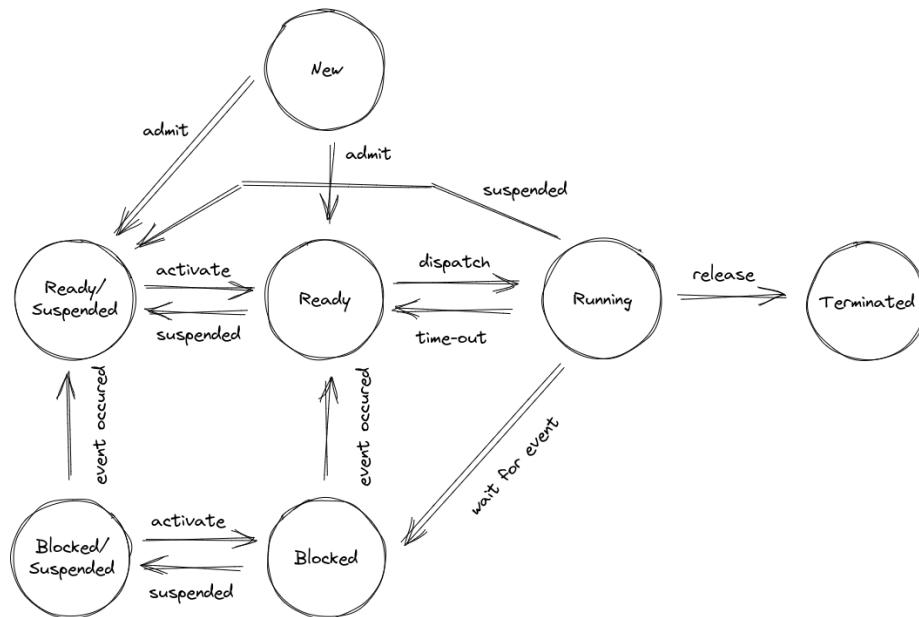


FIGURE 6.2

Fundamental states of a process and their transferring.

There are different types of processes. For example, based on the source of the processes, there are OS triggered processes and user triggered processes, the first of which usually has a higher priority. Based on the running environment, there are front-ground processes and background processes. Based on the resources used, there are CPU processes and I/O processes.

A process usually has a relatively isolated environment, and it does not share memory storage with other processes. Special inter process communication mechanism, which is often referred as “pipe”, is required for processes to talk to each other. Inter process communication requires OS level controls.

6.1.2 Thread

A “thread” is like a work dispatch inside a process. There can be multiple threads in a process, as shown in Fig 6.3. Each thread has its own CPU

register values and stack, but they share the same program, memory and file storage addresses.

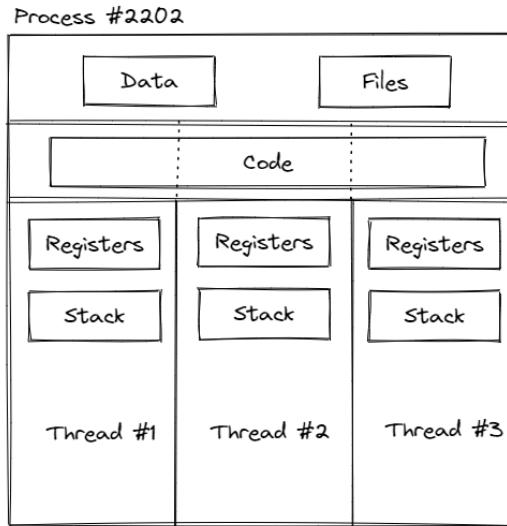


FIGURE 6.3

A demonstration of multiple threads in a process.

Threads differ from processes in the following aspects:

- A thread is lighter than a process, occupying less resources to create.
- Sharing memories and resources among threads in a process is easier than sharing among processes, because they naturally share address space.
- It is easier to enable parallel computation for the threads in the process when it is running on a multi-core CPU.

Notice that for many OS, including Linux, the kernel can provide thread level services.

6.2 Process Management in Linux

Basic process management commands including monitoring and terminating a process are given. Switching a process between foreground and background is also introduced.

6.2.1 Monitor Process

Two commands, `ps` and `top`, are widely used in monitoring the running process in the OS. They can be used stand-alone, without additional arguments as follows.

```
$ ps [options]
```

or

```
$ top
```

The major difference between these two commands is that `ps` provides a screenshot (in a text format) of a list of processes including their names, process IDs (PID) and owners, etc., running at the instance, while `top` provides a dynamic and frequently refreshing display of the running processes as well as their associated resources usage. Figs 6.4 and 6.5 give a quick demo of how the execution of the two commands look like.

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	15:34	?	00:00:02	/sbin/init splash
root	2	0	0	15:34	?	00:00:00	[kthreadd]
root	3	2	0	15:34	?	00:00:00	[rcu_gp]
root	4	2	0	15:34	?	00:00:00	[rcu_par_gp]
root	5	2	0	15:34	?	00:00:00	[netns]
root	7	2	0	15:34	?	00:00:00	[kworker/0:0H-events_highpri]
root	10	2	0	15:34	?	00:00:00	[mm_percpu_wq]
root	11	2	0	15:34	?	00:00:00	[rcu_tasks_rude_]
root	12	2	0	15:34	?	00:00:00	[rcu_tasks_trace]
root	13	2	0	15:34	?	00:00:00	[ksoftirqd/0]
root	14	2	0	15:34	?	00:00:00	[rcu_sched]
root	15	2	0	15:34	?	00:00:00	[migration/0]
root	16	2	0	15:34	?	00:00:00	[idle_inject/0]
root	17	2	0	15:34	?	00:00:00	[cpuhp/0]
root	18	2	0	15:34	?	00:00:00	[cpuhp/1]
root	19	2	0	15:34	?	00:00:00	[idle_inject/1]
root	20	2	0	15:34	?	00:00:00	[migration/1]
root	21	2	0	15:34	?	00:00:00	[ksoftirqd/1]
root	23	2	0	15:34	?	00:00:00	[kworker/1:0H-events_highpri]
root	24	2	0	15:34	?	00:00:00	[cpuhp/2]
root	25	2	0	15:34	?	00:00:00	[idle_inject/2]

FIGURE 6.4
Execution of `ps -ef` command.

Notice that `top` is a running application where the user can keep interacting with to filter for particular processes, while `ps` is more of a one-time command and its output can be saved into a text file for further processing.

Some commonly used options for `ps` are summarized below.

- `-e`: list all processes
- `-f`: list details of the processes
- `-C <command name>` filter by command
- `-u <username>` filter by user

```
top - 15:58:28 up 24 min, 1 user, load average: 0.23, 0.29, 0.42
Tasks: 233 total, 1 running, 232 sleeping, 0 stopped, 0 zombie
%CPU(s): 11.3 us, 1.9 sy, 0.2 ni, 86.4 id, 0.2 wa, 0.0 hi, 0.0 si, 0.0 st
Mem: 11869.6 total, 8798.0 free, 1271.7 used, 1799.9 buff/cache
Swap: 2048.0 total, 2048.0 free, 0.0 used. 10128.1 avail Mem

      PID USER      PR  NI    VIRT    RES   SHR S %CPU %MEM     TIME+ COMMAND
 2105 sunlu    20   0 5252532 266064 120196 S 16.7  2.2  0:11.93 gnome-shell
 4054 sunlu    20   0 786512 46900 36032 S  9.3  0.4  0:00.28 nautilus
 4057 sunlu    20   0 2692040 44404 33308 S  6.7  0.4  0:00.20 org.gnome.Chara
 4051 sunlu    20   0 196716 23468 16576 S  3.3  0.2  0:00.10 gnome-control-c
 1827 sunlu    20   0 7388992 127136 81960 S  3.0  1.0  0:06.72 Xorg
 4055 sunlu    20   0 343916 24564 17716 S  2.7  0.2  0:00.08 gnome-calculato
 3706 sunlu    20   0 894312 55484 41984 S  2.3  0.5  0:01.61 gnome-terminal-
 1769 sunlu    20   0 9648 5936 4132 S  1.3  0.0  0:00.59 dbus-daemon
 1819 sunlu    39  19 710376 28636 19016 S  1.0  0.2  0:00.52 tracker-miner-f
 261 root     19  -1 110588 59760 58172 S  0.7  0.5  0:00.86 systemd-journal
 2254 sunlu    20   0 392040 11920 6852 S  0.7  0.1  0:00.82 ibus-daemon
 14 root     20   0     0     0   0 I  0.3  0.0  0:00.79 rcu_sched
 27 root     20   0     0     0   0 S  0.3  0.0  0:00.44 ksoftirqd/2
 128 root    0 -20     0     0   0 I  0.3  0.0  0:00.39 kworker/u9:0-i915_flip
 604 root     20   0     0     0   0 S  0.3  0.0  0:00.88 nv_queue
```

FIGURE 6.5Execution of `top` command.

6.2.2 Terminate Process

To kill a process, use the `kill` command as follows.

```
$ kill <option> <process ID>
```

The `kill` command offers different options to kill a process. Use `kill -l` to list down the options as given below (and many more).

```
1) SIGHUP  2) SIGINT  3) SIGQUIT  4) SIGILL
5) SIGTRAP  6) SIGABRT  7) SIGBUS  8) SIGFPE
9) SIGKILL 10) SIGUSR1 11) SIGSEGV 12) SIGUSR2
13) SIGPIPE 14) SIGALRM 15) SIGTERM 16) SIGSTKFLT
17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
...
```

Commonly used `kill` options are `kill -9` (SIGKILL) and `kill -15` (SIGTERM) followed by the PID. It is mostly recommended to use `kill -15` over `kill -9`, as it allows the process to clean up and terminate properly. Most well designed software defines the protocol to terminate the process in a safe way. This is also the default termination option. Command `kill -9`, on the other hand, force the OS to terminate the process immediately. It is used mostly when the process is not responding and cannot be terminated using `kill -15`.

Another command `killall` runs similarly as `kill`, except that it kills signals based on the command, not the process ID. As a result, it is possible to use it to kill multiple signals with the same command at a time. Use it with cautions.

Notice that in Linux the process is arranged in a tree structure; killing a parent process will automatically terminate its children processes, and killing a child process may result in its parent process to restart a new child process.

6.2.3 Switch between Foreground and Background

To start a command as a background process, use & in the end of the command as follows

```
$ <command> &
```

This is useful if a command is time consuming and it does not need additional user interaction while running.

To check commands running in the background, use

```
$ jobs
[1] <status> <command>
[2] <status> <command>
[3]+ <status> <command>
[4]- <status> <command>
...
...
```

A plus sign “+” indicates that the command was the most recent one placed in the background, and the minus sign “-”, the second most recent one. To bring a background command to the foreground, use

```
$ fg %<job number>
```

where <job number> is the index given in `jobs` outputs. If no job number is given, the most recent job will be brought to the foreground.

6.2.4 Change Process Priority

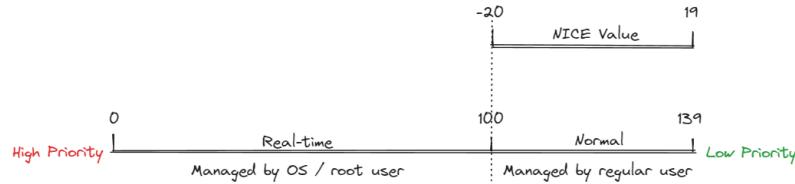
The priority of a process is given by an integer between -100 and 39, the smaller the more prioritized. This value can be viewed using `top`, as shown in Fig. 6.5 under column “PR”. The process with priority -100 is displayed as `rt`, indicating the highest priority.

The entire priority range from -100 to 39 is divided into two portions. A priority between -100 and -1 is known as “real-time”, whereas between 0 and 39, “normal”. In practice, real-time processes are mainly managed by the root user and are not accessible to regular users. A regular user’s process typically falls within the normal priority range of 0 to 39. A “NICE value” is associated with each of these priorities. A priority of 0 corresponds to NICE value of -20, and 39 to 19. The user can adjust the NICE value of his process and hence change its priority. This is demonstrated by Fig. 6.6.

For normal processes, their NICE values can be viewed using `top` as shown in Fig. 6.5 under column “NI”. Real-time processes do not use NICE value, hence NI are displayed as 0 for these processes.

A properly designed OS should have built-in algorithms to manage process priorities and scheduling. However, tools like `nice` and `renice` are provided to give users and administrators the ability to influence these priorities in specific situations. Examples are given below.

To start a command with a specified NICE value, use

**FIGURE 6.6**

Priority levels in Linux.

```
$ nice -n <increment> <command>
```

and <increment> will be the NICE value of the command.

To change the NICE value of an existing process, use

```
$ renice -n <increment> -p <process id>
```

where <increment>, being either positive or negative, will be added to the current NICE value of the process with the specified process ID.

It is also possible to manage the priorities not by individual process ID, but by a group of processes. This is useful when a group of processes belongs to a similar service or to a group of users, and we want to change the priority of the service or users all together.

— | — | —

Part II

Linux Advanced

— | — | —



7

Administration

CONTENTS

7.1	Introduction to Linux Administration	75
7.2	Root Account Management	75
7.3	User Management	76

...

7.1 Introduction to Linux Administration

...

7.2 Root Account Management

Managing and protecting the *root* user account is a key portion in Linux administration. It is worth mentioning that the root user is different from a “*sudoer*”, i.e., a user that can use `sudo` command, although they both have elevated privileges when comes to system administration. A more detailed explanation is given below.

The root user refers to the user that is created by the system upon first installation of the OS. Its username is by default “root”, with a UID of 0, and a GID of 0. Its home directory is by default `/root` instead of `/home/<user name>`. The default prompt for the root user is often # instead of \$ for other users. The root user has administrative privileges and can do almost anything without being denied or questioned by the system. As a commonly used safety practice, the password for the root user is usually disabled, thus nobody can login to the system as the root user using username and password authentication.

Notice that in theory the root user does not necessarily need to use the username “root”, although it is a default convention. The administrative

comes with the UID of 0, not the username. Thus, it is possible to assign the administrative account a different username. It is also possible to create multiple accounts with administrative privileges by assigning UID of 0 to multiple accounts, although it is not recommended to share the same UID among different users.

Although the administrative privilege of root user does not come with its username, in some systems, the username “root” is given elevated privilege anyway. More details about this is introduced later.

Check the root user information as follows.

```
$ cat /etc/passwd | grep ^root
root:x:0:0:root:/root:/bin/bash
```

where the third and fourth column of above give the UID and GID of the root user, respectively.

For comparison, a regular user would have far different UID and GID as shown below.

```
$ cat /etc/passwd | grep ^sunlu
sunlu:x:1000:1000:Sun Lu,,,:/home/sunlu:/bin/bash
```

A sudoer refers to the regular users who can temporarily elevate its privileges and execute administration commands using `sudo <privileged-command>`. A sudoer can also switch to root user prompt using `sudo su` (and quit root user prompt using `exit`). Their sudo privilege comes from the fact that they are included in the “sudo group”.

Use `groups` to check existing defined groups in the system, and `groups <user name>` the groups a user is engaged. An example is given below.

```
$ groups
sunlu adm cdrom sudo dip plugdev lpadmin lxd sambashare docker
$ groups sunlu
sunlu : sunlu adm cdrom sudo dip plugdev lpadmin lxd sambashare docker
```

The elevated privilege of the sudoer group is defined in `/etc/sudoers`. A section of the file is given below, where % appeared in front of `admin` and `sudo` is used to indicate group name.

```
# User privilege specification
root    ALL=(ALL:ALL) ALL

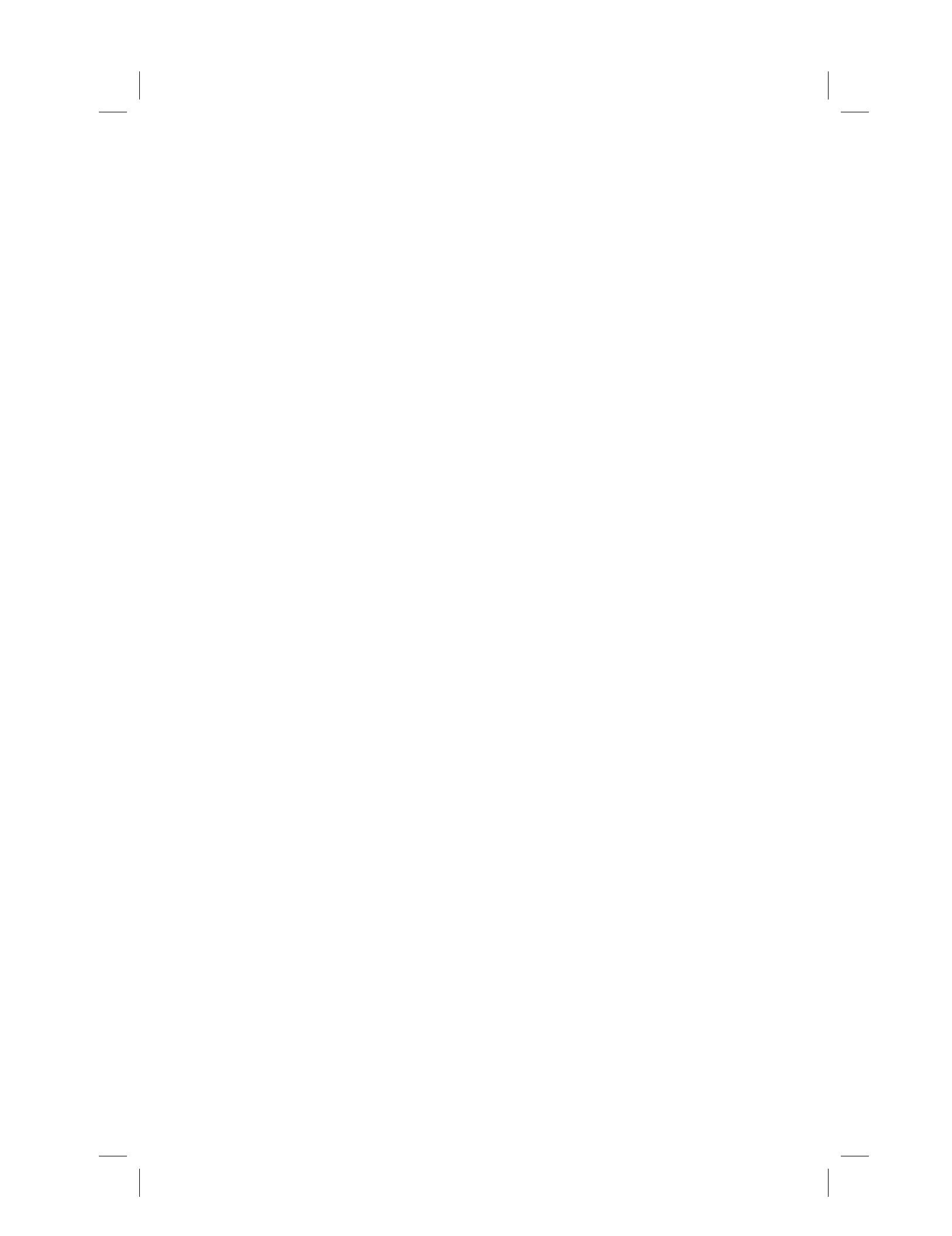
# Members of the admin group may gain root privileges
%admin  ALL=(ALL:ALL) ALL

# Allow members of group sudo to execute any command
%sudo   ALL=(ALL:ALL) ALL
```

It is possible to give the root account a password, thus enabling root authentication. This approach is sometimes used for troubleshooting and recovering purposes, but it is not a good practice in routine operations.

7.3 User Management

xxx



8

Storage Management

CONTENTS

8.1	Partitions and Filesystems	79
8.2	Disk Partition Table Manipulation	82
8.2.1	Disk Partition	82
8.2.2	Disk Partition Table Manipulation	82
8.3	Mount, Unmount and Format a Partition	83

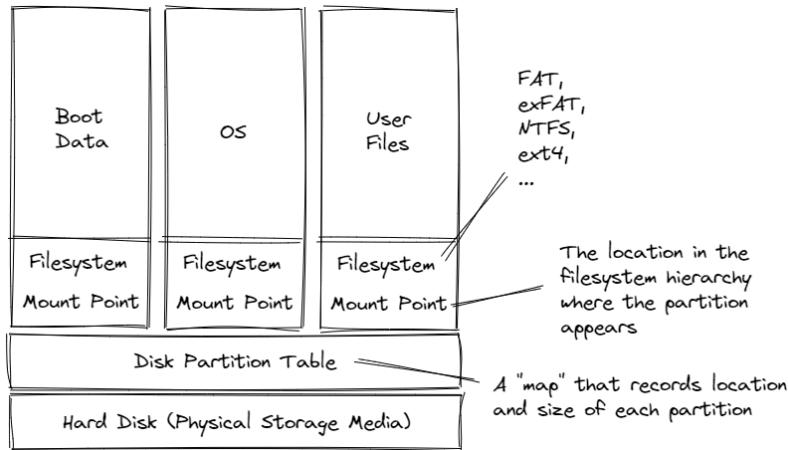
Upon installation of the system, the OS shall scan the machine, look for hard drives, and either automatically or manually partition it into partitions and mount the partitions to different locations in the OS such as / (root directory), /home/, swap space, etc.

It is often not necessary for a casual user to manage the partitions and mountings by himself as there is the option to let Linux installation handle them automatically. However, when comes to Linux servers where hardware and storage is scaled up and down frequently, it is recommended that the administrator shall understand how hard drive can be managed. Linux provides flexible tools to monitor and manage the storage of the machine, including manipulating partition table, format partition, and managing its mounting point.

8.1 Partitions and Filesystems

A partition is a logical slice of the hard drive managed by the OS via partition table which is introduced in more details in a later stage. A filesystem, on the other hand, reflects how OS formats data in the partition and where it is mounted in the OS filesystem hierarchy. This is demonstrated by Fig. 8.1.

Usually, each filesystem is associated with a partition. The partition focuses more on the logical separation of the hard drive, while the filesystem focuses more on how the operating system manages the data within that partition. Ideally they are consistent and there should be a clear correspondence. However, it is possible that when the filesystem is not sized correctly, the filesystem is smaller than its associated partition, and a part of the storage

**FIGURE 8.1**

A demonstration of how a hard drive is used in the OS.

becomes unused. A filesystem is only meaningful on a partition because it describes how that partition is formatted and managed. Therefore, a filesystem cannot exist without a partition. However, a partition can exist without a filesystem. It will be an unused storage space that is not useful to the operating system until it is formatted with a filesystem.

A demonstration of partitions versus filesystems is given in Fig. 8.2.

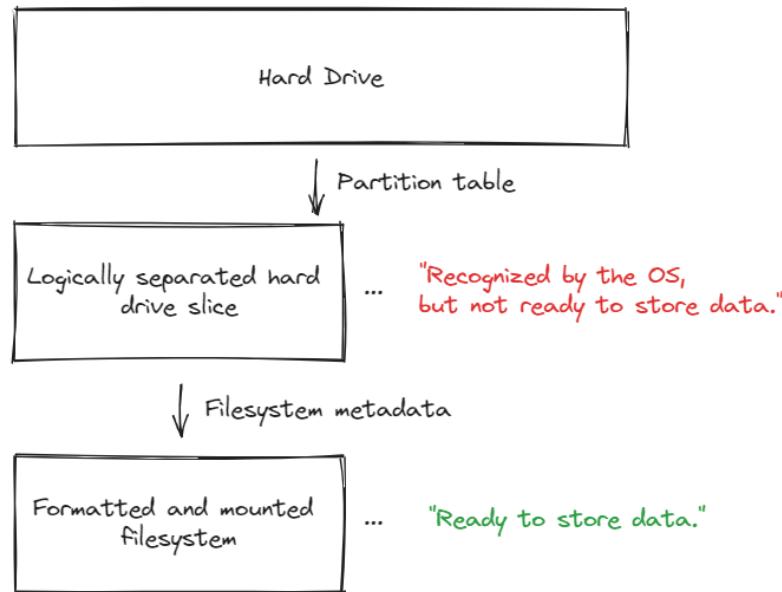
Use `df` to monitor the status of the mounted storage, including the filesystem name, size, and used percentage. Command `df -h` gives a nice snapshot of the storage usage in a human-readable format. An example is given below.

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
tmpfs          1.2G  2.1M  1.2G  1% /run
/dev/sda3       916G  51G  818G  6% /
tmpfs          5.8G    0  5.8G  0% /dev/shm
tmpfs          5.0M  4.0K  5.0M  1% /run/lock
/dev/sda2       512M  5.3M  507M  2% /boot/efi
tmpfs          1.2G 120K  1.2G  1% /run/user/1000
```

from where it can be seen that the most of the storage of the system, in this case 818G, is mounted on the root directory.

Use `lsblk` to list information about block devices in the OS. Block devices refer to nonvolatile mass storage devices whose information can be accessed in any order, such as hard disks and CD-ROMs. An example is given below.

```
$ lsblk
NAME  MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
loop0   7:0    0   4K  1 loop  /snap/bare/5
loop1   7:1    0  62M  1 loop  /snap/core20/1593
```

**FIGURE 8.2**

A demonstration of how partitions and filesystems relate to each other.

```

loop2    7:2    0    62M  1  loop  /snap/core20/1611
loop3    7:3    0   163.3M 1  loop  /snap/firefox/1670
loop4    7:4    0   177M  1  loop  /snap/firefox/1749
loop5    7:5    0   400.8M 1  loop  /snap/gnome-3-38-2004/112
loop6    7:6    0   248.8M 1  loop  /snap/gnome-3-38-2004/99
loop7    7:7    0   81.3M  1  loop  /snap/gtk-common-themes/1534
loop8    7:8    0   91.7M  1  loop  /snap/gtk-common-themes/1535
loop9    7:9    0   45.9M  1  loop  /snap/snap-store/575
loop10   7:10   0    47M   1  loop  /snap/snapd/16010
loop11   7:11   0   45.9M  1  loop  /snap/snap-store/582
loop12   7:12   0    47M   1  loop  /snap/snapd/16292
loop13   7:13   0   284K   1  loop  /snap/snapd-desktop-integration/10
loop14   7:14   0   284K   1  loop  /snap/snapd-desktop-integration/14
sda      8:0    0   931.5G 0  disk
|-sda1   8:1    0     1M   0  part
|-sda2   8:2    0   513M   0  part  /boot/efi
|-sda3   8:3    0   931G   0  part  /
sr0     11:0   1   1024M  0  rom

```

which gives the size, the type and the mount point of all the storages.

Use `blkid` to print block device attributes. An example is given below.

```
$ blkid
/dev/sda3: UUID="d0b15b7c-71f2-41f4-b67a-e7c69446feab" BLOCK_SIZE
```

```
= "4096" TYPE="ext4" PARTUUID="4b570507-6aa0-46c7-ac1b-06cb4c8bdb61"
"
```

8.2 Disk Partition Table Manipulation

Due to the development of computer science and OS, manual disk partition is less necessary than before for a casual user. Nevertheless, Linux provides necessary tools for disk partition table manipulation and they are introduced in this section.

8.2.1 Disk Partition

Disk partitioning refers to the action of creating one or more regions on secondary storage (e.g., disk) so that they can be managed separately, as if they are different “virtual disks”. An example of disk partitioning on a Windows PC would be to partition a *1TB* hard drive into *512GB* of C:\ drive and *512GB* of D:\ drive. Partitioned regions are logically separated. The partitions locations and sizes are stored in the disk partition table.

Some of the reasons for using disk partition include:

- Due to capability limitation, OS cannot handle a very large disk storage as a whole (this is less the case nowadays).
- Different types of files, for example system data and user data, can be stored separately for easy management.
- Different filesystems can be used on different partitions.
- Different partitions can be configured differently with unique settings.
- Sometimes partitioning can speed up hard disk accessing.

8.2.2 Disk Partition Table Manipulation

From the output of `lsblk` command earlier, it is clear that the system’s hard disk name is “`sda`”. To get a bit more details on this disk and its current partitioning status, use

```
$ sudo fdisk -l | grep sda
Disk /dev/sda: 931.51 GiB, 1000204886016 bytes, 1953525168 sectors
/dev/sda1      2048      4095      2048   1M BIOS boot
/dev/sda2     4096    1054719    1050624 513M EFI System
/dev/sda3  1054720  1953523711  1952468992 931G Linux filesystem
```

From the above result, it can be seen that the disk registered under `/dev/` (this is where the devices are represented by default) has been partitioned into 3 partitions, and the user space that can be further modified is `/dev/sda3` which is *931GB*.

There are variety of tools that can be used for disk partitioning. A common way of doing that is to use

```
$ sudo fdisk /dev/sda3
```

to enter the fdisk utility, and follow the wizard.

Other tools such as `cfdisk` and `parted` can also be used similarly for disk partition.

8.3 Mount, Unmount and Format a Partition

The hard disk can be formatted by partition, from where a new filesystem can be created. To format a partition, double check using `lsblk` to make sure that it is not mounted in the system. Use `sudo umount <partition name>` to unmount a partition.

Use `sudo mkfs` to format and create a new formatted filesystem, then use `mount` to mount it back to the OS. The mount of a partition needs to be recorded into `/etc/fstab` so that the OS would remember it after a reboot.



9

Service Control

CONTENTS

9.1 Service Control	85
---------------------------	----

...

9.1 Service Control

There are many services running in the background of the OS, some of which started by the OS while the other by the user. For example, *Apache service* might be used when the system is hosting a webpage. Other commonly used services include keyboard related services, bluetooth services, etc.

To quickly have a glance of the running services, use

```
$ systemctl --type=service
```

These services can be managed using service managing utilities such as **systemctl** and **service**. Some commonly used terminologies are concluded in Table 9.1 with explanations about their differences.

In short, **systemd** is the back-end service of Linux that manages the services. Both **systemctl** and **service** are tools to interact with **systemd** (and other back-end services) to manage the services. Generally speaking, **systemctl** is more straightforward, powerful and more complicated to use, while **service** is usually simpler and user-friendly.

Use the following commands to check the status of a service, and start, stop or reboot the service.

```
$ sudo systemctl status <service name>
$ sudo systemctl start <service name>
$ sudo systemctl stop <service name>
$ sudo systemctl restart <service name>
```

Use the following commands to enable and disable a service. An enabled service automatically starts during the system boot, and a disabled service does not.

TABLE 9.1

Commonly seen terminologies regarding service control.

Term / Tool name	Description
<code>systemd</code>	The <code>systemd</code> , i.e., <i>system daemon</i> , is a suite of basic building blocks for a Linux system that provides a system and service manager that runs as PID 1 and starts the rest of the system.
<code>systemctl</code>	The <code>systemctl</code> command interacts with the <code>systemd</code> service manager to manage the services. Contrary to <code>service</code> command, it manages the services by interacting with the <code>Systemd</code> process instead of running the <code>init</code> script.
<code>service</code>	The <code>service</code> command runs a pre-defined wrapper script that allows system administrators to start, stop, and check the status of services. It is a wrapper for <code>/etc/init.d</code> scripts, Upstart's <code>initctl</code> command, and also <code>systemctl</code> .

```
$ sudo systemctl enable <service name>
$ sudo systemctl disable <service name>
```

Use the following command to mask and unmask a service. A masked service cannot be started even using `systemctl start`.

```
$ sudo systemctl mask <service name>
$ sudo systemctl unmask <service name>
```

The `service` command can be used in a similar manner as follows.

```
$ sudo service <service name> status
$ sudo service <service name> start
$ sudo service <service name> stop
$ sudo service <service name> restart
```

Part III

Widely Used Services



10

Git

CONTENTS

10.1	Introduction	89
10.2	Setup	90
10.2.1	Installation	90
10.2.2	Configuration	90
10.3	Local Repository Management	91
10.3.1	Initialization of a Repository	91
10.3.2	Version Tracking	91
10.3.3	Branch Management	96
10.4	Remote Repository Management	98
10.5	<i>GitHub Actions</i>	101

Git is a distributed version-control system for tracking changes during the software development, and it can be used on cross platforms including Windows, Unix/Linux and MacOS. This chapter introduces the basic use of *Git* on a local Linux machine and with a remote repository host server such as *GitHub*. Continuous integration and continuous development (CI/CD), an important concept in nowadays software development and operation, is also briefly covered. For CI/CD, *GitHub Action* is introduced.

10.1 Introduction

Git, created by Linus Trovalds in 2005, is a distributed version-control system for tracking changes in source code and files. It is helpful with maintaining data integrity during the collaborative development of a software in distributed non-linear work flows. *Git* is free and open-source software under GNU general public license.

With *Git*, all computers participating in the software development store a copy of the full-fledged repository locally with complete history, and it can synchronize with a centralized remote server. *Git* uses “master” and “slave” branches to manage the concurrent development of different features of the project, where the master branch is the stable and shared repository among

everyone, and the slave branches are copies of the master branch where individual features can be developed. For a slave branch, once its developed feature is approved, it can be merged back to the master branch. A demonstration is given in Fig. 10.1.

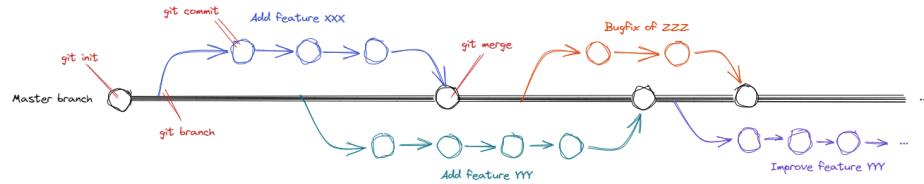


FIGURE 10.1
Git for software development management.

CLI is often used to manage a *Git* repository. For example, `git init` starts a new repository with an empty master branch, and `git branch <branch-name>` creates a new slave branch from the master branch, etc. Some of these commands are shown in Fig. 10.1 and more to be introduced later. Notice that a graphical user interface is also available to interact with *git*. However, in the scope of this notebook, command line is mostly used.

10.2 Setup

To use *Git*, follow the instructions to install and configure the software.

10.2.1 Installation

Git and its relevant documents can be obtained from its official website <https://git-scm.com/>. In many Linux distributions, *Git* is pre-installed. If *Git* is missing in the machine, follow the instructions on the official website to install it on the computer. Notice that the installation procedure may differ with different OS.

10.2.2 Configuration

Upon successful installation, it is recommended to use `git config` for some basic configurations. Notice that there are two types of configurations, namely the global configurations (apply to the machine and the user), and the repository configurations (apply to a particular *Git* repository). By default, the global level configurations are stored under `~/.gitconfig` and the reposi-

tory level configurations `./.git/config` in the repository. A good practice to edit these configurations is to use *Git* commands rather than editing the files directly.

To add user name and email to the global configuration, use

```
$ git config --global user.name '<user name>'  
$ git config --global user.email '<user email>'
```

To retrieve the global configuration, use

```
$ git config --global -l
```

To revoke a global configuration, use

```
$ git config --global --unset <configuration>
```

For example,

```
$ git config --global --unset user.name
```

removes the user name.

More details about `git config` can be found at <https://git-scm.com/docs/git-config>. Usually, the user name and email configurations are mandatory, as they are very useful information in developing collaborative projects.

10.3 Local Repository Management

In practice, repositories are managed locally and they may or may not have their remote associations known as upstreams. *Git* is used to manage local repositories and synchronize them with their associated upstreams. *Git* local repository management is introduced in this section. *Git* is able track versions and control branches.

10.3.1 Initialization of a Repository

Navigate to the project directory. Use the following command to create a new *Git* repository for the project.

```
$ git init
```

From this point forward, *Git* monitors everything that happens inside this directory and its subdirectories and tries to track any change to the files, unless otherwise configured specifically. Many CLI commands, for example `git status` to check the status of the repository, become available. More details are introduced later.

TABLE 10.1Different file status in a *Git* managed project.

Status	Description
Untracked	Newly added or renamed items in the project directory.
Modified (Unstaged)	Modified items from the last version that has not been registered in the staged area.
Modified (Staged)	Modified items from the last version that has been registered in the staged area. Notice that an untracked item can be staged directly, skipping “modified (unstaged)” step. Use <code>git add</code> to stage items.
Unmodified	Unmodified items from the last version. Use <code>git commit</code> to update a new version, which would change all the staged items into unmodified items.

10.3.2 Version Tracking

For simplicity, assume that there is only one branch in the repository, namely the master branch. Notice that when there are multiple branches, the version-tracking works the same for each and every branch in a separate and independent manner.

The mechanism behind version tracking is briefly introduced as follows. This helps the user to better understand how *Git* works and how the CLI commands are designed.

The project directory is split into two parts, outside `./.git/` the workspace, and inside `./.git/` the *Git* repository. The workspace has the up-to-date project contents and it is directly managed by the user, while *Git* repository is managed by *Git*. The user should not manage the repository directly unless using the *Git* interface.

Inside the *Git* repository are metadata of the workspace files such as which files have been changed since the last deployment, etc. It also stores a full back up of every historical versions of the project (with powerful compressing mechanisms, of course). It is worth mentioning that instead of recording the changes of a file from version to version, *Git* records the snapshot of the file in every version, unless it is left untouched between two consecutive versions.

Figure 10.2 gives a demonstration of how *Git* manages the project directory. *Git* classifies files in the workspace into different status, as shown in Fig. 10.2. A brief explanation of each types is given in Table 10.1. Detail explanations of “stage” and “commit” are given later.

Use `git status` to check the file status in the project. An example is given below.

```
$ git status
On branch master
```

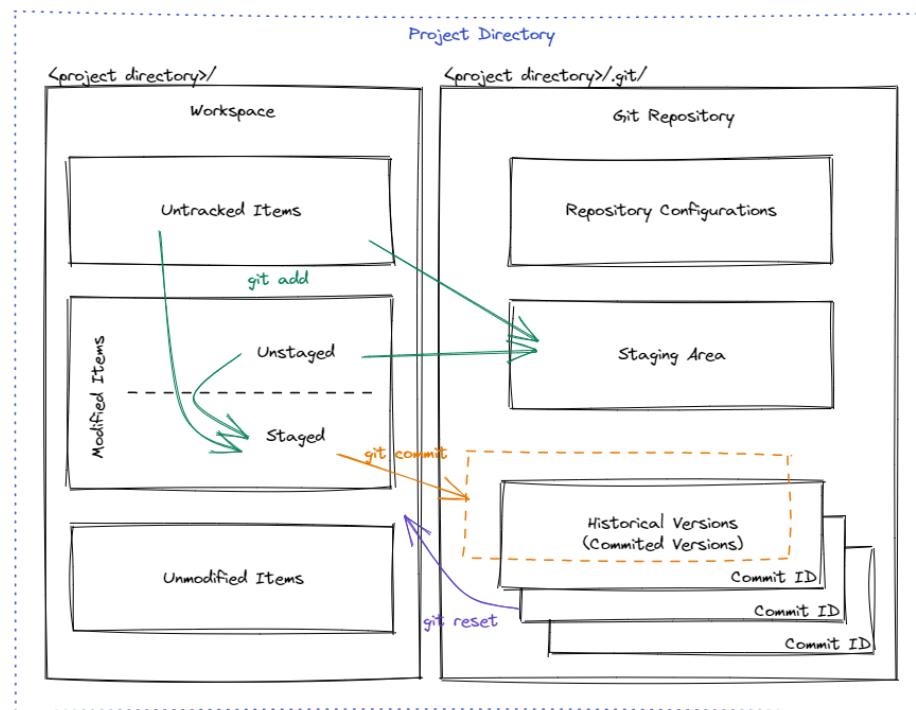


FIGURE 10.2

The project directory managed by *Git*.

```

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified: chapters/ch-software-management-advanced/ch.tex

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified: appendix/ap.tex
    modified: main.pdf

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    chapters/ch-software-management-advanced/figures/

```

where `ch.tex` is a modified (staged) item; `ap.tex` and `main.pdf` modified (unstaged) items, and `figures/` an untracked item.

It takes a two-step procedure to back up the project in the *Git* repository. In the first step, the user flags the changed items (either newly added, renamed, or modified) to be backed up in the next version. In the second step, the user actually backs up the items. The first and second steps are called “stage” and “commit” respectively. Notice that it is possible to run a single line of command to execute both steps, but logically it still takes two steps.

Git tracks the name and content of the items that the user has staged in the “staging area” as shown in Fig. 10.2. Think of staging items as taking a snapshot of the items. However, the snapshot at this stage is temporary and has not been backed up in the repository yet. The actual backup happens later when the staged items are committed. To stage an item, use

```
$ git add <item name>
```

which registers the item in the staging area, thus also changes its status from untracked or modified (unstaged) to modified (staged). If an item is modified after it has been staged, *Git* will distinguish the “staged portion” and “unstaged portion” of that item. If using `git status` to check its status, the item will be listed as both staged and unstaged. Unstaged items, either untracked or modified, will remain its status after the commit. Sometimes for convenience, `git add -A` can be used to add all untracked or modified items to the staging area.

Use `git commit` to commit the staged items and back them up in the repository as follows.

```
$ git commit [<item name>]
```

The above command commits the project and creates a version in the *Git* repository. It is possible to specify items, in which case *Git* only commits the specified items and leave the rest items as they are. A commit ID is automatically assigned to the commit. Notice that the user will be asked to provide a “comment message” with the commit, which should be used to briefly explain what has been changed in this commit.

A flag `-a` with `git commit` stages all changes made to the project, then implements the commit command. A flag `-m` simplifies the message recording process and allows the user to key in the message directly after the command. An example is given below.

```
$ git status
On branch master

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified: A Notebook on Linux/chapters/ch-software-management-
              advanced/ch.tex
    modified: A Notebook on Linux/main.pdf

no changes added to commit (use "git add" and/or "git commit -a")

$ git commit -am "add introduction to git command"
[master e2e977e] add introduction to git command
 2 files changed, 5 insertions(+), 4 deletions(-)

$ git status
On branch master

nothing to commit, working tree clean
```

To check the commit logs, i.e., all historical commits including their associated timestamps, authors, commit IDs and comment messages, use `git log` as shown in the example below. Notice that the commit logs can be very long. Only a few commits are given for illustration purpose.

```
$ git log
commit e3475e673d8c2a087de6b4423188c51e80af3e5d (HEAD -> master)
Author: sunlu <sunlu.electric@gmail.com>
Date:   Wed Aug 31 15:16:52 2022 +0800

    git

commit 3ab6d1473a7e48d4d890509ddb5a87274c023e6c
Author: sunlu <sunlu.electric@gmail.com>
Date:   Tue Aug 30 15:50:06 2022 +0800

    k8s

commit f6a1e3d779305e966602ecd7589c05f56dd2ad0f
Author: sunlu <sunlu.electric@gmail.com>
Date:   Mon Aug 29 16:31:18 2022 +0800

    more on docker and kubernetes
```

```
commit e2de5b28db7982f57d0ad51361d5161b884f2efe
Author: sunlu <sunlu.electric@gmail.com>
Date: Sun Aug 28 21:38:48 2022 +0800

add docker sections
```

where notice that HEAD is a reference that points to the latest commit in a branch. Filters can be added as optional arguments for the `git log` command. More details are given at <https://git-scm.com/docs/git-log>.

And finally, to restore to a previous commit version, use `git reset` or `git revert`. Notice that `git reset` and `git revert` can both be used to restore the workspace to a previous stage, but they differ significantly. In general, `git reset` “erases” the past commits, and it is often used in the case where the changes and commits to be reset have not been uploaded to the upstream or shared distributively. Command `git revert` on the other hand create a new commit that undoes all the changes, and it is often used when the changes and commits to revert are already distributed.

The command `git reset` is often used as follows.

```
$ git reset <option> <commit ID>
```

where `<option>` is often `--hard`, `--mixed` or `--soft`, and `<commit ID>` can be the ID of any commit in the git log, or for shorcut HEAD the latest commit, `HEAD^` or `HEAD^1` the second latest commit, `HEAD^2` the third latest commit, and so on.

The options `--hard`, `--mixed` and `--soft` work differently. All these options move `HEAD` to the specified commit, and remove all commits afterwards from the repository. The `--hard` option reverts the workspace back to when the specified commit happened (meaning that there would be no way to undo the reset command). Both `--mixed` and `--soft` do not change the workspace. The `--mixed` option leaves all the changes from the specified commit to today as unstaged, while `--soft` leaves them as staged. If no `--hard`, `--mixed` or `--soft` is given, `--mixed` will be used as the default option.

Notice that `git reset` does not help if the unwilling changes and commits to be reverted have already been pushed and synchronized to a remote server or to other collaborators. This is because `git reset` erases the history, and as a result `git` would think that the reset repository is “left behind” when it synchronizes with other repositories. Consequently, it will simply synchronizes the changes and commits back.

In such cases, consider using `git revert` instead. Syntax wise they work similarly. Instead of erase the history, `git revert` creates a new commit whose content is copied and pasted from a past commit.

10.3.3 Branch Management

“Branch” is one of the core features of *Git*, and it plays an important role in collaborative development of a project. There are two types of branches,

namely the local branch and the remote branch. The remote branch is often the upstream mirror of the local branch used for sharing and synchronizing the project development with others. The details will be introduced in Section 10.4. Only local branches are considered in this section.

To list down all the local branches, use

```
$ git branch
```

and the current working branch will be highlighted. The current working branch is also referred as the “head branch” or the “active branch”.

To create a new branch from the current branch, and to delete a branch, use

```
$ git branch <new branch name> [<commit ID>]  
$ git branch -d <branch name>
```

respectively. The optional `<commit ID>` when creating a new branch allows the user to create a branch on top of a specified historical commit instead of the latest commit.

To rename a branch, use

```
$ git branch -m [<old branch name>] <new branch name>
```

where if no `<old branch name>` is specified, the current working branch will be renamed.

To switch to a different working branch, use `git checkout` or `git switch` as follows.

```
$ git checkout <branch name>  
$ git switch <branch name>
```

Notice that when there are uncommitted changes in the current branch, *Git* may forbid the user to switch to another branch (the rules are too complicated to be explained here). Therefore, it is recommended to commit the changes before switching to a different branch. When switching to another branch, the workspace will change accordingly to the target branch.

To merge a branch back to the current branch, use

```
$ git merge <branch name>
```

and fill in the comments accordingly. For example, checkout to master branch, and use `git merge <slave branch name>` to merge the features in the slave branch to the master branch. This often happens when the slave branch has finished developing and testing a new feature and is prepared to add the feature to the stable master branch. By default, *Git* automatically deletes the merged branch. This behavior can be changed in the configuration.

Pull Request

In a collaborative project, the master branch is managed very carefully because everything on that branch will affect the project deeply. There is often a group of senior developers who manage the master branch. Any code to be added to the master branch must be checked by one or a few of them.

When a slave branch wants the features developed on that branch to be merged to the master branch, the owner of the slave branch needs to raise a “pull request”. As its name suggests, it requests the master branch to pull the updates from the slave branch. The master branch managers will then view and check the slave branch, making sure that there is no bugs, low-quality codes or conflicts, and then approve the merging.

An alternative way of integrating two branches is `git rebase`, which “rewires” the two branches into a linear single branch. Use `git rebase` as follows.

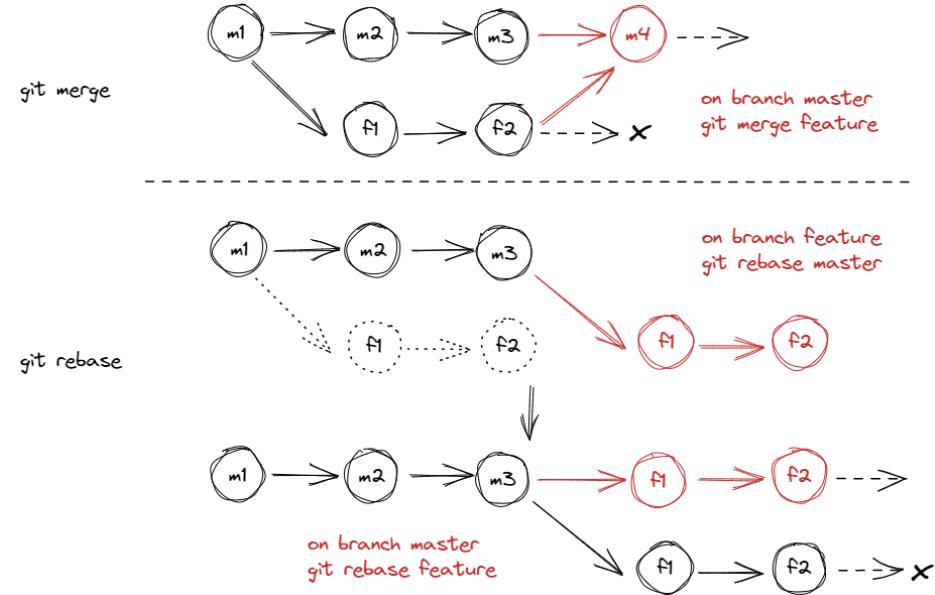
```
$ git rebase <branch name>
```

A demonstrative Fig. 10.3 explains the differences between `git merge` and `git rebase`. It is clear from Fig. 10.3 that by using `git rebase`, all feature commits are integrated into the master repositories to form a single repository tracking line, which differs largely from `git merge`.

An intuitive way to understand `git rebase` is given as follows. Imagine two branches *A* and *B* that have diverged from the same commit *O*. On branch *A*, execute `git rebase B`. Here is a step-by-step breakdown of what happens:

1. Identify common ancestor of the current branch *A* and the rebase branch *B*, which is commit *O* in this example.
2. Back up commits happened on branch *A* since commit *O*. Assume they are commits A_1, A_2, \dots, A_n .
3. Reset branch *A* to Branch *B* and *A* now starts from the tip of *B*.
4. Reapply commits A_1, A_2, \dots, A_n on the reset *A* in a sequential manner, potentially leading to conflicts that need to be resolved manually.

After the rebase operation, the history of branch *A* appears as if it was developed sequentially from the end of branch *B*, effectively integrating the changes of *B* into *A* in a linear history.

**FIGURE 10.3**

Two approaches of integrating branches, `git merge` VS `git rebase`.

10.4 Remote Repository Management

In collaborative projects, the repository and branches are stored and shared in one or more cloud servers. Each developer can interact with the repository on the cloud by pulling changes or pushing updates. The developer does not directly make changes to the repository or branches on the cloud. Instead, he clone the repository to his local machine and manipulate branches locally. The local repository and branches are then synchronized with the cloud repository and branches using *Git* commands. This is known as remote repository management and it is introduced in this section.

Depending on the cloud servers that host the remote repository, different services might be available. In the scope of this notebook, *GitHub* is considered as the remote repository host. *GitHub* is an internet hosting service for software development and sharing using *Git*. A user can create, manage, share and co-develop remote repositories on *GitHub*. Notice that *GitHub* is not the only place to host remote repositories. Some alternatives include *GitLab* and *Gitee*. Using *Git*, it is even possible to build a repository hosting server on premises using a regular computer. In this notebook, only *GitHub* is studied.

An `https` URL is associated with each remote repository on *GitHub*, for example `https://github.com/torvalds/linux.git` for Linux kernel. This URL can

be used to download the remote repository to a local machine, or to link (synchronize) a local repository with that remote repository.

Consider the case where there is already a remote repository, either under the user's *GitHub* account or from the community. To download a clone of an existing repository to the local machine, use

```
$ git clone <repository URL> [<local directory>]
```

and to maintain the local repository up-to-date with the remote repository, regularly use

```
$ git remote update  
$ git pull
```

to synchronize the local repository with the remote one.

Should I use git pull?

One may argue whether it is a good idea to use `git pull`. Although convenient, `git pull` is an integration of two commands, `git fetch` and `git merge` (or `git rebase`) executed sequentially. It may be "safer" to manually run the two commands separately.

The local commits can also be pushed to the remote repository using

```
$ git push
```

if the remote repository is under the user's account, or if the owner of the remote repository gives the user the permission. Notice that when pushing updates to the remote repository, *GitHub* may require login credentials, for example username and password pair or SSH key, for permission checks.

Consider another case where there is already a well developed local repository. An empty remote repository has just been created on *GitHub* which is to be setup to mirror the local repository. To link the remote and local repositories, navigate to the local repository and use

```
$ git remote add <remote repository name> <repository URL>
```

which will register the remote repository URL in the local configuration file. A commonly used remote repository name is "origin". The next step is to map the local current working branch with a branch in the remote repository, which would be used as the default source/target branch when running `git pull` and `git push`. This can be done as follows.

```
$ git branch --set-upstream-to <remote repository name> <remote  
repository branch>
```

Notice that this step is not required if the project is cloned from a remote repository using `git clone` introduced earlier, as it will automatically setup the upstream using the source of the clone.

From there on, use `git push` and `git pull` normally.

By assigning upstream branch to a local branch and `git push` the local branch to the upstream, *Git* will update or create the remote branch accordingly. A remote branch can be deleted with a `-d` flag while using `git push`.

10.5 GitHub Actions

GitHub Actions is essentially a tool that allows the user to define a pipeline of actions when something changed in the *GitHub* remote repository, for example a pull request is raised, a new commit is pushed, etc. *Actions* is useful as part of CI/CD. More details are given in Appendix E.

|||||| HEAD



11

Database

CONTENTS

11.1	Relational Database	103
11.2	Non-Relational Database	105
11.3	Structured Query Language	106
11.3.1	RDB Naming Conventions	106
11.3.2	General Introduction to SQL	107
11.3.3	General Syntax	107
11.3.4	Database Manipulation	111
11.3.5	Table Manipulation	112
11.3.6	Row Manipulation	114
11.3.7	Query	115
11.3.8	Trigger	123
11.3.9	SQL Demonstrative Example	124
11.4	General Ways to Connect to RDB	129

Database and database manage systems (DBMS) are introduced in this and consecutive chapters. Both relational and non-relational databases are covered. Tools and languages to manipulate databases, such as structured query language (SQL) for relational databases, are introduced.

11.1 Relational Database

Database, in a broad view, refers to an organized collection of data of any format. In this sense, any file format that hosts information in a meaningful and explainable way, such as *CSV*, *XML* and *JSON*, is a database. These file formats often work fine when the data is stored in a centralized manner and its size small.

As the data size grows, the robustness and efficiency of storing and retrieving data become challenging, and different database models have been proposed to tackle it. Dedicated software, namely DBMS, is developed to manage and maintain the database and provide an interface for the users to create, retrieve, update and delete data. Different database models require dif-

ferent database engines and DBMS, and different DBMS may provide different interfaces for the users.

There are many database models available in the market. The most widely seen database models can be divided into two categories, namely the relational databases (RDB) and the non-relational databases.

Relational database was proposed in 1970s by IBM. Some important features of a relational database include the following.

- Structure the data as “relations”, which is a collection of tables, each consisting of a set of rows (also known as tuple/record) and columns (also known as attribute/field).
- For each table, a primary key, either being a column or a combination of several columns, is defined that can uniquely distinguish a row from others.
- Provide relational operations that manipulate the data in the tables, for example, joining tables together and aligning them using an attribute.

SQL, a domain-specific language, can be used in managing RDB and interfacing relational DBMS. Most commercialized RDB management systems (RDBMS) adopt SQL as the query language. There are alternatives, but they are rarely used compared to SQL. There is a evolving standard on what operations should an RDBMS support using SQL. The latest version as of this writing is **SQL:2023**.

Examples of relational databases include *Oracle*, *MySQL*, *Microsoft SQL Server*, *MariaDB*, *IBM Db2*, *Amazon Redshift*, *Amazon Aurora* and *PostgreSQL*.

An example of a table used in RDB is given in Table 14.1. The table has a name `user` and 4 attributes `user_id`, `user_email`, `membership` and `referee_id`. A table should have an attribute (or a set of attributes) defined as the primary key. In this example, `user_id` is assigned to be the primary key as denoted by the asterisk. The primary key is used to uniquely identify a row in the table. A primary key can consist a single column or multiple columns. When a primary key is composed of multiple columns, it is called a composite key. A table should have one and only one primary key.

Depending on the meaning behind the primary key, it can be divided into two types, namely surrogate key and natural key. A surrogate key is like a serial number or an incremental ID which serves only for recording and distinguishing rows and does not have a physical meaning. In contrast, a natural key such as a timestamp, email, citizenship IC number, reflects some meaningful information in the real world.

A foreign key is the attribute(s) that link a table to another table. It is often the primary key of another table that in some way links to this table. For example, consider another table `membership_type` as defined in Table 14.2. In the first Table 14.1, column `membership` can be a foreign key which points to `membership_type` in the second Table 14.2.

TABLE 11.1

An example of a relational database table.

user			
user_id*	user_email	membership	referee-id
sunlu	sunlu@xxx.com	premium	NULL
xingzhe	xingzhe@yyy.com	basic	sunlu
...

TABLE 11.2

A second database table in the example.

membership		
membership_type	monthly_price	annual_price
none	0	0
basic	5	50
premium	10	80

The introducing of foreign key helps to maintain the consistency and integrity of the database. For example, when adding a new member to Table 14.1, the DBMS will first check whether the membership type is registered in Table 14.2. If not, the insert operation will be rejected. This guarantees that all registered members have valid membership types.

Notice that a table can have multiple foreign keys. The foreign key can not only relate a table to another, but also relate a table to itself. For example, the “referee-id” attribute in Table 14.1 could be the foreign key that links to “user-id” of the same table.

11.2 Non-Relational Database

Non-relational databases (NoSQL databases) gained their popularity in the 2000s. In contrast to RDBs, NoSQL databases do not store data in tables like an RDB, but in key-value pairs, graphs, documents, or other formats. They are more flexible, efficient and easier to use in some applications than an RDB. Examples of NoSQL databases include *Redis*, *Azure CosmosDB*, *Oracle NoSQL Database*, *Amazon DynamoDB*, *MongoDB*, *AllegroGraph* and many more.

Unlike SQL which applies to almost all RDBMS, there is no universally adopted language for NoSQL DBMS. Each NoSQL DBMS often has its unique query language tailored to its specific data model. We use “NoSQL” to refer to a collective set of (very different and not interchangeable) languages used for

NoSQL databases management. It is impossible to cover all the different types of NoSQL databases and their corresponding DBMS manipulation languages in this notebook. Only brief introductions to some example databases are given in later chapters.

Database services, both RDB and NoSQL, have become critical to our daily life and they are massively deployed on servers. In many applications they work together to deliver the service.

11.3 Structured Query Language

As of this writing, RDB is the most commonly used database, and SQL is its standard manipulating and querying language. They are introduced below.

11.3.1 RDB Naming Conventions

It is recommended that the following naming conventions be applied to databases, tables and columns.

- General rules:
 - Use natural collective terms over plurals, for example, “staff” over “employees”.
 - Use only letters, numbers, and underscores.
 - Begin with a letter and may not end with an underscore.
 - Avoid using abbreviations unless commonly understood.
 - Avoid using prefixes.
- Table:
 - Do NOT use the same name for a table and one of its columns.
 - Do NOT concatenate two table names to create a third relationship table.
- Column:
 - Use singular name for columns.
 - Avoid using over simplified terms such as “id”.
 - Use only lowercase if possible.
- Alias:
 - Use keyword AS to indicate an alias.

- The correlation name should be the first letter of each word of the object name.
- If there is already the same correlation name, append a number.
- Stored procedure: always contain a verb in the name of a stored procedure.
- Uniform suffixes:
 - `_id`: primary key.
 - `_status`: flag values.
 - `_total`: the total number of a collection of values.
 - `_num`: a number.
 - `_date`: a date.
 - `_name`: the name of a person or a product.

11.3.2 General Introduction to SQL

SQL is the most widely used language for interacting with DBMS for data query and maintenance. SQL is very powerful and flexible in its full capability, and it is hardly possible to cover everything in one section. Hence, in this section only the basic SQL operations are introduced.

Notice that the support of different DBMS to SQL may differ slightly. This is because an DBMS may (in fact, very likely) fail to adopt everything in the latest SQL standard. However, most of the commands especially the widely used ones such as creating tables, inserting rows and most of the querying, shall be universally consistent.

SQL is a hybrid language consisting of the following 4 sub-languages.

- Data query language: query information and metadata of a database.
- Data definition language: define database schema.
- Data control language: control user access and permission to a database.
- Data manipulation language: insert, update and delete data from a database.

SQL supports variety of data types, and different DBMS may cover slightly different data types. Some of the most commonly used data types are summarized in Table 14.3. Nowadays, many DBMS supports more complicated types such as objects (the associated database is also known as the object-relational database), and many more. For the full list of data types that a DBMS supports, check the manuals and documents of that DBMS.

SQL defines reserved keywords for database manipulation. The keywords have specific meanings and cannot be used as user-defined variable names. Commonly used SQL keywords are summarized in Tables 14.4, 14.5 and 14.6.

TABLE 11.3

Widely used SQL data types.

Data Type	Description
INT/INTEGER	Integer, with a range of -2147483648 to 2147483647. When marked “UNSIGNED”, the range becomes 0 to 4294967295. Some relevant data types are TINYINT, SMALLINT, MEDIUMINT, and BIGINT, which have different ranges.
DEC/DECIMAL(size,d)	An exact fixed-point number. The total number of digits and the number of digits after decimal point are specified by “size” and “d”, respectively. Some relevant data types are DOUBLE(size,d), which can also be used to specify a floating point. Notice that DEC/DECIMAL is usually preferable in most occasions.
CHAR(size)	A fixed length string with the specified length in characters.
VARCHAR(size)	A variable length string, with the specified maximum string length in characters.
BOOL/BOOLEAN	This is essentially a 1-digit integer, where 0 stands for “false” and otherwise “true”.
BLOB	A binary large object with maximum 65535 bytes.
DATE	A date by format “YYYY-MM-DD”.
TIME	A time by format “hh:mm:ss”.
DATETIME	A combination of date and time by format “YYYY-MM-DD hh:mm:ss”.
TIMESTAMP	A timestamp that measures the number of seconds since the Unix epoch. The format is “YYYY-MM-DD hh:mm:ss”. Unlike DATETIME which is meaningful only if the timezone is known, TIMESTAMP does not rely on timezone.

TABLE 11.4

Widely used SQL keywords (part 1: names).

Keyword	Description
CONSTRAINT	A constraint that limits the value of a column.
DATABASE	A database.
TABLE	A table.
COLUMN	A column (attribute, field) of a table.
VIEW	A view, which is a virtual table that does not store data by itself but only reflects the base tables data.
INDEX	An index, which is a pre-scan of specific column(s) of a table and can be used to speed up future queries related to the column(s). Notice that unlike a view, an index needs to be stored together with the table.
PRIMARY KEY	The primary key of a table.
FOREIGN KEY	A foreign key defined in a table that links to a (different) table.
PROCEDURE	A procedure that defines a list of database operations to be executed one after another

TABLE 11.5

Widely used SQL keywords (part 2: actions).

Keyword	Description
CREATE	Create a database (CREATE DATABASE), a table (CREATE TABLE), a view (CREATE VIEW), an index (CREATE INDEX) or a procedure (CREATE PROCEDURE).
ADD	Add a column in an existing table, or a constraint to an existing column.
ALTER	Modify columns in a table (ALTER TABLE), or a data type of a column (ALTER COLUMN).
SET	Specify the columns and values to be updated in a table.
DROP	Delete a column (DROP COLUMN), a constraint (DROP CONSTRAINT), a database (DROP DATABASE), an index (DROP INDEX), a table (DROP TABLE), or a view (DROP VIEW).
CHECK	Define a constraint that limits the value that can be placed in a column.
DEFAULT	Define a default value for a column.
INSERT INTO	Insert a new row into a table.
UPDATE	Update an existing row in a table.
DELETE	Delete a row from a table.
EXEC	Executes a stored procedure.

TABLE 11.6

Widely used SQL keywords (part 3: queries).

Keyword	Description
SELECT	Query data from a database. Relevant combinations are SELECT DISTINCT which returns only distinct values; SELECT INTO which copies data from one table into another; SELECT TOP which returns part of the results.
AS	Assign an alias to a column or table.
FROM	Specify the table where the query is run.
WHERE	Filter results that fulfill a specified condition.
IN	Specify multiple values in a WHERE clause.
AND	Select rows where both conditions are true.
OR	Select rows where either condition is true.
ALL	Return true if all followed sub-query values meet the condition.
ANY	Return true if any followed sub-query value meet the condition.
BETWEEN	Select values within a given range.
ORDER BY	Sort the results in ascending or descending order.
JOIN	Join tables for query. Relevant combinations are OUTER JOIN, INNER JOIN, LEFT JOIN and RIGHT JOIN.
EXISTS	Tests for the existence of any record in a sub-query.
GROUP BY	Groups the result set when using aggregate functions (COUNT, MAX, MIN, SUM, AVG).
UNION	Combines the result sets of multiple select statements.

11.3.3 General Syntax

Notice that different DBMS may use slightly different syntax for the same or similar function. In the rest of the chapter, unless otherwise mentioned, MySQL/MariaDB syntax is used.

All SQL commands shall end with a semicolon “;”.

The programming of SQL shall follow the following common practices wherever possible. This helps to maintain the good quality and portability of the code.

- Use standard SQL functions over user-defined functions wherever possible for better portability.
- Do NOT use object-oriented design principles in SQL and database schema wherever possible.
- Use UPPERCASE for keywords.
- Use /*<comments>*/ to add comments to the code, otherwise precede comments with -- <comments> and finish them with a new line.

The naming of database, tables and columns shall follow conventions introduced in Section 14.3.1.

During the coding, follow the following rules.

- Use spaces to align the codes.
- Use a space before and after equals (=), and after commas (,).
- Use BETWEEN and IN, instead of combining multiple AND and OR clauses.

When creating a table, follow the following rules.

- Choose standard SQL data types wherever possible.
- Specify default values and set up constraints, and put them close to the declaration of the associated column name.
- Assign primary key carefully and keep it simple.
- Specify the primary key first right after the CREATE TABLE statement.
- Implement validation. For example, for a numerical value, use CHECK to prevent incorrect values.

11.3.4 Database Manipulation

To list down all the databases running on the server, use

```
SHOW DATABASES;
```

To create a database, use

```
CREATE DATABASE <database-name>;
```

To select a database, use

```
USE <database-name>;
```

To delete a database, use

```
DROP DATABASE <database-name>;
```

11.3.5 Table Manipulation

Tables are the fundamental components in an RDB. An example of creating a table using SQL is given below

```
CREATE TABLE <table_name> (
    PRIMARY KEY (<column_name>),
    <column_name_1> <data-type> <constraint>,
    <column_name_2> <data-type> <constraint>,
        CONSTRAINT <constraint-name-1>
        CHECK(<constraint-rule>),
        CONSTRAINT <constraint-name-2>
        CHECK(<constraint-rule>)
);
```

where in this demonstrative table, 2 columns and 2 constraints are defined.

The **<constraint>** that comes after the data type of a column is used to set an additional restriction to the data in the table. When such restriction is violated, an error would raise to stop the operation. For example, if NOT NULL is set as a constraint, then when inserting a row to the table later, the user cannot input NULL for that specific column. Notice that the “primary key” can also be set as a constraint named PRIMARY KEY using this syntax, although it is a better practice to use PRIMARY KEY (<column-name>).

Commonly used such constraints are summarized into Table 14.7. As shown by Table 14.7, a default value can be assigned to a column by using the DEFAULT <value> constraint. When inserting a new row, the column of that row will be assigned to its default value if no other value is assigned. If no such statement is provided for a column, its default value is NULL.

Upon creation of a table, its basic schema can be reviewed using

```
DESCRIBE <table-name>;
```

To list down existing tables, use

```
SHOW TABLES;
```

TABLE 11.7

Commonly used constraints.

Constraint	Description
NOT NULL	Not allowed to be NULL.
UNIQUE	Not allowed to have duplicated values.
PRIMARY KEY	Set as primary key, thus, must be not NULL and must remain unique.
FOREIGN KEY	Set as foreign key.
DEFAULT <value>	Set a default value.
AUTO_INCREMENT = <value>	Each time a new row is inserted and NULL or 0 is set for this column, instead of set the column to NULL or 0, automatically generate the next sequence number. The starting value is defined by <value> which by default is 1.

To delete a table, use

```
DROP TABLE <table-name>;
```

To edit the column of a table, use either of the following

```
ALTER TABLE <table-name>
ADD <column-name> <data-type>; -- add new column
ALTER TABLE <table-name>
DROP COLUMN <column-name>; -- drop column
ALTER TABLE <table-name>
RENAME COLUMN <old-name> TO <new-name>; -- rename column
ALTER TABLE <table-name>
MODIFY COLUMN <column-name> <data-type>; -- modify column data type (
    depending on DBMS, syntax may differ)
```

or

```
ALTER TABLE <table-name>
ADD CONSTRAINT <constraint-name> CHECK(<constraint-rule>); -- add
    constraint
ALTER TABLE <table-name>
DROP CONSTRAINT <constraint-name>; -- drop constraint
```

Notice that it is possible to change the primary key using the above syntax because essentially the primary key is treated as a constraint named “primary”. It is not recommended to do so in general.

The foreign key is a key used to point to another table, in many cases other table’s primary key. Therefore, the foreign key can be nominated only after the other table has been created. Declare foreign key upon creation of a table as follows. As mentioned, to do this, the other tables must be created beforehand. Two methods to declare a foreign key are shown below.

```

CREATE TABLE <table-name> (
    PRIMARY KEY (<column-name>),
    <column-1> <data-type> <constraint>,
    <column-2> <data-type> <constraint>,
        CONSTRAINT <constraint-name-1>
        CHECK(<constraint-rule>),
        CONSTRAINT <constraint-name-2>
        CHECK(<constraint-rule>),
    FOREIGN KEY (<column-name>) REFERENCES <target-table-name>(<target-
        column-name>), -- method 1
        CONSTRAINT <constraint-name-3>
    FOREIGN KEY (<column name>)
        REFERENCES <target-table-name>(<target-column-name>) --
            method 2
);

```

where, as can be seen, foreign key is treated as a constraint in the table.

To define a foreign key in an existing table, use

```

ALTER TABLE <table-name>
ADD FOREIGN KEY (<column name>) REFERENCES <referred-table-name>(<
    referred-column-name>); -- one way
ALTER TABLE <table-name>
ADD CONSTRAINT <constraint-name> FOREIGN KEY (<column name>) REFERENCES
    <referred-table-name>(<referred-column-name>); -- another way

```

To drop a foreign key, use

```

ALTER TABLE <table-name>
DROP CONSTRAINT <constraint-name>;

```

There are variety ways of checking the constraints names of a table. An example is given below.

```

SELECT TABLE_NAME, CONSTRAINT_TYPE, CONSTRAINT_NAME
FROM information_schema.table_constraints
WHERE table_name=<table-name>;

```

One thing to notice is that upon creation of a foreign key, the referred column becomes the “parent” and the foreign key becomes a “child”. As long as the child exists, the parent cannot be removed from its table. This helps to protect the schema of the database. Should there be any quest to break the schema, this restriction can be overwritten. When defining the foreign key, add an additional claim “ON DELETE SET NULL” or “ON DELETE CASCADE” as follows.

```

FOREIGN KEY (<column name>) REFERENCES <referred-table-name>(<referred-
    column-name>) ON DELETE SET NULL
FOREIGN KEY (<column name>) REFERENCES <referred-table-name>(<referred-
    column-name>) ON DELETE CASCADE

```

in the first scenario, the child foreign key will be set to NULL, while in the second scenario, the child relevant rows will be removed.

11.3.6 Row Manipulation

To insert a row into a table, use

```
INSERT INTO <table-name> VALUES (<content>, <content>, ...);
```

where the contents shall follow the field sequence as shown by the DESCRIBE <table-name> command. To specify the column name while inserting a row, use

```
INSERT INTO <table-name>(<column-name>, <column-name>, ...) VALUES (<content>, <content>, ...);
```

Notice that it is also possible to populate multiple rows of a table using one command as follows.

```
INSERT INTO <table-name>(<column-name>, <column-name>, ...)
VALUES (<content>, <content>, ...),
       (<content>, <content>, ...),
       (<content>, <content>, ...);
```

where 3 rows are inserted into the table.

Notice that if a foreign key bound exists between two tables, when inserting a row to the child table, the foreign key value of this row must already be defined in the parent table.

Use the following command to query all items in a table, which can be used to check whether the row is added to the table correctly.

```
SELECT * FROM <table-name>;
```

To modify the attributes of specific row(s), use

```
UPDATE <table-name>
SET <column-name> = <value>, ...
WHERE <filter-criteria>;
```

where <filter-criteria> is used to filter the rows to which the update is carried out. Commonly used filter criteria are a set of <column-name> = <value> separated by AND and OR. The filter criteria can be set very flexibly and more details are given in later sections. Notice that it is possible to change multiple column values together, by stacking multiple <column-name> = <value> separated by “,”. Similarly, to delete rows from a table, use

```
DELETE FROM <table-name>
WHERE <filter-criteria>;
```

Notice that if filter criteria is not specified, i.e., if WHERE is missing, all items in the table will be affected.

11.3.7 Query

A typical query looks like the following and it returns the data in a table-like format.

```
SELECT <column-or-statistics>
FROM <table-name-or-combination>
GROUP BY <column-name>
WHERE <filter-criteria>
ORDER BY <column-name>, ...
LIMIT <number>;
```

where

- **<column-or-statistics>** describes the columns to be returned, and specifies the returning information format.
- **<table-name-or-combination>** describes the source of the information, either being a table, or a joint of multiple tables.
- **GROUP BY <column-name>** groups rows with the same value of the specified column into “summary rows”.
- **<filter-condition>** defines the filter criteria and only rows meet the criteria are returned.
- **ORDER BY <column-name>** allows the items to be returned in a specific order based on ascending/descending order. It is worth mentioning that the **<column-name>** here does not need to appear in the selected returns, and it can be multiple columns separated by “,”. Use **ASC** (default) or **DESC** after each **<column-name>** to specify ascending or descending order.
- **LIMIT <number>** restricts the maximum number of rows to be returned.

Notice that **SELECT** and **FROM** statements are compulsory in all queries, **WHERE** statement very widely used, and other statements optional case by case.

More details are given below.

The statement **<column-or-statistics>** mainly controls the information to be returned. Commonly seen selected items in **<column-or-statistics>** are summarized as follows.

- ***** (asterisk): return all columns.
- **<column-name>**: return selected columns. When multiple fields are returned, use bracket (**<col1>**, **<col2>**, ...). The same applies to other return formats below.
- **<table-name>. <column-name>**: return selected columns, and to avoid ambiguity, specify table name with the column name. This is useful when the source of data is a joint of multiple tables, some of which share the same column name.
- **<column-name> AS <alias>**: return selected columns, and use alias in the returns.
- **DISTINCT <column-name>**: return only distinct rows.

- COUNT(), SUM(), MIN(), MAX(), AVG(): return aggregate function of a column instead of all the items in that column. They can be used along with DISTINCT, for example, COUNT(DISTINCT <column-name>, ...).
- Simple calculations to the above result, for example $1.5 * <\text{column-name}>$. Commonly used arithmetic operations are +, -, *, /, %, DIV (integer division).

When the column is of date time type or interval type, it is possible to use EXTRACT() to select a field of the timestamp or interval. For example,

```
SELECT EXTRACT(YEAR FROM birthday) AS year FROM customer;
```

would look into table `customer`, focus on column `birthday` which should be a datetime type, and extract only `year` from the datetime to return the result.

The statement <table-name-or-combination> mainly indicates the source table(s). It can be a single table, a joint of multiple tables, or a nest query. More details about joint of multiple tables are illustrated below.

Consider the following example, where two tables are given as follows.

```
> SELECT * FROM test;
+-----+-----+-----+
| test_id | value_1 | value_2 |
+-----+-----+-----+
|      1 |     a |      10 |
|      2 |     a |      20 |
|      3 |     a |      30 |
|      4 |     b |     100 |
|      5 |     b |     200 |
|      6 |     c |    1000 |
|      7 |     c |    2000 |
+-----+-----+-----+

> SELECT * FROM test_join;
+-----+-----+-----+-----+
| test_join_id | value_1 | value_2 | value_3 |
+-----+-----+-----+-----+
|       a      |     10 |     99 | alpha   |
|       b      |    100 |    999 | bravo   |
|       d      |  10000 | 99999 | delta   |
+-----+-----+-----+-----+
```

There are different types of joins, namely “inner join” (or “join”), “left join”, “right join” and “cross join”. They are introduced as follows.

The most intuitive join is the cross join. It returns everything in the two tables like a Cartesian product (that explains why cross join is also called Cartesian join), where the total number of columns are the sum of two tables, the number of rows the product of two tables, as shown below.

```
> SELECT * FROM test CROSS JOIN test_join;
```

test_id	value_1	value_2	test_join_id	value_1	value_2	value_3
1 a	10 a		10 a	99 alpha		
1 a	10 b		100 b	999 bravo		
1 a	10 d		10000 d	99999 delta		
2 a	20 a		10 a	99 alpha		
2 a	20 b		100 b	999 bravo		
2 a	20 d		10000 d	99999 delta		
3 a	30 a		10 a	99 alpha		
3 a	30 b		100 b	999 bravo		
3 a	30 d		10000 d	99999 delta		
4 b	100 a		10 a	99 alpha		
4 b	100 b		100 b	999 bravo		
4 b	100 d		10000 d	99999 delta		
5 b	200 a		10 a	99 alpha		
5 b	200 b		100 b	999 bravo		
5 b	200 d		10000 d	99999 delta		
6 c	1000 a		10 a	99 alpha		
6 c	1000 b		100 b	999 bravo		
6 c	1000 d		10000 d	99999 delta		
7 c	2000 a		10 a	99 alpha		
7 c	2000 b		100 b	999 bravo		
7 c	2000 d		10000 d	99999 delta		

where notice that CROSS JOIN can be replaced by a comma “,”.

In this example, value_1 from table test and test_join_id from table test_join are corresponding with each other. In this context, the rows with inconsistent test.value_1 and test_join.test_join_id is meaningless and shall be removed. This can be achieved by the following code.

test_id	value_1	value_2	test_join_id	value_1	value_2	value_3
1 a	10 a		10 a	99 alpha		
2 a	20 a		10 a	99 alpha		
3 a	30 a		10 a	99 alpha		
4 b	100 b		100 b	999 bravo		
5 b	200 b		100 b	999 bravo		

test_id	value_1	value_2	test_join_id	value_1	value_2	value_3
1	a	10	a	10	99	alpha
2	a	20	a	10	99	alpha
3	a	30	a	10	99	alpha
4	b	100	b	100	999	bravo
5	b	200	b	100	999	bravo

and this is equivalent to inner join (or simply, join)

test_id	value_1	value_2	test_join_id	value_1	value_2	value_3
1	a	10	a	10	99	alpha
2	a	20	a	10	99	alpha
3	a	30	a	10	99	alpha
4	b	100	b	100	999	bravo
5	b	200	b	100	999	bravo

where `ON (<table1.column-name> = <table2.column_name>)` is used to indicate the association. The number of columns remain unchanged compared with cross join, but the number of rows is reduced.

From table `test` perspective, its rows regarding `value_1` equals to “a” and “b” are fully included in the inner join results. However, the two rows regarding `value_1=c` is omitted. This is because there is no corresponding row in the other table `test_join` with `test_join_id=c`.

It is possible to “prevent” information loss from table `test` by adding these two rows back, with all the columns from table `test_join` filled with `NULL`. This can be done by left join as follows.

test_id	value_1	value_2	test_join_id	value_1	value_2	value_3
1	a	10	a	10	99	alpha
2	a	20	a	10	99	alpha
3	a	30	a	10	99	alpha
4	b	100	b	100	999	bravo
5	b	200	b	100	999	bravo
6	c	1000	NULL	NULL	NULL	NULL
7	c	2000	NULL	NULL	NULL	NULL

One can think of this as temporarily adding a new row to `test_join` with `test_join_id=c` and everything else NULL, before the inner joining.

The same idea applies to right join as well, as shown below.

```
> SELECT * FROM test RIGHT JOIN test_join ON (test.value_1 = test_join.
   test_join_id);
+-----+-----+-----+-----+-----+-----+
| test_id | value_1 | value_2 | test_join_id | value_1 | value_2 |
|         | value_3 |           |             |         |           |
+-----+-----+-----+-----+-----+-----+
|     1 | a      |    10 | a          |    10 |    99 | alpha   |
|     2 | a      |    20 | a          |    10 |    99 | alpha   |
|     3 | a      |    30 | a          |    10 |    99 | alpha   |
|     4 | b      |   100 | b          |   100 |  999 | bravo   |
|     5 | b      |   200 | b          |   100 |  999 | bravo   |
|  NULL | NULL   |  NULL | d          | 10000 | 99999 | delta   |
+-----+-----+-----+-----+-----+-----+
```

Some DBMS supports “outer join”, which is basically a union of the left and right join results. More about union is introduced later.

The statement <filter-condition> applies filtering to the results. Commonly seen filter criteria <filter-condition> are summarized as follows.

- <column-name> = <value>, where = can be replaced by < (less than), <= (less than or equal to), > (larger than), >= (larger than or equal to) and <> (not equal to).
- <column-name> IN (<value>, <value>, ...)
- <column-name> BETWEEN <value> AND <value>
- <column-name> LIKE <wildcards>, which compares the column value (usually a string) with a given pattern.
- A combination of the above, with AND and OR joining everything together.

A bit more about wildcard query is introduced as follows. A wildcard character is a “placeholder” that represents a group of character(s). Most commonly used wildcard characters in the SQL context include

- _: any single character.
- %: any string of characters (including empty string).
- [<c1><c2> ...]: any single character given in the bracket.
- ^[<c1><c2>...]: any single character not given in the bracket.
- [<c1>-<c2>]: any single character given within the range in the bracket.

Wildcard query can be applied to both CHAR and DATE/TIME types, as they can all be characterized as strings.

The **GROUP BY** groups the rows with the same value of the specified column into “summary rows”. In each summary row, aggregated information is collected. To further explain this, consider the following example.

```
> SELECT * FROM test;
+-----+-----+-----+
| test_id | value_1 | value_2 |
+-----+-----+-----+
| 1 | a | 10 |
| 2 | a | 20 |
| 3 | a | 30 |
| 4 | b | 100 |
| 5 | b | 200 |
| 6 | c | 1000 |
| 7 | c | 2000 |
+-----+-----+-----+
```

Running the following command gives

```
> SELECT * FROM test GROUP BY value_1;
+-----+-----+-----+
| test_id | value_1 | value_2 |
+-----+-----+-----+
| 1 | a | 10 |
| 4 | b | 100 |
| 6 | c | 1000 |
+-----+-----+-----+
```

From the result, it can be seen that summary rows have been created using **GROUP BY**. In the returned table, **value_1** has distinguished values. In this example, it simply picks up the first appearance of the rows in the original table that has distinguished **value_1**, and a lot of information seems to be lost. However, it is worth mentioning that although not displayed, the aggregated information is included.

To verify the presence of the aggregated information, consider running the following command.

```
> SELECT COUNT(*) FROM test GROUP BY value_1;
+-----+
| COUNT(*) |
+-----+
| 3 |
| 2 |
| 2 |
+-----+
```

From the result, we can see that the counted number of each summary row is returned. Similarly, the following SQL returns other aggregation information associated with each summary row.

```
> SELECT value_1, COUNT(*), SUM(value_2) FROM test GROUP BY value_1;
+-----+-----+-----+
| value_1 | COUNT(*) | SUM(value_2) |
+-----+-----+-----+
| a       |      3 |       60 |
| b       |      2 |     300 |
| c       |      2 |   3000 |
+-----+-----+-----+
```

Finally, `ORDER BY` and `LIMIT` controls the sequence and maximum number of returned rows, respectively.

The returns of multiple queries might be able to `UNION` together, if they are union-compatible. Union simply means concatenate the tables vertically. To union the results, use

```
SELECT <...>
UNION
SELECT <...>
UNION
SELECT <...>
...
SELECT <...>;
```

where inside `<...>` are the original query statements. Notice that for the queries to be union-compatible, they must have the same number of columns with identical data type for the associated columns. The names of the column in the returns, if different, follow the first query result. Use alias `AS` to change the names if needed. Duplicated rows in the union will be excluded. If duplications need to be included in the result, certain DBMS provides the `UNION ALL` option.

SQL uses nest queries to add more flexibility. Nest queries plays as the intermediate steps to provide a temporary searching result, from which another query can be executed. Wherever a table name appears in the query, it can be replaced by a `SELECT` statement nested in a bracket “`()`”. A demonstrative example is given below. Consider the same tables `test` and `test_join` as follows.

```
> SELECT * FROM test;
+-----+-----+-----+
| test_id | value_1 | value_2 |
+-----+-----+-----+
| 1 | a | 10 |
| 2 | a | 20 |
| 3 | a | 30 |
| 4 | b | 100 |
| 5 | b | 200 |
| 6 | c | 1000 |
| 7 | c | 2000 |
+-----+-----+-----+
```

```
> SELECT * FROM test_join;
+-----+-----+-----+
| test_join_id | value_1 | value_2 | value_3 |
+-----+-----+-----+
| a           |    10  |    99  | alpha   |
| b           |   100  |   999  | bravo   |
| d           | 10000  | 99999  | delta   |
+-----+-----+-----+
```

A inner join is provided to the above tables. However, for each `value_1` in the first table, the sum of the associated `value_2`, instead of each individual row, is used. This can be achieved using

```
> SELECT temp.value_1 AS type,
      ->          temp.sum_value_2 AS total_value,
      ->          test_join.value_1 AS minval,
      ->          test_join.value_2 AS maxval,
      ->          test_join.value_3 AS abbrev
      -> FROM (SELECT value_1,
      ->             SUM(value_2) AS sum_value_2
      ->           FROM test GROUP BY value_1) AS temp
      -> JOIN test_join
      -> ON (temp.value_1 = test_join.test_join_id);
+-----+-----+-----+
| type | total_value | minval | maxval | abbrev |
+-----+-----+-----+
| a    |      60     |    10  |    99  | alpha   |
| b    |     300     |   100  |   999  | bravo   |
+-----+-----+-----+
```

where notice that alias are quite some times to clarify the logics.

Nest queries can be popular in table joins as well as filter criteria, where the boundary of a variable can be obtained from a nest query.

11.3.8 Trigger

A trigger defines a set of operations to be carried out automatically when something happens to specified tables. For example, in any case a new row is added to a table, a trigger can automatically insert an associated record into another table.

There are mainly 3 types of triggers: DML trigger (triggered by `INSERT`, `UPDATE`, `DELETE`, etc.), DDL trigger (triggered by `CREATE`, `ALTER`, `DROP`, `GRANT`, `DENY`, `REVOKE`, etc.), and CLR trigger (triggered by `LOGON` event).

A quick DML trigger can be defined as follows.

```
CREATE TRIGGER <trigger-name>
  [BEFORE | AFTER] [INSERT | UPDATE | DELETE] ON <table-name>
  FOR EACH ROW <operation>;
```

where `BEFORE` is often used to validate and modify data to be added to

<table-name>, and AFTER is often used to trigger other changes consequent to this change.

In case multiple operations need to be defined, consider using

```
DELIMITER $$  
CREATE TRIGGER <trigger-name>  
    [BEFORE | AFTER] [INSERT | UPDATE | DELETE] ON <table-name>  
    FOR EACH ROW BEGIN  
        <operation>;  
        ...  
        <operation>;  
    END$$  
DELIMITER ;
```

where DELIMITER \$\$ and DELIMITER ; is used to temporarily change the delimiter for the BEGIN...END statement. It is possible to build slightly complicated logics in the operations, for example to build conditional statements.

Use NEW in the operation(s) to represent the rows that is added/updat-ed/deleted from the table-name.

Use the following to drop a trigger.

```
DROP TRIGGER <trigger-name>;
```

11.3.9 SQL Demonstrative Example

An an example to demonstrate the use of SQL, a database is created from scratch. MariaDB is used as the DBMS in this example. More about Mari-aDB is introduced in later Section 15. The database is used in the smart home project to track the resources obtained and consumed by the user. The resources in this context may refer to groceries bought from the supermarket, books purchased online, subscriptions of magazines and services, etc. For simplicity, the prompt is ignored in the rest of this section.

Check the existing databases as follows.

```
SHOW DATABASES;
```

A database named `smart_home` is created as follows.

```
CREATE DATABASE smart_home;
```

Select the database as follows.

```
USE smart_home;
```

With the above command, `smart_home` is selected as the current database.

Based on the database schema design, a few tables need to be created. We shall start with creating `asset`, `accessory`, `consumable` and `subscription` tables as follows.

The `asset` table is used to trace assets in the home. They are often expensive and comes with a serial number or a warranty number, and shall

persist for a long time (a few years, at minimum). Examples of assets include beds, televisions, computers, printers, game consoles. The **accessory** table is used to trace relatively cheaper accessories than assets. Though they are designed to last long, they may not have an serial number. Examples of accessories include books, charging cables, coffee cups. The **consumable** table is used to trace items that is meant to be used up or expire. Examples of consumable items include food, shampoo, A4 printing paper. And finally the **subscription** table is used to trace subscriptions of services. Examples of these services include software license (either permanent license or annual subscription license), magazine subscriptions, membership subscriptions, and digital procurement of a movie.

The serial number or warranty number for assets are used as the primary key of **asset** table. For the other three tables, surrogate keys are used. Each table has a column **product_type_id** that specifies the type of the item, such as “television”, “cooker”, “fruit”, “software”. The types in these tables are given by integer indices. A separate **product_type** relates the indices with their associated meanings. The same applies to **product_brand_id** and **payment_method_id**.

Create asset table as follows.

```
CREATE TABLE asset (
    PRIMARY KEY (serial_num),
    serial_num          VARCHAR(50)      NOT NULL,
    product_type_id    INT(5),
    product_brand_id   INT(5),
    product_name        VARCHAR(50)      NOT NULL,
    receipt_num         VARCHAR(50),
    procured_date       DATE           NOT NULL DEFAULT (
        CURRENT_DATE),
    procured_price      DECIMAL(10,2),
    payment_method_id  INT(5),
    warranty_date_1     DATE           NOT NULL DEFAULT (
        CURRENT_DATE),
    warranty_date_2     DATE           NOT NULL DEFAULT (
        CURRENT_DATE),
    expire_date         DATE           NOT NULL DEFAULT
        '9999-12-31',
    CONSTRAINT warranty_after_procured
    CHECK(warranty_date_1 >= procured_date AND warranty_date_2 >=
        warranty_date_1),
    CONSTRAINT expire_after_procured
    CHECK(expire_date >= procured_date)
);
```

where

- **serial_num**: the serial number, MAC number or registration ID that can be used to uniquely identify the asset.

- `product_type_id`: type index.
- `product_brand_id`: brand index.
- `product_name`: full name of the product that can uniquely specify the asset on the market.
- `receipt_num`: receipt and/or warranty number.
- `procured_date`: date of procurement.
- `procured_price`: price of the product as procured.
- `payment_method_id`: payment method.
- `warranty_date_1`: warranty expiration date (free replace or repair); leave it as the procured date if no such warranty is issued.
- `warranty_date_2`: second warranty expiration date (partially covered repair); leave it as the procured date if no such warranty is issued.
- `expire_date`: the date when the asset expires or needs to be returned. For example, in Singapore a car “expires” in 10 years from the day of procurement.

Notice that constraints and default values have been added to the table creation. An SQL script is used contain the code, and

```
$ mariadb -u <user-name> -p < <script-name>
```

is used to execute the script, which is more convinient than typing all the lines in the MariaDB console.

Similarly, create the rest 3 tables for the resources as follows.

```
CREATE TABLE accessory (
    PRIMARY KEY (item_id),
    item_id          INT(5)      AUTO_INCREMENT,
    product_type_id INT(5),
    product_brand_id INT(5),
    product_name    VARCHAR(50) NOT NULL,
    receipt_num     VARCHAR(50),
    procured_date   DATE        NOT NULL DEFAULT (
        CURRENT_DATE),
    procured_number DECIMAL(10,2) NOT NULL DEFAULT 1.00,
    procured_unit_price DECIMAL(10,2),
    procured_price   DECIMAL(10,2),
    payment_method_id INT(5),
    expire_date     DATE        NOT NULL DEFAULT
        '9999-12-31',
    CONSTRAINT expire_after_procured
    CHECK(expire_date >= procured_date)
);
```

```
CREATE TABLE consumable (
    PRIMARY KEY (item_id),
    item_id          INT(5)      AUTO_INCREMENT,
    product_type_id INT(5),
    product_brand_id INT(5),
    product_name     VARCHAR(50) NOT NULL,
    receipt_num      VARCHAR(50),
    procured_date    DATE       NOT NULL DEFAULT (
        CURRENT_DATE),
    procured_number   DECIMAL(10,2) NOT NULL DEFAULT 1.00,
    procured_unit_price DECIMAL(10,2),
    procured_price    DECIMAL(10,2),
    payment_method_id INT(5),
    expire_date      DATE       NOT NULL DEFAULT (
        CURRENT_DATE),
    CONSTRAINT expire_after_procured
    CHECK(expire_date >= procured_date)
);

CREATE TABLE subscription (
    PRIMARY KEY (item_id),
    item_id          INT(5)      AUTO_INCREMENT,
    product_type_id INT(5),
    product_brand_id INT(5),
    product_name     VARCHAR(50) NOT NULL,
    receipt_num      VARCHAR(50),
    procured_date    DATE       NOT NULL DEFAULT (
        CURRENT_DATE),
    procured_price   DECIMAL(10,2),
    payment_method_id INT(5),
    expire_date      DATE       NOT NULL DEFAULT (
        CURRENT_DATE),
    CONSTRAINT expire_after_procured
    CHECK(expire_date >= procured_date)
);
```

Create the tables for users, product types, product brands and payment methods as follows.

```
CREATE TABLE user (
    PRIMARY KEY (user_id),
    user_id          INT(5),
    first_name       VARCHAR(50) NOT NULL,
    last_name        VARCHAR(50) NOT NULL,
    email            VARCHAR(50) NOT NULL UNIQUE
);

CREATE TABLE product_type (
    PRIMARY KEY (product_type_id),
```

```

    product_type_id      INT(5)      AUTO_INCREMENT,
    product_type_name    VARCHAR(50) NOT NULL UNIQUE,
    product_type_name_sub VARCHAR(50) NOT NULL DEFAULT ('na')
);

CREATE TABLE product_brand (
    PRIMARY KEY (product_brand_id),
    product_brand_id      INT(5)      AUTO_INCREMENT,
    product_brand_name    VARCHAR(50) NOT NULL UNIQUE
);

CREATE TABLE payment_method (
    PRIMARY KEY (payment_method_id),
    payment_method_id      INT(50)      AUTO_INCREMENT,
    user_id                INT(5),
    payment_method_name    VARCHAR(50) NOT NULL
);

```

Finally, create foreign keys as follows.

```

ALTER TABLE payment_method
ADD FOREIGN KEY (user_id)
REFERENCES user(user_id);
ALTER TABLE asset
ADD FOREIGN KEY (product_type_id)
REFERENCES product_type(product_type_id);
ALTER TABLE asset
ADD FOREIGN KEY (product_brand_id)
REFERENCES product_brand(product_brand_id);
ALTER TABLE asset
ADD FOREIGN KEY (payment_method_id)
REFERENCES payment_method(payment_method_id);
ALTER TABLE accessory
ADD FOREIGN KEY (product_type_id)
REFERENCES product_type(product_type_id);
ALTER TABLE accessory
ADD FOREIGN KEY (product_brand_id)
REFERENCES product_brand(product_brand_id);
ALTER TABLE accessory
ADD FOREIGN KEY (payment_method_id)
REFERENCES payment_method(payment_method_id);
ALTER TABLE consumable
ADD FOREIGN KEY (product_type_id)
REFERENCES product_type(product_type_id);
ALTER TABLE consumable
ADD FOREIGN KEY (product_brand_id)
REFERENCES product_brand(product_brand_id);
ALTER TABLE consumable
ADD FOREIGN KEY (payment_method_id)
REFERENCES payment_method(payment_method_id);

```

```
ALTER TABLE subscription
ADD FOREIGN KEY (product_type_id)
REFERENCES product_type(product_type_id);
ALTER TABLE subscription
ADD FOREIGN KEY (product_brand_id)
REFERENCES product_brand(product_brand_id);
ALTER TABLE subscription
ADD FOREIGN KEY (payment_method_id)
REFERENCES payment_method(payment_method_id);
```

11.4 General Ways to Connect to RDB

This section discusses the tools or program interfaces used along with the DBMS. Many software programs provide interface or toolkit to connect to a database. For example, MATLAB uses database toolbox to connect to SQLite or Microsoft SQL server. Python, as a “glue” language, also provide variety of packages to connect to different databases. IDEs such as VSCode can connect to databases using extensions.

Python provides variety of libraries to access RDS, many of which use embedded SQL codes to interact with the DBMS. Depending on the DBMS, different libraries and commands can be used, some of which more general and the other more specific to a particular DBMS.

In this section, both `pandas` and `mariadb` libraries are introduced. The `pandas` library provides data manipulation and analysis tools, and it provides `pandas.io.sql` that allows connecting to a DBMS and embedding SQL commands into the python code. The `mariadb` library, on the other hand, is dedicated for MariaDB connection. Like `pandas.io.sql`, it also allows embedding SQL commands to interface the DMBS.

As pre-requisites, make sure that the following has been done.

- The DBMS has been configured to allow remote access.
- Make sure that an account has been registered in DBMS that has the privilege of operation from a remote machine.
- Make sure that the firewall configuration is correct.

For DBMS configuration, in the case of MariaDB, use the following code in the shell to check the location of the configuration files.

```
$ mysqld --help --verbose
```

Typical locations of the configuration files are `/etc/my.cnf` and `/etc/mysql/my.cnf`. In the configuration file, use the following to disable binding address.

```
[mysqld]
skip-networking=0
skip-bind-address
```

For account setup, in the DBMS console, use something like

```
> GRANT ALL PRIVILEGES ON *.* TO '<user-name>'@<ip-address>
    IDENTIFIED BY '<user-password>' WITH GRANT OPTION;
```

where '*<ip-address>*' is the remote machine that runs the Python program. If the Python codes are running locally, simply use 'localhost'.

It might be necessary to install MariaDB database development files in the DMBS host machine for the Python libraries to be introduced to function properly. Install the development as follows.

```
$ sudo apt install libmariadb-dev
```

PANDAS Library

Python library **pandas** is one of the essential libraries for data analysis. It provides flexible interfaces and tools for data reading and processing and works very well with different data formats and engines including CSV, EXCEL and DBMS. This section focuses mainly on the interaction of **pandas** with DBMS. Therefore, the detailed use of **pandas** for data analysis, etc., are not covered in this section.

A class **pandas.DataFrame** is defined in **pandas** as the backbone to store and process data. The data attribute of **pandas.DataFrame** is a **numpy** array. Many functions are provided to read different data formats into **pandas** data frame, which makes reading data easy and convenient. An example of reading a CSV file is given below.

```
import pandas as pd
df = pd.read_csv(<file-name>)
print(df)
print(df.head(<number>))
print(df.tail(<number>))
print(df.info())
```

where **head()**, **tail()** gives the first and last rows of the data frame, and **info()** checks the data frame basic information including shape and data types of the columns. Check **df.columns** for all the columns of the data frame. Details specific to a column can be accessed via **df.[<column-name>]**. Many functions are provided to further abstract the details, such as grouping and counting. Use **df.loc(<row-index-list>, <column-name-list>)** to check the content of specified rows and columns.

With **pandas** and other relavent libraries, Python can connect to a database and execute a query. An example of using **pandas** to connect to an Microsoft SQL server and implement a query is given below. Notice that different DBMS may require different database connectivity driver standards, and there are mainly two of them, namely open database connectivity (ODBC)

and Java database connectivity (JDBC). Microsoft adopts ODBC, and a separate package is required in the Python program to connect to the Microsoft SQL server.

```
import pyodbc
import pandas.io.sql as psql

server = "<server-url>,<port>"
database = "<database>"
uid = "<uid>"
pwd = "<pwd>"
driver = "<driver>" # such as "{ODBC Driver 17 for SQL Server}"

# connect to database
conn = pyodbc.connect(
    server = server,
    database = database,
    uid = uid,
    pwd = pwd,
    driver = driver
)

# get cursor
cursor = conn.cursor()

# execute sql command
query = """<query>"""
runs = psql.read_sql_query(query, conn)
```

The above codes returns a data frame corresponding to the result set of the query string, which is saved in `runs`.

MARIADB Library

Use the following code to test connectivity from Python to the database.

```
import mariadb
import sys

user = "<user>"
password = "<password>"
host = "<server-url>"
port = "<port>" # MariaDB default: 3306
database = "<database>"

# connect to database
try:
    conn = mariadb.connect(
        user = user,
        password = password,
        host = host,
```

```
port = port,
database = database
)
except mariadb.Error as e:
print(f"Error connecting to MariaDB Platform: {e}")
sys.exit(1)

# get cursor
cur = conn.cursor()

# execute sql command
cur.execute("<sql-command>")
```

Notice that for query, the result is stored in the cursor object. Use a for loop to view the results.

12

RDB Example: MySQL and MariaDB

CONTENTS

12.1	Installation	134
12.2	MariaDB Basic Configuration	134
12.3	DBMS Console	135
12.4	Run SQL File	135

Commonly seen DBMS examples include Oracle Database, MySQL, Microsoft SQL Server, PostgreSQL, SQLite, MariaDB, and many more. Some of them are briefly introduced in this section. Here are some examples.

MariaDB and MySQL are two widely used relational DBMS. MariaDB is initially a fork of MySQL, and in this sense they share many similarities. While MySQL moves towards a dual license approach (free community license and paid enterprise license with proprietary code), MariaDB is designed to be fully open-source under GNU license, and plays as a replacement of MySQL.

In general, MariaDB supports a larger varieties of data engines and new features, and it is claimed to be faster, more powerful and advanced than MySQL. However, it lacks some of the enterprise features provided by MySQL. The users can gain some of these features by using open-source plugins. The most recent new features in MySQL and MariaDB are also diverging.

Some of the main features of MySQL and MariaDB are summarized as follows.

- Good performance (very fast) in general, for a medium size database. Hence, it is popular in many web applications.
- Ease of use.
- Supports in-memory tables to handle read-heavy write-lite tasks.
- Not very flexible, as compared to PostgreSQL where more complex data types, queries, and functions add-ons are supported.

Different databases may propose different minimum system requirements. There is no standard on how these minimum requirements are calculated. Therefore, it is often unfair to directly compare the requirements of different databases. Nevertheless, a summary table is given in Table 15.1. Notice that this is merely an estimation and can differ largely from practice.

TABLE 12.1

An estimation of DBMS hardware requirements.

	OS	Minimum Requirements		
		CPU	RAM	Disk*
MySQL (E)	All*	2 core	2 GB	1.3 GB
MySQL (C)	All	1 core	1 GB	1.3 GB
MariaDB	All	1 core	400MB	660 MB
PostgreSQL	All	1 core	1 GB	40MB
Firebird	All	1 core	12MB	15MB

* Some installations may come with default test database and logs. The disk usage can be reduced if those files are removed after installation.

** “All” refers to Linux, MacOS and Windows. Different platforms may have different minimum requirements.

In the remaining of this section, MariaDB is used for demonstration.

12.1 Installation

To install MariaDB, follow the instruction on the official website. Different OS, such as RHEL and Ubuntu, may require different ways of installation. For example, on RHEL

```
$ sudo yum update
$ sudo yum install mariadb-server
```

and on Ubuntu,

```
$ sudo apt update
$ sudo apt install mariadb-server
```

Notice that MySQL can be installed instead by replacing `mariadb-server` with `mysql-server`. Similarly, replacing `mariadb` with `mysql` in the rest of this section, wherever applicable, would work for MySQL.

MariaDB server can be controlled using `systemctl` which is introduced in Section 9.1. For example, to start MariaDB, use

```
$ sudo systemctl start mariadb.service
```

and to check its status, use

```
$ sudo systemctl status mariadb
```

12.2 MariaDB Basic Configuration

After installation of and starting MariaDB, use

```
$ sudo mysql_secure_installation
```

to run a quick security-related configuration such as creating password for the root user, and deleting test database.

Log in to MariaDB console using

```
$ sudo mariadb
```

Notice that when *sudo* privilege is used, the user should be brought to the root account of the DBMS. Without sudo privilege, a user name and a password is required to log in to the DBMS as follows.

```
$ mariadb -u <user-name> -p  
Enter Password:
```

Depending on the setup, remote log in to the database from other machine, especially using root account, might be forbidden.

12.3 DBMS Console

After logging in to MariaDB console, a prompt that looks like the following would show up.

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

from where an admin account can be created as follows.

```
MariaDB [(none)]> GRANT ALL PRIVILEGES ON *.* TO '<user-name>'@'localhost'  
      IDENTIFIED BY '<user-password>' WITH GRANT OPTION;  
MariaDB [(none)]> FLUSH PRIVILEGES;
```

By using an admin account instead of the root account in daily operations, the security risk is reduced.

Notice that remote access to a database is by default forbidden. A command similar with the above is required to enable remote access. Wildcard expression can be used for the IP address, if necessary, to allow a user to access the database from multiple machines.

To check existing users and their IP addresses to whom access has been granted, use

```
SELECT host, user FROM mysql.user;
```

Finally, use

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

to quite MariaDB console.

12.4 Run SQL File

In many occasions, instead of executing SQL commands line by line, an SQL file is prepared in advance and the DBMS is asked to execute the SQL file as a whole. In the Linux shell, execute the following command and MariaDB shall be able to execute an SQL file on the specified host, on behalf of the user, on the selected database.

```
$ mariadb --host="mysql_server" --user="user_name" --database="database_name" --password="user_password" < "file.sql"
```

If the user already logs into the MariaDB console, use the following command instead.

```
MariaDB [(none)]> source file.sql
```

13

RDB Example: PostgreSQL

CONTENTS

13.1	Brief Introduction	137
13.2	Installation and Authentication	138
13.2.1	Installation	138
13.2.2	Regular Authentication	139
13.2.3	Peer Authentication	139
13.2.4	Run PostgreSQL in Containers	140
13.3	PostgreSQL-Specific New Data Types	140
13.4	PostgreSQL User-Defined Data Types	141
13.5	PostgreSQL Stored Procedural and Functions	142
13.6	Manipulation and Query	144

PostgreSQL (or Postgres) is a powerful open-source relational database project. It is gaining a lot of popularity recently.

13.1 Brief Introduction

PostgreSQL is claimed to be the world's most advanced open-source RDB, and it has some great features. It is a object-relational database where the user can define customized data types in the form of objects. The database functions are modularized, meaning that it has a small base installation (only 40MB) as given in Table 15.1, and the user can expand its capability by installing additional modules per required. It complies very well with the latest SQL standard, with additional powerful add-ons.

Though powerful and efficient, PostgreSQL has not grown as popular as some other databases such as Oracle, Microsoft SQL server, and MySQL. One of the reasons for this is the lack of enterprise tier support. Being under PostgreSQL license (an open-source license similar to BSD and MIT), the available support is mostly from the community. With that being said, PostgreSQL is mature and has been adopted in some large enterprises. Of course the IT department of these companies need to work hard to maintain the database. In addition, great flexibility often means a steeper learning curve,

making mastering PostgreSQL generally more difficult than other aforementioned databases.

However, with the populating use of microservices where the database efficiency and performance becomes absolutely critical, PostgreSQL is gaining more and more attentions. It is especially popular when using inside containers.

Some of the main features of PostgreSQL are summarized as follows.

- Object-relational database.
- Excellent adherence to SQL standard.
- Excellent performance and scalability.
- Multi-version concurrency control.
- Modularization of features, i.e., light weight base installation and scalable add-ons.
- Disk-based database, and does not support in-memory tables. As a compensation, it has robust caching mechanisms that speed up reading and writing.
- Steep learning curve.

13.2 Installation and Authentication

The installation of PostgreSQL depends on the OS. As of this writing, the latest version of RHEL 9.4 is used as an example. Both installation and authentication to the database are introduced.

13.2.1 Installation

PostgreSQL supports a large range of operating systems including Linux, macOS, Windows, and more. The source code and very detailed documentations of PostgreSQL are available on its websites and can be downloaded to a local machine freely. Due to the lite weight of the base installation, PostgreSQL can be used conveniently in containers. The associated images can be found on Docker Hub.

To install PostgreSQL directly on a host machine, simply follow the instructions on the official website, where command line tools are provided for Linux users, and installer for windows users. For example, to install and enable PostgreSQL 16 on RHEL 9.4 running on `x86_64`, use

```
$ sudo dnf module install postgresql:16/server
$ sudo postgresql-setup --initdb
$ sudo systemctl start postgresql.service
$ sudo systemctl enable postgresql.service
```

After installation, use command `pg_ctl` to control the server, and `psql` to log in to the server. Notice that PostgreSQL installation may come with PgAdmin, the GUI for the DBMS, which can also be used to interact with the databases.

13.2.2 Regular Authentication

The default “root” user of a PostgreSQL database is “`postgres`”. A database of the same name “`postgres`” is also created automatically. A sudoer can log in to this account as follows.

```
$ sudo -u postgres psql postgres
postgres=#
```

where `postgres=#` is the prompt of the DBMS console.

After logging in to the database as user `postgres`, the user can create users and databases using

```
postgres=# CREATE ROLE <username> LOGIN PASSWORD '<password>';
postgres=# CREATE DATABASE <database name> WITH OWNER = <username>;
```

and log in to the database as the user by

```
$ psql -h localhost -d <database name> -U <username> -p <port>
```

where `<port>` is defined in the configuration file and it is 5432 by default.

13.2.3 Peer Authentication

Given that the user already logged in to the OS with his username in the OS, it is possible to use peer authentication if the login is from the local host and the PostgreSQL username is the same with the OS username.

Quickly create a user in PostgreSQL with the same username as OS using

```
$ sudo -u postgres createuser -s $USER
```

which essentially logs in as PostgreSQL and creates a user of `$USER`. With the user created, create a database for the user using

```
$ createdb <database name>
```

Then log in to PostgreSQL using peer authentication as follows

```
$ psql -d <database name>
<username>=>
```

where `<username>=>` is the prompt to a regular user

If the database name happens to be the same with `$USER`, simply use

```
$ psql
<username>=>
```

to login to the database.

After logging in to the database, use \du and \l to check the users and the databases in PostgreSQL, and use SHOW port; to check the port that PostgreSQL is running on which by default should be 5432.

13.2.4 Run PostgreSQL in Containers

To use PostgreSQL in containers, either download and run PostgreSQL images from Docker Hub, or create a Dockerfile like the following that installs and configures PostgreSQL on top of an Alpine base image. An example of such Dockerfiles is given below.

```
FROM alpine
RUN apk update
# install postgresql
RUN apk add postgresql
RUN mkdir /run/postgresql
RUN chown postgres:postgres /run/postgresql/
USER postgres
WORKDIR /var/lib/postgresql
RUN mkdir /var/lib/postgresql/data
RUN chmod 0700 /var/lib/postgresql/data
RUN initdb -D /var/lib/postgresql/data
# prepare user scripts
RUN mkdir /var/lib/postgresql/user-scripts
RUN chmod 0700 /var/lib/postgresql/user-scripts
COPY ./start.sh /var/lib/postgresql/user-scripts
COPY ./setup_db.sql /var/lib/postgresql/user-scripts
COPY ./populate_db.py /var/lib/postgresql/user-scripts
# prepare user data
RUN mkdir /var/lib/postgresql/user-data
RUN chmod 0700 /var/lib/postgresql/user-data
COPY ./google_stock_price.csv /var/lib/postgresql/user-data
#
CMD ["/bin/sh", "/var/lib/postgresql/user-scripts/start.sh"]
```

13.3 PostgreSQL-Specific New Data Types

PostgreSQL supports a large range of data types, more than other databases such as MySQL and MariaDB. In addition to the commonly seen numeric types, character types, date and time types and boolean type, the following data types are supported.

- Currency (monetary) types.
- Geometric types, including points, line segments, boxes, paths, polygons and circles.
- Network address types, such as IPv4 and IPv6 addresses and MAC addresses.
- Array types and associated functions such as accessing, modifying and searching arrays.
- Composite types and associated functions.
- Many more.

13.4 PostgreSQL User-Defined Data Types

User-defined data types can be used to demonstrate the object-relational database aspect of PostgreSQL. It essentially means that the attribute of an element can be an object, and can have some complex and comprehensive features.

To create customized data types, use the following syntax

```
/*Composite Types*/
CREATE TYPE name AS
( [ attribute_name data_type [ COLLATE collation ] [ , ... ] ] );

/*Enumerated Types*/
CREATE TYPE name AS ENUM
( [ 'label' [ , ... ] ] );

/*Range Types*/
CREATE TYPE name AS RANGE (
SUBTYPE = subtype
[ , SUBTYPE_OPCLASS = subtype_operator_class ]
[ , COLLATION = collation ]
[ , CANONICAL = canonical_function ]
[ , SUBTYPE_DIFF = subtype_diff_function ]
[ , MULTIRANGE_TYPE_NAME = multirange_type_name ]
);

/*Base Types*/
CREATE TYPE name (
INPUT = input_function,
OUTPUT = output_function
[ , RECEIVE = receive_function ]
```

```
[ , SEND = send_function ]
[ , TYPMOD_IN = type_modifier_input_function ]
[ , TYPMOD_OUT = type_modifier_output_function ]
[ , ANALYZE = analyze_function ]
[ , SUBSCRIPT = subscript_function ]
[ , INTERNALLENGTH = { internallength | VARIABLE } ]
[ , PASSEDBYVALUE ]
[ , ALIGNMENT = alignment ]
[ , STORAGE = storage ]
[ , LIKE = like_type ]
[ , CATEGORY = category ]
[ , PREFERRED = preferred ]
[ , DEFAULT = default ]
[ , ELEMENT = element ]
[ , DELIMITER = delimiter ]
[ , COLLATABLE = collatable ]
);
```

For example,

```
CREATE TYPE sex_type AS
enum ('M', 'F');
```

which creates a new data type called **sex_type**, and it can take enumerated value of either M or F.

13.5 PostgreSQL Stored Procedural and Functions

Many databases including Oracle SQL, Microsoft SQL Server, MySQL, MariaDB and PostgreSQL, support stored procedures and user defined functions. This feature has been there for a very long time, but it is often not introduced in a typical introductory course. Though useful, they may introduce performance and portability issues, hence need to be handled with caution. Besides, nowadays it is often regarded as a better practice to implement the logics in the application layer, not in DBMS. Nevertheless, they are briefly introduced as follows.

The following is an example to define a function using SQL.

```
CREATE OR REPLACE FUNCTION add_int(int, int)
RETURNS int AS
'
SELECT $1+$2;
'
LANGUAGE SQL
```

where notice that the input variable types are given in the bracket (use () if

there is no input), the output following RETURNS (use void if there is no output), and the SQL operations in between quotations ', which is a delimiter and can be replaced by something else, such as the following

```
CREATE OR REPLACE FUNCTION add_int(int, int)
RETURNS int AS
$body$
SELECT $1+$2;
$body$
LANGUAGE SQL
```

Instead of using \$1, \$2 to refer to a input, names can be assigned together with types as follows.

```
CREATE OR REPLACE FUNCTION add_int(var1 int, var2 int)
RETURNS int AS
$body$
SELECT var1+var2;
$body$
LANGUAGE SQL
```

Notice that so far we have been using SQL as the programming language for the functions, as indicated by LANGUAGE SQL. Notice that PostgreSQL also supports other languages, such as PL/pgSQL, which is a procedural programming language supported by PostgreSQL. It closely resembles Oracle's PL/SQL language. "PL" in these terms represents "Procedural Language".

The following is a list of languages supported by PostgreSQL.

- Naive installation:

- PL/pgSQL
- SQL
- C

- Extension:

- PL/Python
- PL/Perl
- PL/Java
- PL/R
- and more.

SQL is sufficient to carry out simple and straight forward tasks such as adding two numbers, as shown in the earlier example. However, when comes to handling conditional statements and loops, etc., procedural language is required. When the function is complex, it is sometimes impossible or inefficient to implement it using SQL, and PL/pgSQL and other procedural languages can solve this problem. An example is given below.

```

CREATE OR REPLACE FUNCTION increment_value(value INT, increment INT)
RETURNS INT AS $$

DECLARE
    result INT;
BEGIN
    IF increment > 0 THEN
        result := value + increment;
    ELSE
        RAISE EXCEPTION 'Increment must be positive';
    END IF;
    RETURN result;
END;
$$ LANGUAGE plpgsql;

```

With the above been said, though convenient and powerful it might be in some use cases, it is often a good practice to keep complex logic in application layer, for logic consistency and database portability.

13.6 Manipulation and Query

PostgreSQL adopts SQL for database manipulation and query. SQL has been introduced in earlier sections, hence it is not repeated here. Only selected unique features to PostgreSQL are introduced.

While PostgreSQL server is running, enter its console using `psql` from the shell. One can tell PostgreSQL console by its prompt which looks something like

```
postgres#
```

or

```
postgres>
```

with “postgres” the current selected database. It is also possible to specify user name, default database, and other configurations when connecting to PostgreSQL server. Instead of running `psql` as admin, use

```
$ psql -h <host> -p <port> -U <username> <default_database>
```

Once in PostgreSQL console, use `help` to display the basic commands, including `\copyright` that shows the distribution terms, `\h` to check SQL commands, `\?` to check psql commands, and `\q` to quit PostgreSQL console, etc. Notice that both SQL and psql commands can be used in PostgreSQL console.

Some widely used psql commands are summarized in Table 16.1. Most, if not all, psql commands start with a back slash “\”.

SQL commands, such as creating a database, have been introduced earlier, hence is not repeated here. =====

TABLE 13.1

Widely used psql commands.

Command	Description
<code>SELECT VERSION();</code>	Check PostgreSQL version.
<code>\l</code>	List databases.
<code>\c <database></code>	Switch databases.
<code>\d</code>	Describe items. By default, it lists the tables in the current database, and describe each of them.
<code>\dt</code>	List tables.
<code>\dv</code>	List views.
<code>\dn</code>	List schema.
<code>\df</code>	List functions.
<code>\du</code>	List users.
<code>\d <table></code>	Describe a table.
<code>\s</code>	Show command history.
<code>\h</code>	Show help.
<code>\?</code>	Show psql commands.
<code>\!cls</code>	Clear screen.
<code>\q</code>	Quit DBMS shell.



14

Database

CONTENTS

14.1	Relational Database	147
14.2	Non-Relational Database	149
14.3	Structured Query Language	150
14.3.1	RDB Naming Conventions	150
14.3.2	General Introduction to SQL	151
14.3.3	General Syntax	151
14.3.4	Database Manipulation	155
14.3.5	Table Manipulation	156
14.3.6	Row Manipulation	158
14.3.7	Query	159
14.3.8	Trigger	167
14.3.9	SQL Demonstrative Example	168
14.4	General Ways to Connect to RDB	173

Database and database manage systems (DBMS) are introduced in this and consecutive chapters. Both relational and non-relational databases are covered. Tools and languages to manipulate databases, such as structured query language (SQL) for relational databases, are introduced.

14.1 Relational Database

Database, in a broad view, refers to an organized collection of data of any format. In this sense, any file format that hosts information in a meaningful and explainable way, such as *CSV*, *XML* and *JSON*, is a database. These file formats often work fine when the data is stored in a centralized manner and its size small.

As the data size grows, the robustness and efficiency of storing and retrieving data become challenging, and different database models have been proposed to tackle it. Dedicated software, namely DBMS, is developed to manage and maintain the database and provide an interface for the users to create, retrieve, update and delete data. Different database models require dif-

ferent database engines and DBMS, and different DBMS may provide different interfaces for the users.

There are many database models available in the market. The most widely seen database models can be divided into two categories, namely the relational databases (RDB) and the non-relational databases.

Relational database was proposed in 1970s by IBM. Some important features of a relational database include the following.

- Structure the data as “relations”, which is a collection of tables, each consisting of a set of rows (also known as tuple/record) and columns (also known as attribute/field).
- For each table, a primary key, either being a column or a combination of several columns, is defined that can uniquely distinguish a row from others.
- Provide relational operations that manipulate the data in the tables, for example, joining tables together and aligning them using an attribute.

SQL, a domain-specific language, can be used in managing RDB and interfacing relational DBMS. Most commercialized RDB management systems (RDBMS) adopt SQL as the query language. There are alternatives, but they are rarely used compared to SQL. There is a evolving standard on what operations should an RDBMS support using SQL. The latest version as of this writing is **SQL:2023**.

Examples of relational databases include *Oracle*, *MySQL*, *Microsoft SQL Server*, *MariaDB*, *IBM Db2*, *Amazon Redshift*, *Amazon Aurora* and *PostgreSQL*.

An example of a table used in RDB is given in Table 14.1. The table has a name `user` and 4 attributes `user_id`, `user_email`, `membership` and `referee_id`. A table should have an attribute (or a set of attributes) defined as the primary key. In this example, `user_id` is assigned to be the primary key as denoted by the asterisk. The primary key is used to uniquely identify a row in the table. A primary key can consist a single column or multiple columns. When a primary key is composed of multiple columns, it is called a composite key. A table should have one and only one primary key.

Depending on the meaning behind the primary key, it can be divided into two types, namely surrogate key and natural key. A surrogate key is like a serial number or an incremental ID which serves only for recording and distinguishing rows and does not have a physical meaning. In contrast, a natural key such as a timestamp, email, citizenship IC number, reflects some meaningful information in the real world.

A foreign key is the attribute(s) that link a table to another table. It is often the primary key of another table that in some way links to this table. For example, consider another table `membership_type` as defined in Table 14.2. In the first Table 14.1, column `membership` can be a foreign key which points to `membership_type` in the second Table 14.2.

TABLE 14.1

An example of a relational database table.

user			
user_id*	user_email	membership	referee-id
sunlu	sunlu@xxx.com	premium	NULL
xingzhe	xingzhe@yyy.com	basic	sunlu
...

TABLE 14.2

A second database table in the example.

membership		
membership_type	monthly_price	annual_price
none	0	0
basic	5	50
premium	10	80

The introducing of foreign key helps to maintain the consistency and integrity of the database. For example, when adding a new member to Table 14.1, the DBMS will first check whether the membership type is registered in Table 14.2. If not, the insert operation will be rejected. This guarantees that all registered members have valid membership types.

Notice that a table can have multiple foreign keys. The foreign key can not only relate a table to another, but also relate a table to itself. For example, the “referee-id” attribute in Table 14.1 could be the foreign key that links to “user-id” of the same table.

14.2 Non-Relational Database

Non-relational databases (NoSQL databases) gained their popularity in the 2000s. In contrast to RDBs, NoSQL databases do not store data in tables like an RDB, but in key-value pairs, graphs, documents, or other formats. They are more flexible, efficient and easier to use in some applications than an RDB. Examples of NoSQL databases include *Redis*, *Azure CosmosDB*, *Oracle NoSQL Database*, *Amazon DynamoDB*, *MongoDB*, *AllegroGraph* and many more.

Unlike SQL which applies to almost all RDBMS, there is no universally adopted language for NoSQL DBMS. Each NoSQL DBMS often has its unique query language tailored to its specific data model. We use “NoSQL” to refer to a collective set of (very different and not interchangeable) languages used for

NoSQL databases management. It is impossible to cover all the different types of NoSQL databases and their corresponding DBMS manipulation languages in this notebook. Only brief introductions to some example databases are given in later chapters.

Database services, both RDB and NoSQL, have become critical to our daily life and they are massively deployed on servers. In many applications they work together to deliver the service.

14.3 Structured Query Language

As of this writing, RDB is the most commonly used database, and SQL is its standard manipulating and querying language. They are introduced below.

14.3.1 RDB Naming Conventions

It is recommended that the following naming conventions be applied to databases, tables and columns.

- General rules:
 - Use natural collective terms over plurals, for example, “staff” over “employees”.
 - Use only letters, numbers, and underscores.
 - Begin with a letter and may not end with an underscore.
 - Avoid using abbreviations unless commonly understood.
 - Avoid using prefixes.
- Table:
 - Do NOT use the same name for a table and one of its columns.
 - Do NOT concatenate two table names to create a third relationship table.
- Column:
 - Use singular name for columns.
 - Avoid using over simplified terms such as “id”.
 - Use only lowercase if possible.
- Alias:
 - Use keyword AS to indicate an alias.

- The correlation name should be the first letter of each word of the object name.
- If there is already the same correlation name, append a number.
- Stored procedure: always contain a verb in the name of a stored procedure.
- Uniform suffixes:
 - `_id`: primary key.
 - `_status`: flag values.
 - `_total`: the total number of a collection of values.
 - `_num`: a number.
 - `_date`: a date.
 - `_name`: the name of a person or a product.

14.3.2 General Introduction to SQL

SQL is the most widely used language for interacting with DBMS for data query and maintenance. SQL is very powerful and flexible in its full capability, and it is hardly possible to cover everything in one section. Hence, in this section only the basic SQL operations are introduced.

Notice that the support of different DBMS to SQL may differ slightly. This is because an DBMS may (in fact, very likely) fail to adopt everything in the latest SQL standard. However, most of the commands especially the widely used ones such as creating tables, inserting rows and most of the querying, shall be universally consistent.

SQL is a hybrid language consisting of the following 4 sub-languages.

- Data query language: query information and metadata of a database.
- Data definition language: define database schema.
- Data control language: control user access and permission to a database.
- Data manipulation language: insert, update and delete data from a database.

SQL supports variety of data types, and different DBMS may cover slightly different data types. Some of the most commonly used data types are summarized in Table 14.3. Nowadays, many DBMS supports more complicated types such as objects (the associated database is also known as the object-relational database), and many more. For the full list of data types that a DBMS supports, check the manuals and documents of that DBMS.

SQL defines reserved keywords for database manipulation. The keywords have specific meanings and cannot be used as user-defined variable names. Commonly used SQL keywords are summarized in Tables 14.4, 14.5 and 14.6.

TABLE 14.3

Widely used SQL data types.

Data Type	Description
INT/INTEGER	Integer, with a range of -2147483648 to 2147483647. When marked “UNSIGNED”, the range becomes 0 to 4294967295. Some relevant data types are TINYINT, SMALLINT, MEDIUMINT, and BIGINT, which have different ranges.
DEC/DECIMAL(size,d)	An exact fixed-point number. The total number of digits and the number of digits after decimal point are specified by “size” and “d”, respectively. Some relevant data types are DOUBLE(size,d), which can also be used to specify a floating point. Notice that DEC/DECIMAL is usually preferable in most occasions.
CHAR(size)	A fixed length string with the specified length in characters.
VARCHAR(size)	A variable length string, with the specified maximum string length in characters.
BOOL/BOOLEAN	This is essentially a 1-digit integer, where 0 stands for “false” and otherwise “true”.
BLOB	A binary large object with maximum 65535 bytes.
DATE	A date by format “YYYY-MM-DD”.
TIME	A time by format “hh:mm:ss”.
DATETIME	A combination of date and time by format “YYYY-MM-DD hh:mm:ss”.
TIMESTAMP	A timestamp that measures the number of seconds since the Unix epoch. The format is “YYYY-MM-DD hh:mm:ss”. Unlike DATETIME which is meaningful only if the timezone is known, TIMESTAMP does not rely on timezone.

TABLE 14.4

Widely used SQL keywords (part 1: names).

Keyword	Description
CONSTRAINT	A constraint that limits the value of a column.
DATABASE	A database.
TABLE	A table.
COLUMN	A column (attribute, field) of a table.
VIEW	A view, which is a virtual table that does not store data by itself but only reflects the base tables data.
INDEX	An index, which is a pre-scan of specific column(s) of a table and can be used to speed up future queries related to the column(s). Notice that unlike a view, an index needs to be stored together with the table.
PRIMARY KEY	The primary key of a table.
FOREIGN KEY	A foreign key defined in a table that links to a (different) table.
PROCEDURE	A procedure that defines a list of database operations to be executed one after another

TABLE 14.5

Widely used SQL keywords (part 2: actions).

Keyword	Description
CREATE	Create a database (CREATE DATABASE), a table (CREATE TABLE), a view (CREATE VIEW), an index (CREATE INDEX) or a procedure (CREATE PROCEDURE).
ADD	Add a column in an existing table, or a constraint to an existing column.
ALTER	Modify columns in a table (ALTER TABLE), or a data type of a column (ALTER COLUMN).
SET	Specify the columns and values to be updated in a table.
DROP	Delete a column (DROP COLUMN), a constraint (DROP CONSTRAINT), a database (DROP DATABASE), an index (DROP INDEX), a table (DROP TABLE), or a view (DROP VIEW).
CHECK	Define a constraint that limits the value that can be placed in a column.
DEFAULT	Define a default value for a column.
INSERT INTO	Insert a new row into a table.
UPDATE	Update an existing row in a table.
DELETE	Delete a row from a table.
EXEC	Executes a stored procedure.

TABLE 14.6

Widely used SQL keywords (part 3: queries).

Keyword	Description
SELECT	Query data from a database. Relevant combinations are SELECT DISTINCT which returns only distinct values; SELECT INTO which copies data from one table into another; SELECT TOP which returns part of the results.
AS	Assign an alias to a column or table.
FROM	Specify the table where the query is run.
WHERE	Filter results that fulfill a specified condition.
IN	Specify multiple values in a WHERE clause.
AND	Select rows where both conditions are true.
OR	Select rows where either condition is true.
ALL	Return true if all followed sub-query values meet the condition.
ANY	Return true if any followed sub-query value meet the condition.
BETWEEN	Select values within a given range.
ORDER BY	Sort the results in ascending or descending order.
JOIN	Join tables for query. Relevant combinations are OUTER JOIN, INNER JOIN, LEFT JOIN and RIGHT JOIN.
EXISTS	Tests for the existence of any record in a sub-query.
GROUP BY	Groups the result set when using aggregate functions (COUNT, MAX, MIN, SUM, AVG).
UNION	Combines the result sets of multiple select statements.

14.3.3 General Syntax

Notice that different DBMS may use slightly different syntax for the same or similar function. In the rest of the chapter, unless otherwise mentioned, MySQL/MariaDB syntax is used.

All SQL commands shall end with a semicolon “;”.

The programming of SQL shall follow the following common practices wherever possible. This helps to maintain the good quality and portability of the code.

- Use standard SQL functions over user-defined functions wherever possible for better portability.
- Do NOT use object-oriented design principles in SQL and database schema wherever possible.
- Use UPPERCASE for keywords.
- Use /*<comments>*/ to add comments to the code, otherwise precede comments with -- <comments> and finish them with a new line.

The naming of database, tables and columns shall follow conventions introduced in Section 14.3.1.

During the coding, follow the following rules.

- Use spaces to align the codes.
- Use a space before and after equals (=), and after commas (,).
- Use BETWEEN and IN, instead of combining multiple AND and OR clauses.

When creating a table, follow the following rules.

- Choose standard SQL data types wherever possible.
- Specify default values and set up constraints, and put them close to the declaration of the associated column name.
- Assign primary key carefully and keep it simple.
- Specify the primary key first right after the CREATE TABLE statement.
- Implement validation. For example, for a numerical value, use CHECK to prevent incorrect values.

14.3.4 Database Manipulation

To list down all the databases running on the server, use

```
SHOW DATABASES;
```

To create a database, use

```
CREATE DATABASE <database-name>;
```

To select a database, use

```
USE <database-name>;
```

To delete a database, use

```
DROP DATABASE <database-name>;
```

14.3.5 Table Manipulation

Tables are the fundamental components in an RDB. An example of creating a table using SQL is given below

```
CREATE TABLE <table_name> (
    PRIMARY KEY (<column_name>),
    <column_name_1> <data-type> <constraint>,
    <column_name_2> <data-type> <constraint>,
        CONSTRAINT <constraint-name-1>
        CHECK(<constraint-rule>),
        CONSTRAINT <constraint-name-2>
        CHECK(<constraint-rule>)
);
```

where in this demonstrative table, 2 columns and 2 constraints are defined.

The **<constraint>** that comes after the data type of a column is used to set an additional restriction to the data in the table. When such restriction is violated, an error would raise to stop the operation. For example, if NOT NULL is set as a constraint, then when inserting a row to the table later, the user cannot input NULL for that specific column. Notice that the “primary key” can also be set as a constraint named PRIMARY KEY using this syntax, although it is a better practice to use PRIMARY KEY (<column-name>).

Commonly used such constraints are summarized into Table 14.7. As shown by Table 14.7, a default value can be assigned to a column by using the DEFAULT <value> constraint. When inserting a new row, the column of that row will be assigned to its default value if no other value is assigned. If no such statement is provided for a column, its default value is NULL.

Upon creation of a table, its basic schema can be reviewed using

```
DESCRIBE <table-name>;
```

To list down existing tables, use

```
SHOW TABLES;
```

TABLE 14.7

Commonly used constraints.

Constraint	Description
NOT NULL	Not allowed to be NULL.
UNIQUE	Not allowed to have duplicated values.
PRIMARY KEY	Set as primary key, thus, must be not NULL and must remain unique.
FOREIGN KEY	Set as foreign key.
DEFAULT <value>	Set a default value.
AUTO_INCREMENT = <value>	Each time a new row is inserted and NULL or 0 is set for this column, instead of set the column to NULL or 0, automatically generate the next sequence number. The starting value is defined by <value> which by default is 1.

To delete a table, use

```
DROP TABLE <table-name>;
```

To edit the column of a table, use either of the following

```
ALTER TABLE <table-name>
ADD <column-name> <data-type>; -- add new column
ALTER TABLE <table-name>
DROP COLUMN <column-name>; -- drop column
ALTER TABLE <table-name>
RENAME COLUMN <old-name> TO <new-name>; -- rename column
ALTER TABLE <table-name>
MODIFY COLUMN <column-name> <data-type>; -- modify column data type (
    depending on DBMS, syntax may differ)
```

or

```
ALTER TABLE <table-name>
ADD CONSTRAINT <constraint-name> CHECK(<constraint-rule>); -- add
    constraint
ALTER TABLE <table-name>
DROP CONSTRAINT <constraint-name>; -- drop constraint
```

Notice that it is possible to change the primary key using the above syntax because essentially the primary key is treated as a constraint named “primary”. It is not recommended to do so in general.

The foreign key is a key used to point to another table, in many cases other table’s primary key. Therefore, the foreign key can be nominated only after the other table has been created. Declare foreign key upon creation of a table as follows. As mentioned, to do this, the other tables must be created beforehand. Two methods to declare a foreign key are shown below.

```

CREATE TABLE <table-name> (
    PRIMARY KEY (<column-name>),
    <column-1> <data-type> <constraint>,
    <column-2> <data-type> <constraint>,
        CONSTRAINT <constraint-name-1>
        CHECK(<constraint-rule>),
        CONSTRAINT <constraint-name-2>
        CHECK(<constraint-rule>),
    FOREIGN KEY (<column-name>) REFERENCES <target-table-name>(<target-
        column-name>), -- method 1
        CONSTRAINT <constraint-name-3>
    FOREIGN KEY (<column name>)
        REFERENCES <target-table-name>(<target-column-name>) --
            method 2
);

```

where, as can be seen, foreign key is treated as a constraint in the table.

To define a foreign key in an existing table, use

```

ALTER TABLE <table-name>
ADD FOREIGN KEY (<column name>) REFERENCES <referred-table-name>(<
    referred-column-name>); -- one way
ALTER TABLE <table-name>
ADD CONSTRAINT <constraint-name> FOREIGN KEY (<column name>) REFERENCES
    <referred-table-name>(<referred-column-name>); -- another way

```

To drop a foreign key, use

```

ALTER TABLE <table-name>
DROP CONSTRAINT <constraint-name>;

```

There are variety ways of checking the constraints names of a table. An example is given below.

```

SELECT TABLE_NAME, CONSTRAINT_TYPE, CONSTRAINT_NAME
FROM information_schema.table_constraints
WHERE table_name=<table-name>;

```

One thing to notice is that upon creation of a foreign key, the referred column becomes the “parent” and the foreign key becomes a “child”. As long as the child exists, the parent cannot be removed from its table. This helps to protect the schema of the database. Should there be any quest to break the schema, this restriction can be overwritten. When defining the foreign key, add an additional claim “ON DELETE SET NULL” or “ON DELETE CASCADE” as follows.

```

FOREIGN KEY (<column name>) REFERENCES <referred-table-name>(<referred-
    column-name>) ON DELETE SET NULL
FOREIGN KEY (<column name>) REFERENCES <referred-table-name>(<referred-
    column-name>) ON DELETE CASCADE

```

in the first scenario, the child foreign key will be set to NULL, while in the second scenario, the child relevant rows will be removed.

14.3.6 Row Manipulation

To insert a row into a table, use

```
INSERT INTO <table-name> VALUES (<content>, <content>, ...);
```

where the contents shall follow the field sequence as shown by the DESCRIBE <table-name> command. To specify the column name while inserting a row, use

```
INSERT INTO <table-name>(<column-name>, <column-name>, ...) VALUES (<content>, <content>, ...);
```

Notice that it is also possible to populate multiple rows of a table using one command as follows.

```
INSERT INTO <table-name>(<column-name>, <column-name>, ...)
VALUES (<content>, <content>, ...),
       (<content>, <content>, ...),
       (<content>, <content>, ...);
```

where 3 rows are inserted into the table.

Notice that if a foreign key bound exists between two tables, when inserting a row to the child table, the foreign key value of this row must already be defined in the parent table.

Use the following command to query all items in a table, which can be used to check whether the row is added to the table correctly.

```
SELECT * FROM <table-name>;
```

To modify the attributes of specific row(s), use

```
UPDATE <table-name>
SET <column-name> = <value>, ...
WHERE <filter-criteria>;
```

where <filter-criteria> is used to filter the rows to which the update is carried out. Commonly used filter criteria are a set of <column-name> = <value> separated by AND and OR. The filter criteria can be set very flexibly and more details are given in later sections. Notice that it is possible to change multiple column values together, by stacking multiple <column-name> = <value> separated by “,”. Similarly, to delete rows from a table, use

```
DELETE FROM <table-name>
WHERE <filter-criteria>;
```

Notice that if filter criteria is not specified, i.e., if WHERE is missing, all items in the table will be affected.

14.3.7 Query

A typical query looks like the following and it returns the data in a table-like format.

```
SELECT <column-or-statistics>
FROM <table-name-or-combination>
GROUP BY <column-name>
WHERE <filter-criteria>
ORDER BY <column-name>, ...
LIMIT <number>;
```

where

- **<column-or-statistics>** describes the columns to be returned, and specifies the returning information format.
- **<table-name-or-combination>** describes the source of the information, either being a table, or a joint of multiple tables.
- **GROUP BY <column-name>** groups rows with the same value of the specified column into “summary rows”.
- **<filter-condition>** defines the filter criteria and only rows meet the criteria are returned.
- **ORDER BY <column-name>** allows the items to be returned in a specific order based on ascending/descending order. It is worth mentioning that the **<column-name>** here does not need to appear in the selected returns, and it can be multiple columns separated by “,”. Use **ASC** (default) or **DESC** after each **<column-name>** to specify ascending or descending order.
- **LIMIT <number>** restricts the maximum number of rows to be returned.

Notice that **SELECT** and **FROM** statements are compulsory in all queries, **WHERE** statement very widely used, and other statements optional case by case.

More details are given below.

The statement **<column-or-statistics>** mainly controls the information to be returned. Commonly seen selected items in **<column-or-statistics>** are summarized as follows.

- ***** (asterisk): return all columns.
- **<column-name>**: return selected columns. When multiple fields are returned, use bracket (**<col1>**, **<col2>**, ...). The same applies to other return formats below.
- **<table-name>. <column-name>**: return selected columns, and to avoid ambiguity, specify table name with the column name. This is useful when the source of data is a joint of multiple tables, some of which share the same column name.
- **<column-name> AS <alias>**: return selected columns, and use alias in the returns.
- **DISTINCT <column-name>**: return only distinct rows.

- COUNT(), SUM(), MIN(), MAX(), AVG(): return aggregate function of a column instead of all the items in that column. They can be used along with DISTINCT, for example, COUNT(DISTINCT <column-name>, ...).
- Simple calculations to the above result, for example $1.5 * <\text{column-name}>$. Commonly used arithmetic operations are +, -, *, /, %, DIV (integer division).

When the column is of date time type or interval type, it is possible to use EXTRACT() to select a field of the timestamp or interval. For example,

```
SELECT EXTRACT(YEAR FROM birthday) AS year FROM customer;
```

would look into table `customer`, focus on column `birthday` which should be a datetime type, and extract only `year` from the datetime to return the result.

The statement <table-name-or-combination> mainly indicates the source table(s). It can be a single table, a joint of multiple tables, or a nest query. More details about joint of multiple tables are illustrated below.

Consider the following example, where two tables are given as follows.

```
> SELECT * FROM test;
+-----+-----+-----+
| test_id | value_1 | value_2 |
+-----+-----+-----+
|      1 |     a |      10 |
|      2 |     a |      20 |
|      3 |     a |      30 |
|      4 |     b |     100 |
|      5 |     b |     200 |
|      6 |     c |    1000 |
|      7 |     c |    2000 |
+-----+-----+-----+

> SELECT * FROM test_join;
+-----+-----+-----+-----+
| test_join_id | value_1 | value_2 | value_3 |
+-----+-----+-----+-----+
|       a      |     10 |     99 | alpha   |
|       b      |    100 |    999 | bravo   |
|       d      |  10000 | 99999 | delta   |
+-----+-----+-----+-----+
```

There are different types of joins, namely “inner join” (or “join”), “left join”, “right join” and “cross join”. They are introduced as follows.

The most intuitive join is the cross join. It returns everything in the two tables like a Cartesian product (that explains why cross join is also called Cartesian join), where the total number of columns are the sum of two tables, the number of rows the product of two tables, as shown below.

```
> SELECT * FROM test CROSS JOIN test_join;
```

	test_id	value_1	value_2	test_join_id	value_1	value_2	
							value_3
	1 a	10 a		10 a	99 alpha		
	1 a	10 b		100 b	999 bravo		
	1 a	10 d		10000 d	99999 delta		
	2 a	20 a		10 a	99 alpha		
	2 a	20 b		100 b	999 bravo		
	2 a	20 d		10000 d	99999 delta		
	3 a	30 a		10 a	99 alpha		
	3 a	30 b		100 b	999 bravo		
	3 a	30 d		10000 d	99999 delta		
	4 b	100 a		10 a	99 alpha		
	4 b	100 b		100 b	999 bravo		
	4 b	100 d		10000 d	99999 delta		
	5 b	200 a		10 a	99 alpha		
	5 b	200 b		100 b	999 bravo		
	5 b	200 d		10000 d	99999 delta		
	6 c	1000 a		10 a	99 alpha		
	6 c	1000 b		100 b	999 bravo		
	6 c	1000 d		10000 d	99999 delta		
	7 c	2000 a		10 a	99 alpha		
	7 c	2000 b		100 b	999 bravo		
	7 c	2000 d		10000 d	99999 delta		

where notice that CROSS JOIN can be replaced by a comma “,”.

In this example, value_1 from table test and test_join_id from table test_join are corresponding with each other. In this context, the rows with inconsistent test.value_1 and test_join.test_join_id is meaningless and shall be removed. This can be achieved by the following code.

> SELECT * FROM test CROSS JOIN test_join -> where test.value_1 = test_join.test_join_id;							
+-----+-----+-----+-----+-----+-----+							
	test_id	value_1	value_2	test_join_id	value_1	value_2	
							value_3
	1 a	10 a		10 a	99 alpha		
	2 a	20 a		10 a	99 alpha		
	3 a	30 a		10 a	99 alpha		
	4 b	100 b		100 b	999 bravo		
	5 b	200 b		100 b	999 bravo		

test_id	value_1	value_2	test_join_id	value_1	value_2	value_3
1	a	10	a	10	99	alpha
2	a	20	a	10	99	alpha
3	a	30	a	10	99	alpha
4	b	100	b	100	999	bravo
5	b	200	b	100	999	bravo

and this is equivalent to inner join (or simply, join)

test_id	value_1	value_2	test_join_id	value_1	value_2	value_3
1	a	10	a	10	99	alpha
2	a	20	a	10	99	alpha
3	a	30	a	10	99	alpha
4	b	100	b	100	999	bravo
5	b	200	b	100	999	bravo

where `ON (<table1.column-name> = <table2.column_name>)` is used to indicate the association. The number of columns remain unchanged compared with cross join, but the number of rows is reduced.

From table `test` perspective, its rows regarding `value_1` equals to “a” and “b” are fully included in the inner join results. However, the two rows regarding `value_1=c` is omitted. This is because there is no corresponding row in the other table `test_join` with `test_join_id=c`.

It is possible to “prevent” information loss from table `test` by adding these two rows back, with all the columns from table `test_join` filled with `NULL`. This can be done by left join as follows.

test_id	value_1	value_2	test_join_id	value_1	value_2	value_3
1	a	10	a	10	99	alpha
2	a	20	a	10	99	alpha
3	a	30	a	10	99	alpha
4	b	100	b	100	999	bravo
5	b	200	b	100	999	bravo
6	c	1000	NULL	NULL	NULL	NULL
7	c	2000	NULL	NULL	NULL	NULL

One can think of this as temporarily adding a new row to `test_join` with `test_join_id=c` and everything else NULL, before the inner joining.

The same idea applies to right join as well, as shown below.

```
> SELECT * FROM test RIGHT JOIN test_join ON (test.value_1 = test_join.
   test_join_id);
+-----+-----+-----+-----+-----+-----+
| test_id | value_1 | value_2 | test_join_id | value_1 | value_2 |
|         | value_3 |           |             |         |           |
+-----+-----+-----+-----+-----+-----+
|     1 | a      |    10 | a          |    10 |    99 | alpha   |
|     2 | a      |    20 | a          |    10 |    99 | alpha   |
|     3 | a      |    30 | a          |    10 |    99 | alpha   |
|     4 | b      |   100 | b          |   100 |  999 | bravo  |
|     5 | b      |   200 | b          |   100 |  999 | bravo  |
|  NULL | NULL   |  NULL | d          | 10000 | 99999 | delta   |
+-----+-----+-----+-----+-----+-----+
```

Some DBMS supports “outer join”, which is basically a union of the left and right join results. More about union is introduced later.

The statement <filter-condition> applies filtering to the results. Commonly seen filter criteria <filter-condition> are summarized as follows.

- <column-name> = <value>, where = can be replaced by < (less than), <= (less than or equal to), > (larger than), >= (larger than or equal to) and <> (not equal to).
- <column-name> IN (<value>, <value>, ...)
- <column-name> BETWEEN <value> AND <value>
- <column-name> LIKE <wildcards>, which compares the column value (usually a string) with a given pattern.
- A combination of the above, with AND and OR joining everything together.

A bit more about wildcard query is introduced as follows. A wildcard character is a “placeholder” that represents a group of character(s). Most commonly used wildcard characters in the SQL context include

- _: any single character.
- %: any string of characters (including empty string).
- [<c1><c2> ...]: any single character given in the bracket.
- ^[<c1><c2>...]: any single character not given in the bracket.
- [<c1>-<c2>]: any single character given within the range in the bracket.

Wildcard query can be applied to both CHAR and DATE/TIME types, as they can all be characterized as strings.

The **GROUP BY** groups the rows with the same value of the specified column into “summary rows”. In each summary row, aggregated information is collected. To further explain this, consider the following example.

```
> SELECT * FROM test;
+-----+-----+-----+
| test_id | value_1 | value_2 |
+-----+-----+-----+
| 1 | a | 10 |
| 2 | a | 20 |
| 3 | a | 30 |
| 4 | b | 100 |
| 5 | b | 200 |
| 6 | c | 1000 |
| 7 | c | 2000 |
+-----+-----+-----+
```

Running the following command gives

```
> SELECT * FROM test GROUP BY value_1;
+-----+-----+-----+
| test_id | value_1 | value_2 |
+-----+-----+-----+
| 1 | a | 10 |
| 4 | b | 100 |
| 6 | c | 1000 |
+-----+-----+-----+
```

From the result, it can be seen that summary rows have been created using **GROUP BY**. In the returned table, **value_1** has distinguished values. In this example, it simply picks up the first appearance of the rows in the original table that has distinguished **value_1**, and a lot of information seems to be lost. However, it is worth mentioning that although not displayed, the aggregated information is included.

To verify the presence of the aggregated information, consider running the following command.

```
> SELECT COUNT(*) FROM test GROUP BY value_1;
+-----+
| COUNT(*) |
+-----+
| 3 |
| 2 |
| 2 |
+-----+
```

From the result, we can see that the counted number of each summary row is returned. Similarly, the following SQL returns other aggregation information associated with each summary row.

```
> SELECT value_1, COUNT(*), SUM(value_2) FROM test GROUP BY value_1;
+-----+-----+-----+
| value_1 | COUNT(*) | SUM(value_2) |
+-----+-----+-----+
| a       |      3 |      60 |
| b       |      2 |     300 |
| c       |      2 |   3000 |
+-----+-----+-----+
```

Finally, `ORDER BY` and `LIMIT` controls the sequence and maximum number of returned rows, respectively.

The returns of multiple queries might be able to `UNION` together, if they are union-compatible. Union simply means concatenate the tables vertically. To union the results, use

```
SELECT <...>
UNION
SELECT <...>
UNION
SELECT <...>
...
SELECT <...>;
```

where inside `<...>` are the original query statements. Notice that for the queries to be union-compatible, they must have the same number of columns with identical data type for the associated columns. The names of the column in the returns, if different, follow the first query result. Use alias `AS` to change the names if needed. Duplicated rows in the union will be excluded. If duplications need to be included in the result, certain DBMS provides the `UNION ALL` option.

SQL uses nest queries to add more flexibility. Nest queries plays as the intermediate steps to provide a temporary searching result, from which another query can be executed. Wherever a table name appears in the query, it can be replaced by a `SELECT` statement nested in a bracket “`()`”. A demonstrative example is given below. Consider the same tables `test` and `test_join` as follows.

```
> SELECT * FROM test;
+-----+-----+-----+
| test_id | value_1 | value_2 |
+-----+-----+-----+
| 1 | a | 10 |
| 2 | a | 20 |
| 3 | a | 30 |
| 4 | b | 100 |
| 5 | b | 200 |
| 6 | c | 1000 |
| 7 | c | 2000 |
+-----+-----+-----+
```

```
> SELECT * FROM test_join;
+-----+-----+-----+
| test_join_id | value_1 | value_2 | value_3 |
+-----+-----+-----+
| a           |    10  |     99 | alpha   |
| b           |   100  |    999 | bravo   |
| d           | 10000  | 99999 | delta   |
+-----+-----+-----+
```

A inner join is provided to the above tables. However, for each `value_1` in the first table, the sum of the associated `value_2`, instead of each individual row, is used. This can be achieved using

```
> SELECT temp.value_1 AS type,
      ->          temp.sum_value_2 AS total_value,
      ->          test_join.value_1 AS minval,
      ->          test_join.value_2 AS maxval,
      ->          test_join.value_3 AS abbrev
      -> FROM (SELECT value_1,
      ->             SUM(value_2) AS sum_value_2
      ->           FROM test GROUP BY value_1) AS temp
      -> JOIN test_join
      -> ON (temp.value_1 = test_join.test_join_id);
+-----+-----+-----+
| type | total_value | minval | maxval | abbrev |
+-----+-----+-----+
| a    |       60  |    10  |     99 | alpha   |
| b    |      300  |   100  |    999 | bravo   |
+-----+-----+-----+
```

where notice that alias are quite some times to clarify the logics.

Nest queries can be popular in table joins as well as filter criteria, where the boundary of a variable can be obtained from a nest query.

14.3.8 Trigger

A trigger defines a set of operations to be carried out automatically when something happens to specified tables. For example, in any case a new row is added to a table, a trigger can automatically insert an associated record into another table.

There are mainly 3 types of triggers: DML trigger (triggered by `INSERT`, `UPDATE`, `DELETE`, etc.), DDL trigger (triggered by `CREATE`, `ALTER`, `DROP`, `GRANT`, `DENY`, `REVOKE`, etc.), and CLR trigger (triggered by `LOGON` event).

A quick DML trigger can be defined as follows.

```
CREATE TRIGGER <trigger-name>
  [BEFORE | AFTER] [INSERT | UPDATE | DELETE] ON <table-name>
  FOR EACH ROW <operation>;
```

where `BEFORE` is often used to validate and modify data to be added to

<table-name>, and AFTER is often used to trigger other changes consequent to this change.

In case multiple operations need to be defined, consider using

```
DELIMITER $$  
CREATE TRIGGER <trigger-name>  
    [BEFORE | AFTER] [INSERT | UPDATE | DELETE] ON <table-name>  
    FOR EACH ROW BEGIN  
        <operation>;  
        ...  
        <operation>;  
    END$$  
DELIMITER ;
```

where DELIMITER \$\$ and DELIMITER ; is used to temporarily change the delimiter for the BEGIN...END statement. It is possible to build slightly complicated logics in the operations, for example to build conditional statements.

Use NEW in the operation(s) to represent the rows that is added/updat-ed/deleted from the table-name.

Use the following to drop a trigger.

```
DROP TRIGGER <trigger-name>;
```

14.3.9 SQL Demonstrative Example

An an example to demonstrate the use of SQL, a database is created from scratch. MariaDB is used as the DBMS in this example. More about Mari-aDB is introduced in later Section 15. The database is used in the smart home project to track the resources obtained and consumed by the user. The resources in this context may refer to groceries bought from the supermarket, books purchased online, subscriptions of magazines and services, etc. For simplicity, the prompt is ignored in the rest of this section.

Check the existing databases as follows.

```
SHOW DATABASES;
```

A database named `smart_home` is created as follows.

```
CREATE DATABASE smart_home;
```

Select the database as follows.

```
USE smart_home;
```

With the above command, `smart_home` is selected as the current database.

Based on the database schema design, a few tables need to be created. We shall start with creating `asset`, `accessory`, `consumable` and `subscription` tables as follows.

The `asset` table is used to trace assets in the home. They are often expensive and comes with a serial number or a warranty number, and shall

persist for a long time (a few years, at minimum). Examples of assets include beds, televisions, computers, printers, game consoles. The **accessory** table is used to trace relatively cheaper accessories than assets. Though they are designed to last long, they may not have an serial number. Examples of accessories include books, charging cables, coffee cups. The **consumable** table is used to trace items that is meant to be used up or expire. Examples of consumable items include food, shampoo, A4 printing paper. And finally the **subscription** table is used to trace subscriptions of services. Examples of these services include software license (either permanent license or annual subscription license), magazine subscriptions, membership subscriptions, and digital procurement of a movie.

The serial number or warranty number for assets are used as the primary key of **asset** table. For the other three tables, surrogate keys are used. Each table has a column **product_type_id** that specifies the type of the item, such as “television”, “cooker”, “fruit”, “software”. The types in these tables are given by integer indices. A separate **product_type** relates the indices with their associated meanings. The same applies to **product_brand_id** and **payment_method_id**.

Create asset table as follows.

```
CREATE TABLE asset (
    PRIMARY KEY (serial_num),
    serial_num          VARCHAR(50)  NOT NULL,
    product_type_id    INT(5),
    product_brand_id   INT(5),
    product_name        VARCHAR(50)  NOT NULL,
    receipt_num         VARCHAR(50),
    procured_date      DATE        NOT NULL DEFAULT (
        CURRENT_DATE),
    procured_price     DECIMAL(10,2),
    payment_method_id  INT(5),
    warranty_date_1    DATE        NOT NULL DEFAULT (
        CURRENT_DATE),
    warranty_date_2    DATE        NOT NULL DEFAULT (
        CURRENT_DATE),
    expire_date        DATE        NOT NULL DEFAULT
        '9999-12-31',
    CONSTRAINT warranty_after_procured
    CHECK(warranty_date_1 >= procured_date AND warranty_date_2 >=
        warranty_date_1),
    CONSTRAINT expire_after_procured
    CHECK(expire_date >= procured_date)
);
```

where

- **serial_num**: the serial number, MAC number or registration ID that can be used to uniquely identify the asset.

- `product_type_id`: type index.
- `product_brand_id`: brand index.
- `product_name`: full name of the product that can uniquely specify the asset on the market.
- `receipt_num`: receipt and/or warranty number.
- `procured_date`: date of procurement.
- `procured_price`: price of the product as procured.
- `payment_method_id`: payment method.
- `warranty_date_1`: warranty expiration date (free replace or repair); leave it as the procured date if no such warranty is issued.
- `warranty_date_2`: second warranty expiration date (partially covered repair); leave it as the procured date if no such warranty is issued.
- `expire_date`: the date when the asset expires or needs to be returned. For example, in Singapore a car “expires” in 10 years from the day of procurement.

Notice that constraints and default values have been added to the table creation. An SQL script is used contain the code, and

```
$ mariadb -u <user-name> -p < <script-name>
```

is used to execute the script, which is more convinient than typing all the lines in the MariaDB console.

Similarly, create the rest 3 tables for the resources as follows.

```
CREATE TABLE accessory (
    PRIMARY KEY (item_id),
    item_id          INT(5)      AUTO_INCREMENT,
    product_type_id INT(5),
    product_brand_id INT(5),
    product_name    VARCHAR(50) NOT NULL,
    receipt_num     VARCHAR(50),
    procured_date   DATE        NOT NULL DEFAULT (
        CURRENT_DATE),
    procured_number DECIMAL(10,2) NOT NULL DEFAULT 1.00,
    procured_unit_price DECIMAL(10,2),
    procured_price   DECIMAL(10,2),
    payment_method_id INT(5),
    expire_date     DATE        NOT NULL DEFAULT
        '9999-12-31',
    CONSTRAINT expire_after_procured
    CHECK(expire_date >= procured_date)
);
```

```
CREATE TABLE consumable (
    PRIMARY KEY (item_id),
    item_id          INT(5)      AUTO_INCREMENT,
    product_type_id INT(5),
    product_brand_id INT(5),
    product_name     VARCHAR(50) NOT NULL,
    receipt_num      VARCHAR(50),
    procured_date    DATE       NOT NULL DEFAULT (
        CURRENT_DATE),
    procured_number   DECIMAL(10,2) NOT NULL DEFAULT 1.00,
    procured_unit_price DECIMAL(10,2),
    procured_price    DECIMAL(10,2),
    payment_method_id INT(5),
    expire_date      DATE       NOT NULL DEFAULT (
        CURRENT_DATE),
    CONSTRAINT expire_after_procured
    CHECK(expire_date >= procured_date)
);

CREATE TABLE subscription (
    PRIMARY KEY (item_id),
    item_id          INT(5)      AUTO_INCREMENT,
    product_type_id INT(5),
    product_brand_id INT(5),
    product_name     VARCHAR(50) NOT NULL,
    receipt_num      VARCHAR(50),
    procured_date    DATE       NOT NULL DEFAULT (
        CURRENT_DATE),
    procured_price   DECIMAL(10,2),
    payment_method_id INT(5),
    expire_date      DATE       NOT NULL DEFAULT (
        CURRENT_DATE),
    CONSTRAINT expire_after_procured
    CHECK(expire_date >= procured_date)
);
```

Create the tables for users, product types, product brands and payment methods as follows.

```
CREATE TABLE user (
    PRIMARY KEY (user_id),
    user_id          INT(5),
    first_name       VARCHAR(50) NOT NULL,
    last_name        VARCHAR(50) NOT NULL,
    email            VARCHAR(50) NOT NULL UNIQUE
);

CREATE TABLE product_type (
    PRIMARY KEY (product_type_id),
```

```

    product_type_id      INT(5)      AUTO_INCREMENT,
    product_type_name    VARCHAR(50) NOT NULL UNIQUE,
    product_type_name_sub VARCHAR(50) NOT NULL DEFAULT ('na')
);

CREATE TABLE product_brand (
    PRIMARY KEY (product_brand_id),
    product_brand_id      INT(5)      AUTO_INCREMENT,
    product_brand_name    VARCHAR(50) NOT NULL UNIQUE
);

CREATE TABLE payment_method (
    PRIMARY KEY (payment_method_id),
    payment_method_id      INT(50)      AUTO_INCREMENT,
    user_id                INT(5),
    payment_method_name    VARCHAR(50) NOT NULL
);

```

Finally, create foreign keys as follows.

```

ALTER TABLE payment_method
ADD FOREIGN KEY (user_id)
REFERENCES user(user_id);
ALTER TABLE asset
ADD FOREIGN KEY (product_type_id)
REFERENCES product_type(product_type_id);
ALTER TABLE asset
ADD FOREIGN KEY (product_brand_id)
REFERENCES product_brand(product_brand_id);
ALTER TABLE asset
ADD FOREIGN KEY (payment_method_id)
REFERENCES payment_method(payment_method_id);
ALTER TABLE accessory
ADD FOREIGN KEY (product_type_id)
REFERENCES product_type(product_type_id);
ALTER TABLE accessory
ADD FOREIGN KEY (product_brand_id)
REFERENCES product_brand(product_brand_id);
ALTER TABLE accessory
ADD FOREIGN KEY (payment_method_id)
REFERENCES payment_method(payment_method_id);
ALTER TABLE consumable
ADD FOREIGN KEY (product_type_id)
REFERENCES product_type(product_type_id);
ALTER TABLE consumable
ADD FOREIGN KEY (product_brand_id)
REFERENCES product_brand(product_brand_id);
ALTER TABLE consumable
ADD FOREIGN KEY (payment_method_id)
REFERENCES payment_method(payment_method_id);

```

```
ALTER TABLE subscription
ADD FOREIGN KEY (product_type_id)
REFERENCES product_type(product_type_id);
ALTER TABLE subscription
ADD FOREIGN KEY (product_brand_id)
REFERENCES product_brand(product_brand_id);
ALTER TABLE subscription
ADD FOREIGN KEY (payment_method_id)
REFERENCES payment_method(payment_method_id);
```

14.4 General Ways to Connect to RDB

This section discusses the tools or program interfaces used along with the DBMS. Many software programs provide interface or toolkit to connect to a database. For example, MATLAB uses database toolbox to connect to SQLite or Microsoft SQL server. Python, as a “glue” language, also provide variety of packages to connect to different databases. IDEs such as VSCode can connect to databases using extensions.

Python provides variety of libraries to access RDS, many of which use embedded SQL codes to interact with the DBMS. Depending on the DBMS, different libraries and commands can be used, some of which more general and the other more specific to a particular DBMS.

In this section, both `pandas` and `mariadb` libraries are introduced. The `pandas` library provides data manipulation and analysis tools, and it provides `pandas.io.sql` that allows connecting to a DBMS and embedding SQL commands into the python code. The `mariadb` library, on the other hand, is dedicated for MariaDB connection. Like `pandas.io.sql`, it also allows embedding SQL commands to interface the DMBS.

As pre-requisites, make sure that the following has been done.

- The DBMS has been configured to allow remote access.
- Make sure that an account has been registered in DBMS that has the privilege of operation from a remote machine.
- Make sure that the firewall configuration is correct.

For DBMS configuration, in the case of MariaDB, use the following code in the shell to check the location of the configuration files.

```
$ mysqld --help --verbose
```

Typical locations of the configuration files are `/etc/my.cnf` and `/etc/mysql/my.cnf`. In the configuration file, use the following to disable binding address.

```
[mysqld]
skip-networking=0
skip-bind-address
```

For account setup, in the DBMS console, use something like

```
> GRANT ALL PRIVILEGES ON *.* TO '<user-name>'@<ip-address>
    IDENTIFIED BY '<user-password>' WITH GRANT OPTION;
```

where '*<ip-address>*' is the remote machine that runs the Python program. If the Python codes are running locally, simply use 'localhost'.

It might be necessary to install MariaDB database development files in the DMBS host machine for the Python libraries to be introduced to function properly. Install the development as follows.

```
$ sudo apt install libmariadb-dev
```

PANDAS Library

Python library **pandas** is one of the essential libraries for data analysis. It provides flexible interfaces and tools for data reading and processing and works very well with different data formats and engines including CSV, EXCEL and DBMS. This section focuses mainly on the interaction of **pandas** with DBMS. Therefore, the detailed use of **pandas** for data analysis, etc., are not covered in this section.

A class **pandas.DataFrame** is defined in **pandas** as the backbone to store and process data. The data attribute of **pandas.DataFrame** is a **numpy** array. Many functions are provided to read different data formats into **pandas** data frame, which makes reading data easy and convenient. An example of reading a CSV file is given below.

```
import pandas as pd
df = pd.read_csv(<file-name>)
print(df)
print(df.head(<number>))
print(df.tail(<number>))
print(df.info())
```

where **head()**, **tail()** gives the first and last rows of the data frame, and **info()** checks the data frame basic information including shape and data types of the columns. Check **df.columns** for all the columns of the data frame. Details specific to a column can be accessed via **df.[<column-name>]**. Many functions are provided to further abstract the details, such as grouping and counting. Use **df.loc(<row-index-list>, <column-name-list>)** to check the content of specified rows and columns.

With **pandas** and other relavent libraries, Python can connect to a database and execute a query. An example of using **pandas** to connect to an Microsoft SQL server and implement a query is given below. Notice that different DBMS may require different database connectivity driver standards, and there are mainly two of them, namely open database connectivity (ODBC)

and Java database connectivity (JDBC). Microsoft adopts ODBC, and a separate package is required in the Python program to connect to the Microsoft SQL server.

```
import pyodbc
import pandas.io.sql as psql

server = "<server-url>,<port>"
database = "<database>"
uid = "<uid>"
pwd = "<pwd>"
driver = "<driver>" # such as "{ODBC Driver 17 for SQL Server}"

# connect to database
conn = pyodbc.connect(
    server = server,
    database = database,
    uid = uid,
    pwd = pwd,
    driver = driver
)

# get cursor
cursor = conn.cursor()

# execute sql command
query = """<query>"""
runs = psql.read_sql_query(query, conn)
```

The above codes returns a data frame corresponding to the result set of the query string, which is saved in `runs`.

MARIADB Library

Use the following code to test connectivity from Python to the database.

```
import mariadb
import sys

user = "<user>"
password = "<password>"
host = "<server-url>"
port = "<port>" # MariaDB default: 3306
database = "<database>"

# connect to database
try:
    conn = mariadb.connect(
        user = user,
        password = password,
        host = host,
```

```
port = port,
database = database
)
except mariadb.Error as e:
print(f"Error connecting to MariaDB Platform: {e}")
sys.exit(1)

# get cursor
cur = conn.cursor()

# execute sql command
cur.execute("<sql-command>")
```

Notice that for query, the result is stored in the cursor object. Use a for loop to view the results.

15

RDB Example: MySQL and MariaDB

CONTENTS

15.1	Installation	178
15.2	MariaDB Basic Configuration	178
15.3	DBMS Console	179
15.4	Run SQL File	179

Commonly seen DBMS examples include Oracle Database, MySQL, Microsoft SQL Server, PostgreSQL, SQLite, MariaDB, and many more. Some of them are briefly introduced in this section. Here are some examples.

MariaDB and MySQL are two widely used relational DBMS. MariaDB is initially a fork of MySQL, and in this sense they share many similarities. While MySQL moves towards a dual license approach (free community license and paid enterprise license with proprietary code), MariaDB is designed to be fully open-source under GNU license, and plays as a replacement of MySQL.

In general, MariaDB supports a larger varieties of data engines and new features, and it is claimed to be faster, more powerful and advanced than MySQL. However, it lacks some of the enterprise features provided by MySQL. The users can gain some of these features by using open-source plugins. The most recent new features in MySQL and MariaDB are also diverging.

Some of the main features of MySQL and MariaDB are summarized as follows.

- Good performance (very fast) in general, for a medium size database. Hence, it is popular in many web applications.
- Ease of use.
- Supports in-memory tables to handle read-heavy write-lite tasks.
- Not very flexible, as compared to PostgreSQL where more complex data types, queries, and functions add-ons are supported.

Different databases may propose different minimum system requirements. There is no standard on how these minimum requirements are calculated. Therefore, it is often unfair to directly compare the requirements of different databases. Nevertheless, a summary table is given in Table 15.1. Notice that this is merely an estimation and can differ largely from practice.

TABLE 15.1

An estimation of DBMS hardware requirements.

	OS	Minimum Requirements		
		CPU	RAM	Disk*
MySQL (E)	All*	2 core	2 GB	1.3 GB
MySQL (C)	All	1 core	1 GB	1.3 GB
MariaDB	All	1 core	400MB	660 MB
PostgreSQL	All	1 core	1 GB	40MB
Firebird	All	1 core	12MB	15MB

* Some installations may come with default test database and logs. The disk usage can be reduced if those files are removed after installation.

** “All” refers to Linux, MacOS and Windows. Different platforms may have different minimum requirements.

In the remaining of this section, MariaDB is used for demonstration.

15.1 Installation

To install MariaDB, follow the instruction on the official website. Different OS, such as RHEL and Ubuntu, may require different ways of installation. For example, on RHEL

```
$ sudo yum update
$ sudo yum install mariadb-server
```

and on Ubuntu,

```
$ sudo apt update
$ sudo apt install mariadb-server
```

Notice that MySQL can be installed instead by replacing `mariadb-server` with `mysql-server`. Similarly, replacing `mariadb` with `mysql` in the rest of this section, wherever applicable, would work for MySQL.

MariaDB server can be controlled using `systemctl` which is introduced in Section 9.1. For example, to start MariaDB, use

```
$ sudo systemctl start mariadb.service
```

and to check its status, use

```
$ sudo systemctl status mariadb
```

15.2 MariaDB Basic Configuration

After installation of and starting MariaDB, use

```
$ sudo mysql_secure_installation
```

to run a quick security-related configuration such as creating password for the root user, and deleting test database.

Log in to MariaDB console using

```
$ sudo mariadb
```

Notice that when *sudo* privilege is used, the user should be brought to the root account of the DBMS. Without sudo privilege, a user name and a password is required to log in to the DBMS as follows.

```
$ mariadb -u <user-name> -p  
Enter Password:
```

Depending on the setup, remote log in to the database from other machine, especially using root account, might be forbidden.

15.3 DBMS Console

After logging in to MariaDB console, a prompt that looks like the following would show up.

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

from where an admin account can be created as follows.

```
MariaDB [(none)]> GRANT ALL PRIVILEGES ON *.* TO '<user-name>'@'  
localhost' IDENTIFIED BY '<user-password>' WITH GRANT OPTION;  
MariaDB [(none)]> FLUSH PRIVILEGES;
```

By using an admin account instead of the root account in daily operations, the security risk is reduced.

Notice that remote access to a database is by default forbidden. A command similar with the above is required to enable remote access. Wildcard expression can be used for the IP address, if necessary, to allow a user to access the database from multiple machines.

To check existing users and their IP addresses to whom access has been granted, use

```
SELECT host, user FROM mysql.user;
```

Finally, use

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

to quite MariaDB console.

15.4 Run SQL File

In many occasions, instead of executing SQL commands line by line, an SQL file is prepared in advance and the DBMS is asked to execute the SQL file as a whole. In the Linux shell, execute the following command and MariaDB shall be able to execute an SQL file on the specified host, on behalf of the user, on the selected database.

```
$ mariadb --host="mysql_server" --user="user_name" --database="database_name" --password="user_password" < "file.sql"
```

If the user already logs into the MariaDB console, use the following command instead.

```
MariaDB [(none)]> source file.sql
```

16

RDB Example: PostgreSQL

CONTENTS

16.1	Brief Introduction	181
16.2	Installation and Authentication	182
16.2.1	Installation	182
16.2.2	Regular Authentication	183
16.2.3	Peer Authentication	183
16.2.4	Run PostgreSQL in Containers	184
16.3	PostgreSQL-Specific New Data Types	185
16.4	PostgreSQL User-Defined Data Types	185
16.5	PostgreSQL Stored Procedural and Functions	186
16.6	Manipulation and Query	188
16.7	Non-RDB Example: MongoDB	189
16.7.1	Installation	191
16.7.2	Basic Global Operations	191
16.7.3	Data Format	192
16.7.4	Create Collection and Document	193
16.7.5	Query	195
16.7.6	Update and Remove Document	195
16.7.7	Sharding and Indexing	196
16.7.8	Other Features	197
16.8	Non-RDB Example: Redis	198
16.9	Non-RDB Example: AllegroGraph	198
16.10	Database Accessory	198
16.10.1	RDB Interface with Python	199

PostgreSQL (or Postgres) is a powerful open-source relational database project. It is gaining a lot of popularity recently.

16.1 Brief Introduction

PostgreSQL is claimed to be the world's most advanced open-source RDB, and it has some great features. It is a object-relational database where the user can

define customized data types in the form of objects. The database functions are modularized, meaning that it has a small base installation (only 40MB) as given in Table 15.1, and the user can expand its capability by installing additional modules per required. It complies very well with the latest SQL standard, with additional powerful add-ons.

Though powerful and efficient, PostgreSQL has not grown as popular as some other databases such as Oracle, Microsoft SQL server, and MySQL. One of the reasons for this is the lack of enterprise tier support. Being under PostgreSQL license (an open-source license similar to BSD and MIT), the available support is mostly from the community. With that being said, PostgreSQL is mature and has been adopted in some large enterprises. Of course the IT department of these companies need to work hard to maintain the database. In addition, great flexibility often means a steeper learning curve, making mastering PostgreSQL generally more difficult than other aforementioned databases.

However, with the populating use of microservices where the database efficiency and performance becomes absolutely critical, PostgreSQL is gaining more and more attentions. It is especially popular when using inside containers.

Some of the main features of PostgreSQL are summarized as follows.

- Object-relational database.
- Excellent adherence to SQL standard.
- Excellent performance and scalability.
- Multi-version concurrency control.
- Modularization of features, i.e., light weight base installation and scalable add-ons.
- Disk-based database, and does not support in-memory tables. As a compensation, it has robust caching mechanisms that speed up reading and writing.
- Steep learning curve.

16.2 Installation and Authentication

The installation of PostgreSQL depends on the OS. As of this writing, the latest version of RHEL 9.4 is used as an example. Both installation and authentication to the database are introduced.

16.2.1 Installation

PostgreSQL supports a large range of operating systems including Linux, macOS, Windows, and more. The source code and very detailed documentations of PostgreSQL are available on its websites and can be downloaded to a local machine freely. Due to the lite weight of the base installation, PostgreSQL can be used conveniently in containers. The associated images can be found on Docker Hub.

To install PostgreSQL directly on a host machine, simply follow the instructions on the official website, where command line tools are provided for Linux users, and installer for windows users. For example, to install and enable PostgreSQL 16 on RHEL 9.4 running on `x86_64`, use

```
$ sudo dnf module install postgresql:16/server  
$ sudo postgresql-setup --initdb  
$ sudo systemctl start postgresql.service  
$ sudo systemctl enable postgresql.service
```

After installation, use command `pg_ctl` to control the server, and `psql` to log in to the server. Notice that PostgreSQL installation may come with PgAdmin, the GUI for the DBMS, which can also be used to interact with the databases.

16.2.2 Regular Authentication

The default “root” user of a PostgreSQL database is “`postgres`”. A database of the same name “`postgres`” is also created automatically. A sudoer can log in to this account as follows.

```
$ sudo -u postgres psql postgres  
postgres=#
```

where `postgres=#` is the prompt of the DBMS console.

After logging in to the database as user `postgres`, the user can create users and databases using

```
postgres=# CREATE ROLE <username> LOGIN PASSWORD '<password>';  
postgres=# CREATE DATABASE <database name> WITH OWNER = <username>;
```

and log in to the database as the user by

```
$ psql -h localhost -d <database name> -U <username> -p <port>
```

where `<port>` is defined in the configuration file and it is 5432 by default.

16.2.3 Peer Authentication

Given that the user already logged in to the OS with his username in the OS, it is possible to use peer authentication if the login is from the local host and the PostgreSQL username is the same with the OS username.

Quickly create a user in PostgreSQL with the same username as OS using

```
$ sudo -u postgres createuser -s $USER
```

which essentially logs in as PostgreSQL and creates a user of \$USER. With the user created, create a database for the user using

```
$ createdb <database name>
```

Then log in to PostgreSQL using peer authentication as follows

```
$ psql -d <database name>
<username>=>
```

where <username>=> is the prompt to a regular user

If the database name happens to be the same with \$USER, simply use

```
$ psql
<username>=>
```

to login to the database.

After logging in to the database, use \du and \l to check the users and the databases in PostgreSQL, and use SHOW port; to check the port that PostgreSQL is running on which by default should be 5432.

16.2.4 Run PostgreSQL in Containers

To use PostgreSQL in containers, either download and run PostgreSQL images from Docker Hub, or create a Dockerfile like the following that installs and configures PostgreSQL on top of an Alpine base image. An example of such Dockerfiles is given below.

```
FROM alpine
RUN apk update
# install postgresql
RUN apk add postgresql
RUN mkdir /run/postgresql
RUN chown postgres:postgres /run/postgresql/
USER postgres
WORKDIR /var/lib/postgresql
RUN mkdir /var/lib/postgresql/data
RUN chmod 0700 /var/lib/postgresql/data
RUN initdb -D /var/lib/postgresql/data
# prepare user scripts
RUN mkdir /var/lib/postgresql/user-scripts
RUN chmod 0700 /var/lib/postgresql/user-scripts
COPY ./start.sh /var/lib/postgresql/user-scripts
COPY ./setup_db.sql /var/lib/postgresql/user-scripts
COPY ./populate_db.py /var/lib/postgresql/user-scripts
# prepare user data
RUN mkdir /var/lib/postgresql/user-data
RUN chmod 0700 /var/lib/postgresql/user-data
COPY ./google_stock_price.csv /var/lib/postgresql/user-data
```

```
#  
CMD ["/bin/sh", "/var/lib/postgresql/user-scripts/start.sh"]
```

16.3 PostgreSQL-Specific New Data Types

PostgreSQL supports a large range of data types, more than other databases such as MySQL and MariaDB. In addition to the commonly seen numeric types, character types, date and time types and boolean type, the following data types are supported.

- Currency (monetary) types.
- Geometric types, including points, line segments, boxes, paths, polygons and circles.
- Network address types, such as IPv4 and IPv6 addresses and MAC addresses.
- Array types and associated functions such as accessing, modifying and searching arrays.
- Composite types and associated functions.
- Many more.

16.4 PostgreSQL User-Defined Data Types

User-defined data types can be used to demonstrate the object-relational database aspect of PostgreSQL. It essentially means that the attribute of an element can be an object, and can have some complex and comprehensive features.

To create customized data types, use the following syntax

```
/*Composite Types*/  
CREATE TYPE name AS  
( [ attribute_name data_type [ COLLATE collation ] [, ... ] ] );  
  
/*Enumerated Types*/  
CREATE TYPE name AS ENUM  
( [ 'label' [, ... ] ] );  
  
/*Range Types*/
```

```

CREATE TYPE name AS RANGE (
    SUBTYPE = subtype
    [ , SUBTYPE_OPCLASS = subtype_operator_class ]
    [ , COLLATION = collation ]
    [ , CANONICAL = canonical_function ]
    [ , SUBTYPE_DIFF = subtype_diff_function ]
    [ , MULTIRANGE_TYPE_NAME = multirange_type_name ]
);

/*Base Types*/
CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [ , RECEIVE = receive_function ]
    [ , SEND = send_function ]
    [ , TYPMOD_IN = type_modifier_input_function ]
    [ , TYPMOD_OUT = type_modifier_output_function ]
    [ , ANALYZE = analyze_function ]
    [ , SUBSCRIPT = subscript_function ]
    [ , INTERNALLENGTH = { internallength | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
    [ , LIKE = like_type ]
    [ , CATEGORY = category ]
    [ , PREFERRED = preferred ]
    [ , DEFAULT = default ]
    [ , ELEMENT = element ]
    [ , DELIMITER = delimiter ]
    [ , COLLATABLE = collatable ]
);

```

For example,

```

CREATE TYPE sex_type AS
enum ('M', 'F');

```

which creates a new data type called `sex_type`, and it can take enumerated value of either M or F.

16.5 PostgreSQL Stored Procedural and Functions

Many databases including Oracle SQL, Microsoft SQL Server, MySQL, MariaDB and PostgreSQL, support stored procedures and user defined functions. This feature has been there for a very long time, but it is often not introduced in a typical introductory course. Though useful, they may introduce performance and portability issues, hence need to be handled with caution. Besides,

nowadays it is often regarded as a better practice to implement the logics in the application layer, not in DBMS. Nevertheless, they are briefly introduced as follows.

The following is an example to define a function using SQL.

```
CREATE OR REPLACE FUNCTION add_int(int, int)
RETURNS int AS
'
SELECT $1+$2;
'
LANGUAGE SQL
```

where notice that the input variable types are given in the bracket (use () if there is no input), the output following RETURNS (use void if there is no output), and the SQL operations in between quotations ', which is a delimiter and can be replaced by something else, such as the following

```
CREATE OR REPLACE FUNCTION add_int(int, int)
RETURNS int AS
$body$
SELECT $1+$2;
$body$
LANGUAGE SQL
```

Instead of using \$1, \$2 to refer to a input, names can be assigned together with types as follows.

```
CREATE OR REPLACE FUNCTION add_int(var1 int, var2 int)
RETURNS int AS
$body$
SELECT var1+var2;
$body$
LANGUAGE SQL
```

Notice that so far we have been using SQL as the programming language for the functions, as indicated by LANGUAGE SQL. Notice that PostgreSQL also supports other languages, such as PL/pgSQL, which is a procedural programming language supported by PostgreSQL. It closely resembles Oracle's PL/SQL language. "PL" in these terms represents "Procedural Language".

The following is a list of languages supported by PostgreSQL.

- Naive installation:

- PL/pgSQL
- SQL
- C

- Extension:

- PL/Python

- PL/Perl
- PL/Java
- PL/R
- and more.

SQL is sufficient to carry out simple and straight forward tasks such as adding two numbers, as shown in the earlier example. However, when comes to handling conditional statements and loops, etc., procedural language is required. When the function is complex, it is sometimes impossible or inefficient to implement it using SQL, and PL/pgSQL and other procedural languages can solve this problem. An example is given below.

```
CREATE OR REPLACE FUNCTION increment_value(value INT, increment INT)
RETURNS INT AS $$

DECLARE
    result INT;
BEGIN
    IF increment > 0 THEN
        result := value + increment;
    ELSE
        RAISE EXCEPTION 'Increment must be positive';
    END IF;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

With the above been said, though convenient and powerful it might be in some use cases, it is often a good practice to keep complex logic in application layer, for logic consistency and database portability.

16.6 Manipulation and Query

PostgreSQL adopts SQL for database manipulation and query. SQL has been introduced in earlier sections, hence it is not repeated here. Only selected unique features to PostgreSQL are introduced.

While PostgreSQL server is running, enter its console using `psql` from the shell. One can tell PostgreSQL console by its prompt which looks something like

```
postgres#
```

or

```
postgres>
```

TABLE 16.1

Widely used psql commands.

Command	Description
<code>SELECT VERSION();</code>	Check PostgreSQL version.
<code>\l</code>	List databases.
<code>\c <database></code>	Switch databases.
<code>\d</code>	Describe items. By default, it lists the tables in the current database, and describe each of them.
<code>\dt</code>	List tables.
<code>\dv</code>	List views.
<code>\dn</code>	List schema.
<code>\df</code>	List functions.
<code>\du</code>	List users.
<code>\d <table></code>	Describe a table.
<code>\s</code>	Show command history.
<code>\h</code>	Show help.
<code>\?</code>	Show psql commands.
<code>\!cls</code>	Clear screen.
<code>\q</code>	Quit DBMS shell.

with “postgres” the current selected database. It is also possible to specify user name, default database, and other configurations when connecting to PostgreSQL server. Instead of running `psql` as admin, use

```
$ psql -h <host> -p <port> -U <username> <default_database>
```

Once in PostgreSQL console, use `help` to display the basic commands, including `\copyright` that shows the distribution terms, `\h` to check SQL commands, `\?` to check psql commands, and `\q` to quit PostgreSQL console, etc. Notice that both SQL and psql commands can be used in PostgreSQL console.

Some widely used psql commands are summarized in Table 16.1. Most, if not all, psql commands start with a back slash “\”.

SQL commands, such as creating a database, have been introduced earlier, hence is not repeated here. [HEAD:A Notebook on Linux/chapters/ch-database/ch.tex](#)

16.7 Non-RDB Example: MongoDB

Unlike relational databases where data is stored in relevant tables with pre-designed schematics such as column names, types, and references, non-

relational database stores data in different structures such as directed graphs, dictionaries and documents.

Though relational databases are intuitive and came early in time, non-relational databases are also spreading, as in some occasions they can be easier to use and can be more flexible and powerful. For example, some NoSQL databases are more effective when dealing with massive parallel requests, handling large data, and providing highly scalable and available services.

There are many types of non-relational databases including document-oriented databases, directed graph databases, etc., Each of them has some unique features, pros and cons over other databases, and may adopt its own database manipulation languages. It is impossible to cover everything in this notebook. Only brief introductions of selected databases examples are given here. More details might be found on other relevant notebooks, for example graphical database in *A Notebook on Probability, Statistics and Data Science* as part of the semantic web.

MongoDB is a source-available cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with flexible schemas. This is not surprising, as MongoDB has a close connection with JavaScript. It is powered by a JavaScript engine. MongoDB, together with other JavaScript-relevant software such as *Express.js*, *React.js*, *Node.js*, etc., can be used to create database-powered web applications.

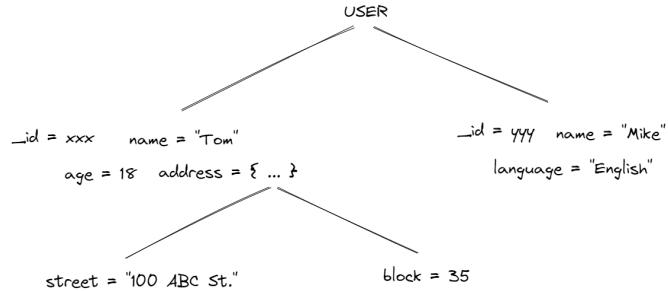
Comparing with conventional RDBs, MongoDB is more capable at

- Massive data storage (in the order of TBs and PBs);
- Frequent and parallel operations such as insert and query;
- Flexible scalability and high availability.

Notice that MongoDB as well as many other NoSQL databases are not suitable to handle financial transactions due to the consistency issue that many NoSQL databases suffer. But things might change as new technologies emerge. Examples of scenarios that MongoDB can be used include

- Posts and streaming management on social media websites or APPs;
- Online gaming information storage;
- Logistics industry and supply chain management;
- IoT data management.

As a document-oriented database, MongoDB stores data not in relational tables, but in separated collections (corresponding with tables) and documents (corresponding with rows). Each document in the same collection can adopt its own schema. When querying data, the user does not need to join collections and map documents. Figure 17.1 is given as a demonstration of how MongoDB stores and organizes data.

**FIGURE 16.1**

A demonstration of MongoDB storing data as object-of-object.

In Fig. 17.1, a collection “USER” is defined. The collection contains two documents. Each document has a few properties. Different document may share common properties such as “name” in this example. Each of them can also have unique properties of its own such as “age”, “address” and “language”. Different properties may have different data types. For example, a property can be string, numeric, or an object, or a list of the above. When querying MongoDB, the DBMS is able to return selected properties of documents that meet specific criteria, and sort them in required order.

16.7.1 Installation

MongoDB, like many other DBMS, has both community and enterprise versions. MongoDB community server can be installed following the instructions in the official website

<https://www.mongodb.com/try/download/community>

To interact with MongoDB DBMS, the quickest way is to use MongoDB shell (also known as “mongosh”). A description of the shell can be found at

<https://www.mongodb.com/try/download/shell>

Notice that when installing MongoDB server, it is possible that the server comes with MongoDB Compass, the GUI for MongoDB DBMS. The GUI can also be used to interact with the databases.

Different versions of MongoDB are available. Choose the correct MongoDB version depending on the CPU and the OS of the machine. MongoDB community server installation size is about 500MB.

MongoDB provides enterprise service, MongoDB Atlas, as part of its cloud solution where user can deploy clusters to host databases. With proper gateway setup, the user can connect to MongoDB Atlas clusters from his local server to retrieve data or to manipulate the databases. MongoDB Atlas is not the covered in this notebook.

TABLE 16.2

MongoDB basic commands.

Command	Description
<code>db.help()</code>	Show a list of methods of <code>db</code> object.
<code>db.version()</code>	Show database server version.
<code>db.getUsers()</code>	Show users.
<code>db.createUser(<content>)</code>	Create a user. The username, password, roles, etc., needs to be included in the <code><content></code> area.
<code>db.dropUser(<username>)</code>	Drop a user.
<code>db.dropDatabase()</code>	Drop current database.
<code>db.status()</code>	Show the basic status of the currently selected database, such as its name, number of collections, storage size, etc.
<code>db.getUsers()</code>	Show users.

16.7.2 Basic Global Operations

After installing both MongoDB server and MonghDB shell, use

```
$ mongosh
```

in the command line to login to the DBMS. JavaScript-like commands are used to manipulate the database, including creating databases and inserting data.

The object `db` contains many methods using which the user can access and modify the basic configurations of the database. For example,

```
> db.version()
```

gives the version of the database server. More commands can be found using `db.help()`. Some commonly used commands are listed in Table 17.1.

To display the existing databases, use

```
> show dbs
```

On a clean installation, the above should return the 3 default databases, `admin`, `config` and `local`. To change to a particular database, use

```
> use <database_name>
```

Notice that there is no “create database” command in MongoDB. To create a database, switch to that database using the above command (even though it does not exist yet), and create some data such as a collection there. The database will be automatically created. This is again a feature closely related to JavaScript.

16.7.3 Data Format

MongoDB is a document-oriented database program. The data is stored in the format of “binary JSON” (BSON), which can be taken as an extension of JSON that supports more data types. Since BSON and JSON have a strong connection to the JavaScript object datatype, which looks very similar to a “dictionary”, one may claim that MongoDB uses key-value pairs to store data. That statement is partially correct because BSON indeed adopts key-value pair structure to store data, but it may over simplifies the reality and be misleading sometimes. In fact, BSON supports much more complicated data types other than string-to-string, such as array and nested objects.

JSON supports 6 datatypes: number, string, boolean (true or false), object, array and null. Notice that JSON is text-based, which is essentially a string. It is a notation of data. It does not concern how the data would be stored and used in the program that takes JSON as the input file. For example, from JSON’s point of view, it does not distinguish different types of number, being integer or float or double. In JSON, it is just a notation of number, say “108”. It is the program’s responsibility to decide how to treat the number, either as an integer, or as a 32-bit float, or a 64-bit float.

BSON, on the other hand, is binary-based, which is essentially a list of binary numbers. This makes BSON more efficient (but less flexible and more difficult to use sometimes) than JSON. In BSON, more datatypes are supported, including: double (64-bit float), string, object, array, binary data (a binary string), object id, boolean, date, null, regular expression, JavaScript code, 32-bit integer, 64-bit integer, timestamp, etc. When using BSON, the user needs to be more specific on how the data should be stored. For example, for a number “108”, the user needs to specify whether to store it as a 32-bit integer or a 64-bit integer, or maybe as a 64-bit float.

Both JSON and BSON are intuitive and human-readable. MongoDB uses BSON due to its enhanced capability.

16.7.4 Create Collection and Document

A MongoDB database contains multiple collections. A collection is similar to a table in an RDB in the sense that it is the “host” of similar data. However, unlike tables where schematics such as column names and datatypes are enforced upon creation of the table, a collection does not enforce fields and data types. An example of creating a collection inside a database is given below.

```
> use testdb;
> db.createCollection("users");
```

Use `show collections` to show the collections in the current database.

Should I use semicolon in the end of each MongoDB command?

If you are using MongoDB shell, then technically speaking, you don't have to. It works both ways. However, if you are using JavaScript environment such as *Node.js* and integrating MongoDB commands as part of the program, you should use semicolon.

For example, to create a new database, in MongoDB shell

```
> use myNewDatabase
```

or

```
> use myNewDatabase;
```

would both work just fine. However, in *Node.js*,

```
const newDb = client.db("myNewDatabase");
```

the semicolon is required.

As a conclusion, semicolon is recommended mostly, just to follow the general JavaScript good practice. Although in JavaScript semicolon is also optional due to Automatic Semicolon Insertion (ASI), it is still widely recommended to use semicolon anyway.

Data can be installed into a collection. An entry to be inserted to a collection, corresponding with a row of a table in RDB, is called a document. It is possible to insert one or multiple entries at a time. To insert documents, first prepare the document in BSON format. For example, consider the following documents.

```
{
  name: "Alice",
  age: 20,
  address: "123 Center Park",
  hobbies: ["football", "reading"],
  parents: {
    father: "Chris",
    mother: "Kite"
  }
}

{
  name: "Bob",
  age: Long.fromNumber(25),
  address: "135 Center Park",
  hobbies: BSON.Array(["basketball", "jogging"])
}
```

Notice that user “Alice” and “Bob” are represented by JSON and BSON, respectively. MongoDB uses BSON internally, but it can also take JSON as input, in which case MongoDB driver converts JSON to BSON.

Then use the following syntax to insert the document into the collection.

```
db.<collection_name>.insertOne({...});
db.<collection_name>.insertMany([{...}, {...}, {...}]);
```

where {...} is the document in JSON or BSON format as shown earlier. To make it more readable, consider do the following instead.

```
const doc = {...};
db.<collection_name>.insertOne(doc);
```

which first store the document in doc, then pass it to `insertOne()` method. Upon successful insertion, an insert id (also known as object id) will be created automatically.

16.7.5 Query

MongoDB uses `find({...})`, `findOne({...})` to find documents, where {...} is a query argument in the form of JavaScript object. Details are given below.

To obtain all the document under a collection, use

```
db.getCollection('<collection_name>').find({});
```

where an empty query simply matches all documents in the collection. Of course, when `findOne()` is used, it will return only one document. When an object is given in the query, MongoDB returns only the documents containing the same fields.

Notice that

```
db.collection_name.some_function()
```

is equivalent to

The first implementation is more convenient while the second more flexible as it supports dynamic naming, i.e., something like

```
collectionName = 'some_collection';
db.getCollection(collectionName).some_function();
```

Several examples are given below. To find documents that has a certain field with certain values, use the following

```
db.getCollection('posts').find({comments: 1})
```

The above query search documents with field `comments` whose value is 1 under collection `posts`. To get those posts with at least 1 comment, use the following

```
db.getCollection('posts').find({comments: {$gt: 0}})
```

where `{$gt: 0}` stands for any value greater than 0. Commonly seen query operators are given in Table 17.2, each with an example.

TABLE 16.3

MongoDB basic query operators.

Operator	Description	Example
\$gt	Greater than	{age: {\$gt: 18}}
\$lt	Less than	{age: {\$lt: 18}}
\$and	And	{\$and: [{age: {\$gt: 18}}, {sex: 'M'}]}
\$or	Or	{\$or: [{age: 18}, {age: 21}]}
\$in	In	{name: {\$in: ['Alice', 'Bob']}}
\$nin	Not in	{name: {\$nin: ['Alice', 'Bob']}}}

TABLE 16.4

MongoDB basic update operators..

Operator	Description & Example
\$set	Replace field value {<query>}, {\$set: {age: 30} }
\$unset	Remove field {<query>}, {\$unset: {age: ""} }
\$inc	Increment a field by a value {<query>}, {\$inc: {age: 1} }
\$rename	Rename field {<query>}, {\$rename: {"age": "yearsOld"}}
\$currentDate	Set field value to current date {<query>}, {\$currentDate: {lastModified: true}}
\$addToSet	Add element to array field {<query>}, {\$addToSet: {hobbies: "reading"}}

16.7.6 Update and Remove Document

Two functions, `updateOne(...)` and `updateMany(...)`, are provided to update documents. Details are given below via an example.

```
db.getCollection('users').updateOne(
  {userId: '0015'},
  {$set: {age: 25}}
);
```

The above code query `users` collection, find the document with `userId` being '`0015`', and change its `age` field to 25. From this example, it can be seen that the document updating function contains a query and a set of updating operators. Commonly seen updating operators are given in Table 17.3.

Use either `deleteOne({<query>})` or `deleteMany({<query>})` to remove documents.

16.7.7 Sharding and Indexing

Given that MongoDB is often used with massive data storage, efficient query from massive storage becomes critical. MongoDB implements a few technologies to speed up the query, including sharding and indexing.

Sharding is a method of distributing data across multiple servers or instances. In MongoDB, a shard consists of a subset of the total dataset, and each shard is responsible for managing a portion of the data. This distribution allows MongoDB to scale horizontally by adding more servers, thereby spreading the load and the data volume across a cluster. When a query is executed, it only needs to be processed by the shards that contain relevant data, rather than the entire dataset. This can significantly reduce query times in a large, distributed system. Sharding is particularly useful for very large datasets and high throughput operations, where a single server would not be sufficient to store the data or provide acceptable performance.

Indexing is a technique used to speed up the retrieval of documents within a database. MongoDB uses indexes to quickly locate data without having to scan every document in a collection. Indexes are a critical component of database optimization, as they can drastically reduce the amount of data MongoDB needs to look through to find documents that match a query.

MongoDB primarily uses B-Tree data structures for its indexes. A B-Tree is a self-balancing tree data structure that maintains sorted data in a way that allows searches, sequential access, insertions, and deletions in logarithmic time. The “B” in B-Tree stands for “balanced” and indicates that the tree is designed to keep the data balanced, ensuring that operations are efficient even as the dataset grows. The B-Tree structure allows MongoDB to perform efficient searches. Instead of scanning every document in a collection, MongoDB can use the B-Tree index to quickly navigate through a small subset of the data to find the documents that match the query criteria.

There are many ways to define the index. For example, it is possible to use a single field to form a single field index, or to use multiple fields to form a compound index. Index by itself is also an argumented data structure and it consumes disk space. Each time there is a write operation, the index needs to be updated. Therefore, a very complicated index schema slows down data insertion. It is critical to design appropriate index to optimize the overall performance of the database.

MongoDB provides commands to check, set and remove indexes. To check the indexes of a system, use

```
db.collection_name.getIndexes()
```

16.7.8 Other Features

MongoDB uses aggregation framework provides way to perform complex data transformations in a “pipeline”. The data is processed step-by-step.

When a document is added to a collection, a default field `_id` is added.

Querying via `_id` is usually faster than with other criteria because otherwise MongoDB has to search for the entire collection to find relevant documents.

Data in MongoDB can be exported or imported from files, such as BSON and JSON files.

MongoDB, like many other DBMS especially cloud-based enterprise tier ones, provide replica setting to tackle read-heavy applications. A primary server is replicated to multiple replicas. Writing is allowed only to the primary server, but reading is allowed to all replicas as well as the primary server. In the case of MongoDB, when primary server fails, one of the replicas is automatically promoted to be the primary server.

16.8 Non-RDB Example: Redis

Redis, short for *REmote DIctionary Server*, is an open-source in-memory distributed key-value database. It is often used as a lightweight database, cache tool, or message broker. Some key features are listed below.

- In-memory storage. This speeds up reading and writing operations.
- Persistence. While primarily it is an in-memory database, it also offers variety of ways to persist data on disk without compromising a lot on performance.
- Complex data structures and associated atomic operations. Though Redis is key-value store, it supports more complex data structures than that.
- High availability via replicas.
- Distributed storage via horizontal partitioning.
- Lightweight.

16.9 Non-RDB Example: AllegroGraph

Graph-based database is elsewhere introduced in details in notebook *A Notebook on Probability, Statistics and Data Science* under semantic web. Check that notebook for more details.

16.10 Database Accessory

This section discusses the tools or program interfaces used along with the DBMS. Many software programs provide interface or toolkit to connect to a database. For example, MATLAB uses database toolbox to connect to SQLite or Microsoft SQL server. Python, as a “glue” language, also provides variety of packages to connect to different databases. IDEs such as VSCode can connect to databases using extensions.

Some of these tools and packages are introduced in this section.

16.10.1 RDB Interface with Python

Python provides variety of libraries to access RDS, many of which use embedded SQL codes to interact with the DBMS. Depending on the DBMS, different libraries and commands can be used, some of which more general and the other more specific to a particular DBMS.

In this section, both `pandas` and `mariadb` libraries are introduced. The `pandas` library provides data manipulation and analysis tools, and it provides `pandas.io.sql` that allows connecting to a DBMS and embedding SQL commands into the python code. The `mariadb` library, on the other hand, is dedicated for MariaDB connection. Like `pandas.io.sql`, it also allows embedding SQL commands to interface the DBMS.

As pre-requisites, make sure that the following has been done.

- The DBMS has been configured to allow remote access.
- Make sure that an account has been registered in DBMS that has the privilege of operation from a remote machine.
- Make sure that the firewall configuration is correct.

For DBMS configuration, in the case of MariaDB, use the following code in the shell to check the location of the configuration files.

```
$ mysqld --help --verbose
```

Typical locations of the configuration files are `/etc/my.cnf` and `/etc/mysql/my.cnf`. In the configuration file, use the following to disable binding address.

```
[mysqld]
skip-networking=0
skip-bind-address
```

For account setup, in the DBMS console, use something like

```
> GRANT ALL PRIVILEGES ON *.* TO '<user-name>'@<ip-address>
    IDENTIFIED BY '<user-password>' WITH GRANT OPTION;
```

where '**<ip-address>**' is the remote machine that runs the Python program. If the Python codes are running locally, simply use '**localhost**'.

It might be necessary to install MariaDB database development files in the DMBS host machine for the Python libraries to be introduced to function properly. Install the development as follows.

```
$ sudo apt install libmariadb-dev
```

PANDAS Library

Python library **pandas** is one of the essential libraries for data analysis. It provides flexible interfaces and tools for data reading and processing and works very well with different data formats and engines including CSV, EXCEL and DBMS. This section focuses mainly on the interaction of **pandas** with DBMS. Therefore, the detailed use of **pandas** for data analysis, etc., are not covered in this section.

A class **pandas.DataFrame** is defined in **pandas** as the backbone to store and process data. The data attribute of **pandas.DataFrame** is a **numpy** array. Many functions are provided to read different data formats into **pandas** data frame, which makes reading data easy and convenient. An example of reading a CSV file is given below.

```
import pandas as pd
df = pd.read_csv(<file-name>)
print(df)
print(df.head(<number>))
print(df.tail(<number>))
print(df.info())
```

where **head()**, **tail()** gives the first and last rows of the data frame, and **info()** checks the data frame basic information including shape and data types of the columns. Check **df.columns** for all the columns of the data frame. Details specific to a column can be accessed via **df.[<column-name>]**. Many functions are provided to further abstract the details, such as grouping and counting. Use **df.loc(<row-index-list>, <column-name-list>)** to check the content of specified rows and columns.

With **pandas** and other relevant libraries, Python can connect to a database and execute a query. An example of using **pandas** to connect to an Microsoft SQL server and implement a query is given below. Notice that different DBMS may require different database connectivity driver standards, and there are mainly two of them, namely open database connectivity (ODBC) and Java database connectivity (JDBC). Microsoft adopts ODBC, and a separate package is required in the Python program to connect to the Microsoft SQL server.

```
import pyodbc
import pandas.io.sql as psql
```

```
server = "<server-url>,<port>"  
database = "<database>"  
uid = "<uid>"  
pwd = "<pwd>"  
driver = "<driver>" # such as "{ODBC Driver 17 for SQL Server}"  
  
# connect to database  
conn = pyodbc.connect(  
    server = server,  
    database = database,  
    uid = uid,  
    pwd = pwd,  
    driver = driver  
)  
  
# get cursor  
cursor = conn.cursor()  
  
# execute sql command  
query = """<query>"""  
runs = psql.read_sql_query(query, conn)
```

The above codes returns a data frame corresponding to the result set of the query string, which is saved in `runs`.

MARIADB Library

Use the following code to test connectivity from Python to the database.

```
import mariadb  
import sys  
  
user = "<user>"  
password = "<password>"  
host = "<server-url>"  
port = "<port>" # MariaDB default: 3306  
database = "<database>"  
  
# connect to database  
try:  
    conn = mariadb.connect(  
        user = user,  
        password = password,  
        host = host,  
        port = port,  
        database = database  
    )  
except mariadb.Error as e:  
    print(f"Error connecting to MariaDB Platform: {e}")  
    sys.exit(1)
```

```
# get cursor
cur = conn.cursor()

# execute sql command
cur.execute("<sql-command>")
```

Notice that for query, the result is stored in the cursor object. Use a for loop to view the results.

===== 9590561686dbab978e33ec2e9fc9dcc148803fe4:A
Notebook on Linux/chapters/part-3/ch`sql'database.tex 9eeb46352eb2580dff8745d93d78e28721fb793

17

NoSQL Database Example: MongoDB

CONTENTS

17.1	Features and Data Model	204
17.1.1	Features	204
17.1.2	Data Model	205
17.2	Installation	206
17.3	MongoDB Shell	207
17.3.1	Basic Operation	207
17.3.2	Create Collection and Document	207
17.3.3	Query	210
17.3.4	Update Document	212
17.3.5	Remove Document	213
17.4	MongoDB Compass	213
17.5	Advanced Query	213
17.5.1	More about <code>find()</code>	213
17.5.2	Sorting	214
17.5.3	Limiting the Number of Returns	214
17.5.4	Limiting the Fields of Returns	215
17.5.5	Counting the Number of Returns	215
17.6	Sharding and Indexing	215
17.6.1	Sharding	216
17.6.2	Indexing	216
17.6.3	Simple and Compound Index	217
17.6.4	Multi-key Index	218
17.6.5	Hide and Remove Index	218
17.6.6	Performance Evaluation with Index	219
17.7	Third-Party Connection	220
17.8	Aggregation Framework	221
17.8.1	Aggregation Pipeline Syntax	222
17.8.2	Stage: <code>\$ match</code>	223
17.8.3	Stage: <code>\$ group</code>	224
17.8.4	Stage: <code>\$ sort</code>	225
17.8.5	Stage: <code>\$ limit</code>	225
17.8.6	Stage: <code>\$ set</code>	225
17.8.7	Stage: <code>\$ count</code>	226
17.8.8	Stage: <code>\$ project</code>	226

17.8.9	Stage: \$ out	228
17.9	MongoDB Atlas	228
17.9.1	MongoDB Atlas Dashboard	229
17.9.2	Connection String	229
17.9.3	Atlas Search	231
17.9.4	Schema Pattern	233

MongoDB is a source-available, cross-platform, general-purpose, document-oriented database program. Classified as a NoSQL database program, MongoDB uses BSON (an extension of JSON) documents with flexible schema to store data. This is not surprising, as MongoDB itself is powered by a JavaScript engine. MongoDB, together with other JavaScript-relevant software such as *Express.js*, *React.js*, *Node.js*, etc., is often used to create database-powered web applications.

Data in MongoDB can be exported or imported from files, such as BSON and JSON files.

17.1 Features and Data Model

MongoDB has some unique features and it applies a different data model compared with RDB and other NoSQL databases.

17.1.1 Features

Comparing with conventional RDBs, MongoDB is better at

- Massive data storage (in the order of TBs and PBs);
- Frequent and parallel operations when performing insert and query;
- Flexible scalability and high availability.

Notice that MongoDB as well as many other NoSQL databases are not suitable to handle financial transactions due to the consistency issue that many NoSQL databases suffer. But things might change as new technologies become handy.

Examples of MongoDB-suitable applications include

- Posts and streaming management on social media websites or APPs;
- Online gaming;
- Logistics industry and supply chain management;
- IoT data management.

As a document-oriented database, MongoDB stores data in documents and collections instead of rows and tables. A document is the basic unit of the storage. A collection is a group of documents on the same/similar topic. Each document in the same collection can adopt its own schema, and it should be self-contained so that when querying the data the user does not need to join collections and map documents. Figure 17.1 gives a demonstrative example of how MongoDB stores and organizes data.

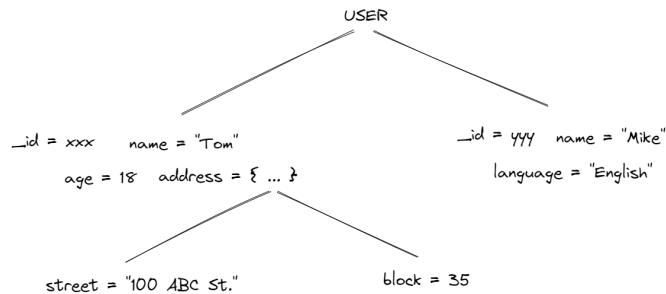


FIGURE 17.1

A demonstrative example of how MongoDB stores data as (nested) object.

In Fig. 17.1, a collection “USER” is defined. The collection contains two documents. Each document has a few properties. Different document may share common properties such as “name” in this example. Each of them can also have unique properties of its own such as “age”, “address” and “language”. Different properties may have different data types. For example, a property can be string, numeric, or a nested object, or an array of the above. When querying MongoDB, the DBMS is able to return selected properties of documents that meet specific criteria, and sort them in the required order.

17.1.2 Data Model

MongoDB is a document-oriented database program. The data is stored in the format of “binary JSON” known as BSON. BSON is an extension of JSON that supports more data types.

JSON VS BSON

JSON supports the following 6 data types: number, string, boolean, object, array, and null. JSON is text-based, which means that it is essentially a string. It is only a notation of data and does not concern itself with how the data is interpreted in the program that processes the JSON file. For example, “0” as a number in a JSON file does not imply whether it is an integer, a 32-bit float, or a 64-bit float.

BSON, on the other hand, is binary-based and hence contains more information. In BSON, more data types are supported, such as different floats, date, timestamp, binary string, regular expression, code, etc. For the same example, “0”, in BSON different number types are compiled and stored differently. This makes BSON more powerful and efficient (but less intuitive) than JSON.

MongoDB uses BSON to store and process data due to its enhanced capability, and uses JSON to display the data.

It is strongly recommended to make the data (document and collection) self-contained. The design of MongoDB architecture should follow depend on how the data would be queried. Data that is accessed together should reside together. This is known as embedding, which adds redundancy to the data storage and make documents bigger, but will make query simpler. Notice that when using embedding, make sure the size of the document is bounded and would not go towards infinity. MongoDB has a maximum 16MB size limitation for each document.

MongoDB database, unlike relational DB, is by nature not designed for joining. However, MongoDB does support referencing documents and even performing joins (to some extent) through different mechanisms, such as \$lookup stage in the aggregation pipe.

17.2 Installation

MongoDB, like many other DBMS, has both community and enterprise versions. MongoDB community server can be installed following the instructions in the official website

<https://www.mongodb.com/try/download/community>

A MongoDB server can host a MongoDB cluster, inside which are many MongoDB databases. Each database can have its own access policy, etc.

To interact with MongoDB DBMS, the quickest way is to use MongoDB shell (also known as “mongosh”). A description of the shell can be found at

<https://www.mongodb.com/try/download/shell>

Notice that when installing MongoDB server, it is possible that the server comes with MongoDB Compass, the GUI for MongoDB DBMS. The GUI can also be used to interact with the databases.

Different versions of MongoDB are available. Choose the correct MongoDB version depending on the CPU and the OS of the machine. MongoDB community server installation size is about 500MB.

MongoDB provides enterprise service, MongoDB Atlas, as part of its cloud solution where user can deploy clusters to host databases. With proper gateway setup, the user can connect to MongoDB Atlas clusters from his local server to retrieve data or to manipulate the databases. MongoDB Atlas is briefly introduced in a later section.

17.3 MongoDB Shell

After installing both MongoDB server and MonghDB shell, use

```
$ mongosh
```

in the command line to login to the DBMS. JavaScript-like commands are used to manipulate the database, such as creating databases and inserting data.

17.3.1 Basic Operation

The object `db` contains many methods using which the user can access and modify the basic configurations of the database. For example,

```
> db.version()
```

gives the version of the database server. More commands can be found using `db.help()`. Some commonly used commands are listed in Table 17.1.

To display the existing databases, use

```
> show dbs
```

On a clean installation, the above should return the 3 default databases, `admin`, `config` and `local`. To switch to a particular database, use

```
> use <database_name>
```

Notice that there is no “create database” command in MongoDB. To create a database, switch to that database using the above command (even though it does not exist yet), and add some data. The database will be automatically created. This is again a feature closely related to JavaScript.

TABLE 17.1

MongoDB basic commands.

Command	Description
<code>db.help()</code>	Show a list of methods of db object.
<code>db.version()</code>	Show database server version.
<code>db.getUsers()</code>	Show users.
<code>db.createUser(<content>)</code>	Create a user. The username, password, roles, etc., needs to be included in the <content> area.
<code>db.dropUser(<username>)</code>	Drop a user.
<code>db.dropDatabase()</code>	Drop current database.
<code>db.status()</code>	Show the basic status of the currently selected database, such as its name, number of collections, storage size, etc.

17.3.2 Create Collection and Document

A MongoDB database contains multiple collections. A collection is similar to a table in an RDB in the sense that it is the collection of data on the same / similar topic. However, unlike tables where schematics such as column names and datatypes are enforced upon creation of the table, a collection does not enforce fields and their data types. The syntax is given below.

```
db.createCollection(<name>, <options>)
```

The <options> field is optional and allows creating different types of collections, such as time-series collection which is optimized to store time-series sensor data. Probable <options> configuration are summarized below.

```
db.createCollection( <name>,
{
    capped: <boolean>,
    timeseries: { // Added in MongoDB 5.0
        timeField: <string>,
        metaField: <string>,
        granularity: <string>,
        bucketMaxSpanSeconds: <number>, // Added in MongoDB 6.3
        bucketRoundingSeconds: <number> // Added in MongoDB 6.3
    },
    expireAfterSeconds: <number>,
    clusteredIndex: <document>, // Added in MongoDB 5.3
    changeStreamPreAndPostImages: <document>, // Added in MongoDB
        6.0
    size: <number>,
    max: <number>,
    storageEngine: <document>,
    validator: <document>,
    validationLevel: <string>,
}
```

```
    validationAction: <string>,
    indexOptionDefaults: <document>,
    viewOn: <string>,
    pipeline: <pipeline>,
    collation: <document>,
    writeConcern: <document>
})
```

If left empty, a normal collection will be created.

An example of creating a collection inside a database is given below.

```
> use testdb;
> db.createCollection("users");
```

Use `show collections` to show the collections in the current database.

Should I use semicolon in the end of each MongoDB command?

If you are using MongoDB shell, then technically speaking, you don't have to. It works both ways. However, if you are using JavaScript environment such as *Node.js* and integrating MongoDB commands as part of the program, you should use semicolon.

For example, to select a database, in MongoDB shell

```
> use myNewDatabase
```

or

```
> use myNewDatabase;
```

would both work just fine. However, in *Node.js*,

```
const newDb = client.db("myNewDatabase");
```

the semicolon is highly recommended by JavaScript.

As a conclusion, semicolon is recommended mostly, just to follow the general JavaScript good practice. Although in JavaScript semicolon is also optional due to Automatic Semicolon Insertion (ASI), it is still widely recommended to use semicolon anyway.

Data can be inserted into a collection in the form of documents. It is possible to insert one or multiple documents at a time. To insert documents, first prepare the document in JSON-like format. For example, consider the following documents.

```
{
  name: "Alice",
  age: 20,
  address: "123 Center Park",
  hobbies: ["football", "reading"],
  parents: {
    father: "Chris",
```

```

        mother: "Kite"
    }
}

{
  name: "Bob",
  age: Long.fromNumber(25),
  address: "135 Center Park",
  hobbies: BSON.Array(["basketball", "jogging"])
}

```

Notice that user “Alice” and “Bob” in the above example are represented by JSON and BSON respectively. MongoDB uses BSON internally, but it can also take JSON as input, in which case MongoDB driver converts JSON to BSON.

Then use the following syntax to insert the document into the collection.

```

db.<collection_name>.insertOne({...});
db.<collection_name>.insertMany([ {...}, {...}, {...} ]);

```

where `{...}` is the document in JSON or BSON format as shown earlier. To make it more readable, consider do the following instead.

```

const doc = {...};
db.<collection_name>.insertOne(doc);

```

which first store the document in `doc`, then pass it to `insertOne()` method. Upon successful insertion, an object id “`_id`” will be created automatically and added to the document as another field. The user can also specify `_id` manually. Notice that `_id` serves as the “primary key”: it must exist and be unique for each document under the same collection.

The above inserting document method creates a collection, if the collection has not been created.

17.3.3 Query

MongoDB uses `find()`, `findOne()` to find documents, where the input is a filter document in the form of a JSON-like string. Details are given below.

To obtain all the document under a collection, use

```

db.getCollection('<collection_name>').find({});

```

where an empty query simply matches all documents in the collection. Of course, when `findOne()` is used, it will return only one document.

TABLE 17.2

MongoDB basic query operators.

Operator	Description	Example
\$eq	Equal	{age: {\$eq: 18}}
\$gt	Greater than	{age: {\$gt: 18}}
\$gte	Greater or equal	{age: {\$gte: 18}}
\$lt	Less than	{age: {\$lt: 18}}
\$lte	Less or equal	{age: {\$lte: 18}}
\$and	And	{\$and: [{age: {\$gt: 18}}, {sex: 'M'}]}
\$or	Or	{\$or: [{age: 18}, {age: 21}]}
\$in	In	{name: {\$in: ['Alice', 'Bob']}}
\$nin	Not in	{name: {\$nin: ['Alice', 'Bob']}}

Two Ways to Refer to a Collection

Notice that

`db.<collection_name>.<function>()`

is equivalent with

`db.getCollection('<collection_name>').<function>();`

The first implementation is more convenient while the second more flexible as it supports dynamic naming.

More examples are given below. Assume that there is a collection `posts`, inside which are documents recording the posts by a blogger. Each post may have some comments. The number of comments are recorded in the field `comments` of the documents. To find documents that has a certain field with certain values, use the following

`db.getCollection('posts').find({comments: 1})`

The above query search documents with field `comments` whose value is 1 (or whose value is an array that contains 1 as its element). under collection `posts`. To get those posts with at least 1 comment, use the following

`db.getCollection('posts').find({comments: {$gt: 0}})`

where `{$gt: 0}` stands for any value greater than 0. Commonly seen query operators are given in Table 17.2, each with an example.

To specifically query for elements in an array, use `$elemMatch` as follows.

`{ <field>: { $elemMatch: { <query1>, <query2>, ... } } }`

For example,

`db.scores.find(
 { results: { $elemMatch: { $gte: 80, $lt: 85 } } })`

TABLE 17.3

MongoDB basic update operators..

Operator	Description	Example
\$set	Add/update field	<code>{\$set: {age: 30}}</code>
\$unset	Remove field	<code>{\$unset: {age: ""}}</code>
\$push	Append to array	<code>{\$push: {age: 25}}</code>
\$inc	Increase a field	<code>{\$inc: {age: 1}}</code>
\$rename	Rename field	<code>{\$rename: {"age": "yearsOld"}}</code>
\$addToSet	Append to array	<code>{\$addToSet: {hobbies: "reading"}}</code>

Notice that `$addToSet` would not append the element if it is already in the array to avoid duplication, while `$push` will append regardless of whether it exists or not.

)

which returns a document only if: it has a field `results`, and the field `results` is an array, and the array contains at least one such element that is greater or equal than 80 and less than 85.

17.3.4 Update Document

Function `updateOne()` is provided to update documents, where the input argument is a tuple containing filtering condition, update and options, following the syntax below.

```
db.<collection_name>.updateOne(filter, update, options)
```

Details are given below via an example.

```
db.getCollection('users').updateOne(
    {userId: '0015'},
    {$set: {age: 25}}
);
```

The above code query `users` collection, find the document with `userId` being '`0015`', and change its `age` field to 25. From this example, it can be seen that the document updating function contains a query and a set of updating operators. Commonly seen updating operators are given in Table 17.3.

Notice that there is another function, `replaceOne()`, that functions similarly with `updateOne()`. The difference is that `replaceOne()` will replace the entire document, removing all the fields (except `_id`) not mentioned in the update, while `updateOne()` allows editing the mentioned update while leaving unmentioned fields untouched.

It is possible to add `{upsert:true}` as the option to `replaceOne()` function, in which case if no document meets the criteria of the filter, it will create an empty document and implement the updates.

To update documents that meet certain criteria all at once, use

`updateMany()` instead. Do note that `updateMany()` is not an atomic operation. It is possible that, for unpredictable reasons, only some of the documents are updated and they will not be rolled back. Use it with caution!

17.3.5 Remove Document

Use either `deleteOne()` or `deleteMany()` to remove documents, with the input the filter document.

17.4 MongoDB Compass

MongoDB Compass is the desktop GUI developed by MongoDB. As a MongoDB client, it can connect to a MongoDB server, either local, remote or MongoDB Atlas, using the connection string. It provides an intuitive user interface that can be used to view and edit the data in the server.

MongoDB Compass comes with analytical tools and data visualization tools that help the user understand the insight of the data as well as the database structure. It allows the user to define and compose aggregation pipelines that automatically abstract useful aggregated information from the data.

17.5 Advanced Query

It been introduced in the earlier section how `find()` `findOne()` can be used to retrieve documents. The input argument to the above functions is the filter document in JSON-like format, and the documents in the collection that match with the filter document will be returned.

More about query is introduced in this section, such as sorting and limiting the results, returning only selected fields, returning aggregated information, etc.

17.5.1 More about `find()`

It is worth mentioning that `find()` returns a cursor, but not the documents themselves. One can iterate through the cursor to get all the documents matching the query. This is different from `findOne()` which directly returns a document.

When viewing the data in MongoDB Compass or Atlas, this does not pose an issue. One can simply take `find()` as a function that returns multiple

documents. However, in the programming interface, it is important to note the differences between a cursor and a document.

The below are two demonstrative examples how a Python program should treat the returns from `find()` and `findOne()` differently. The same idea applies to other programming languages as well.

```
from pymongo import MongoClient

def find_users():
    client = MongoClient('mongodb://localhost:27017/')
    db = client.mydatabase
    collection = db.users

    cursor = collection.find({ 'age': { '$gte': 18 } })

    # Iterate over the cursor
    for doc in cursor:
        print(doc)

    # Alternatively, convert to a list
    results = list(cursor)
    print(results)

    client.close()
```

```
from pymongo import MongoClient

def find_one_user():
    client = MongoClient('mongodb://localhost:27017/')
    db = client.mydatabase
    collection = db.users

    # Find one document
    user = collection.find_one({ 'name': "John Doe" })
    print(user)

    client.close()
```

Since a cursor, instead of the documents themselves, is returned, we can perform further actions on the return such as sorting the result, etc.

17.5.2 Sorting

Use the following syntax to retrieve and sort the return.

```
cursor.sort({<field>:<value>})
```

which sorts the result according to `<field>`. The value “1” or “-1” represents the sorting order, either ascending or descending, respectively.

17.5.3 Limiting the Number of Returns

When it is unnecessary to return all the details of all the documents fulfilling the query criteria, it is possible to limit the field number of returned documents which is often helpful with improving the data processing efficiency.

It is possible to limit the number of documents in the returns to improve the query efficiency. This is often used with the `sort()` method. The syntax is as follows.

```
cursor.sort({<field>:<value>}).limit(<number>)
```

where `<number>` specifies the number of returns.

17.5.4 Limiting the Fields of Returns

Notice that the full syntax of `find` is

```
db.collection.find( <query>, <projection>, <options> )
```

The second argument, `<projection>`, is used to indicate which fields of the documents should be returned. If left blank as by the default, all the fields of the documents will be returned. Notice that the same applies to `findOne()` method.

For example,

```
cursor = collection.find({'age': {'$gte': 18}}, {name:1, age: 1})
```

will return the names and ages of all the users whose age is greater than or equal to 18. By using

```
{ <field1>: <value>, <field2>: <value> ... }
```

with `<value>` be either 1 or 0, one can control which fields to be included or excluded respectively. The value 0 is often used with the `_id` field which is always returned unless specifically required not to. Notice that the `<field>` in the syntax can also contain sub fields, in which case quotation marks "`<field>. <subfield>`" must be used.

With some more advanced setups, `<projection>` can also be used to specify what elements in an array to return, and more. For details, check the user manual of `db.collection.find()` function.

17.5.5 Counting the Number of Returns

Consider counting the number of documents fulfilling a particular filter document. Use

```
db.<collection>.countDocuments( <query>, <options> )
```

17.6 Sharding and Indexing

Given that MongoDB is often used with big data such as web data or IoT data, efficient query from massive data pool becomes critical. MongoDB implements a few technologies to speed up the query such as sharding and indexing.

17.6.1 Sharding

Sharding is a method of distributing data across multiple servers or instances. In MongoDB, a shard consists of a subset of the total dataset, and each shard is responsible for managing a portion of the data. This distribution allows MongoDB to scale horizontally by adding more servers, thereby spreading the load and the data volume across a cluster. When a query is executed, it only needs to be processed by the shards that contain relevant data, rather than the entire dataset. This can significantly reduce query times in a large, distributed system. Sharding is particularly useful for very large datasets and high throughput operations, where a single server would not be sufficient to store the data or provide acceptable performance.

17.6.2 Indexing

Indexing is a technique used to speed up the retrieval of documents within a database. MongoDB uses indexes to quickly locate a document without having to scan every document in a collection. Indexes are a critical component of database optimization, as they can drastically reduce the amount of data MongoDB needs to retrieve documents for a query.

When a filter such as `find()`, `findOne()`, or in an aggregation pipeline, `$match` are used, MongoDB automatically utilizes the available indexes to speed up the query, making it more efficient.

When projection is used to limit the fields to be returned, and if it happens that all the returned fields are part of the index, MongoDB would not need to look back into the documents. In this case, the query speed can be significantly faster.

MongoDB's B-Tree

MongoDB primarily uses B-Tree data structures for its indexes. A B-Tree is a self-balancing tree data structure that maintains sorted data in a way that allows searches, sequential access, insertions, and deletions in logarithmic time. The “B” in B-Tree stands for “balanced” and indicates that the tree is designed to keep the data balanced, ensuring that operations are efficient as the dataset grows. The B-Tree structure allows MongoDB to perform efficient query. Instead of scanning every document in a collection, MongoDB can use the B-Tree index to quickly navigate through a small subset of the data to find the documents that match the query criteria.

However, index by itself is also an argument data structure and it consumes disk and memory space. Each time there is a write operation, the index needs to be updated. Therefore, a very complicated index schema slows down data insertion. It is critical to design appropriate index to optimize the overall performance of the database.

The default index for a collection is “`_id`”, a compulsory and unique field required by MongoDB. The user can also define index of different types. User-defined index can be unique or non-unique. The index of a collection can be created when the collection is empty, or when there are already some documents.

A MongoDB collection can have multiple indexes, and when you perform a query, MongoDB will choose the most suitable index to use based on the query’s criteria.

17.6.3 Simple and Compound Index

There are many options when creating the user-defined index. The following function

```
db.collection.createIndex(<keys>, <options>, <commitQuorum>)
```

creates a user-defined index. A single or a compound of multiple fields in a collection can be used as the index using

```
db.collection.createIndex(  
  {  
    <field1>: 1,  
    <field2>: -1,  
    ...  
  })
```

Those fields assigned with “1” or “-1” is listed as the ascending and descending index. The choice of order determines how the index sorts the indexed field’s values, which can affect the performance of certain queries.

The data types of the selected field(s) can be scalar, string, or even a nested

object. MongoDB has an internal mechanism that can convert different data types into a format that can be indexed and sorted efficiently.

The order of fields in a compound index can significantly impact query performance in MongoDB. Following the principle of “equality, sort, range” when defining compound indexes is a good practice to optimize queries.

- Equality. Fields that are used in equality conditions (=) should come first in the index. These fields are used to narrow down the search space quickly.
- Sort. Fields that are used for sorting the results should come next. Sorting fields in the index help MongoDB to efficiently retrieve documents in the desired order without additional sorting.
- Range. Fields that are used in range queries (<, <=, >, >=, \$in, etc.) should be placed last. Range queries can benefit from being at the end of the index because they can take advantage of the already reduced search space provided by the equality and sort fields.

It is possible to use the <options> argument to set additional constraints to the index, for example, to enforce its value to be unique.

17.6.4 Multi-key Index

When defining index on an array field, the index is known as multi-key index. Like the scalar fields, multi-key index can also be both single or compound. It is also possible to mix scalar keys with array keys.

Syntax wise, defining a multi-key index looks similar with defining a regular single or compound index as follows.

```
db.collection.createIndex(
  {
    <scalar field1>: 1,
    <scalar field2>: 1,
    <array field>: 1
  }
)
```

The only limitation is that there can be only one array key per index.

Behind the screen, MongoDB treats each element in the array key as a separate index value. Likewise, each element in the array of the index can be scalar, string, or nested object.

17.6.5 Hide and Remove Index

It is not recommended to modify the index of a collection frequently, as deleting and recreating index take time and resources. Sometimes we do need to remove a redundant index if it is adding too much writing cost.

It is possible to test the query performance if an index were to be removed

before actually removing it. This can be done by temporarily hide the index. MongoDB would not use hidden index in the query, though it will still update the index when new data is written in.

To hide an index, use

```
db.<collection>.hideIndex(<index>)
```

where `<index>` can be either a string or a document that indicates the status of the index. An example is given below.

Consider a restaurant collection, with the following user-defined index:

```
db.restaurants.createIndex( { borough: 1, ratings: 1 } );
```

Now the collection should have 2 indexes, the first the default index with `_id` field, the second the compound index composed of `borough` and `ratings` fields.

The user-defined compound index can be hidden by one of the two syntax below.

```
db.restaurants.hideIndex( "borough_1_ratings_1" ); // option 1
db.restaurants.hideIndex( { borough: 1, ratings: 1 } ); // option 2
```

After hiding the user-defined index, the index status becomes

```
[ {
    "v" : 2,
    "key" : {
        "_id" : 1
    },
    "name" : "_id_"
},
{
    "v" : 2,
    "key" : {
        "borough" : 1,
        "ratings" : 1
    },
    "name" : "borough_1_ratings_1",
    "hidden" : true
}]
```

To remove an index instead of hiding it, use `dropIndex()` instead of `hideIndex()`. A similar-looking command `dropIndexes()` allows removing multiple indexes at the same time.

17.6.6 Performance Evaluation with Index

To check the indexes of a system, use

```
db.collection_name.getIndexes()
```

The index can also be viewed from the graphical UI.

Consider using `explain()` function as follows when carrying out a find operation.

```
db.<collection>.explain().find()
```

which will list down the down-to-the-ground stages to carry out the query. Apply the same query to two identical collections, one with index and the other without on different scenarios where:

1. The filter document and the projection only contains indexed fields.
2. The filter document contains indexed fields, but the returns require more fields.
3. The filter document does not contain indexed fields.

and compare the differences.

17.7 Third-Party Connection

Both MongoDB shell CLI and MongoDB Compass GUI can connect to MongoDB as introduced in earlier sections. Third-party applications can also connect to MongoDB. As of this writing, MongoDB supports libraries for the following developing languages:

- | | | | |
|---------|-----------|---------|--------------|
| • C/C++ | • Kotlin | • Ruby | • TypeScript |
| • C# | • Node.js | • Rust | |
| • Go | • PHP | • Scala | |
| • Java | • Python | • Swift | |

A full list can be found at

```
https://www.mongodb.com/docs/drivers/
```

Here is a quick example to connect to MongoDB using Python library PyMongo. The installation of the library is not given.

```
import pymongo
from pymongo import MongoClient

def initialize_mongodb_collection(connection_string, database_name,
    collection_name, index_field):
    client = MongoClient(connection_string)
    db = client[database_name]
    collection = db[collection_name]
```

```
collection.create_index(index_field)

def push_to_mongodb(connection_string, database_name, collection_name,
                     data):
    client = MongoClient(connection_string)
    try:
        db = client[database_name]
        collection = db[collection_name]
        if isinstance(data, dict):
            result = collection.insert_one(data)
            print(result.acknowledged)
        elif isinstance(data, list):
            result = collection.insert_many(data)
            print(result.acknowledged)
        else:
            raise Exception("Data should be a dictionary or a
                            list of dictionaries")
        client.close()
    except Exception as e:
        raise Exception("Unable to insert the document: ", e)

def pull_from_mongodb(connection_string, database_name, collection_name
                      , query_dict):
    results = None
    client = MongoClient(connection_string)
    try:
        db = client[database_name]
        collection = db[collection_name]
        results = collection.find(query_dict) # return iterative
                                                object
        client.close()
    except Exception as e:
        raise Exception("Unable to insert the document: ", e)
    return results
```

A full instruction to execute the above example can be found at

<https://www.mongodb.com/docs/languages/python/pymongo-driver/current/get-started/>

17.8 Aggregation Framework

Aggregation is the analysis and summary of data. MongoDB, like many other databases, provides powerful tools to perform aggregation functions such as counting the number of documents, calculating the sum / average of a particular field of filtered documents, etc. Beyond that, MongoDB allows the user

to define a pipeline composed of a sequence of data processing procedures for aggregation. This enables autonomous data processing and analysis, and it is useful especially for big data applications. This feature, known as the data aggregation framework or data aggregation pipeline, is one of the most powerful and useful features that MongoDB offers.

Aggregation pipeline is supported in all MongoDB tiers, including MongoDB Community, Enterprise and Atlas.

17.8.1 Aggregation Pipeline Syntax

To apply different aggregation functions, use `aggregate()` as follows.

```
db.<collection>.aggregate( <pipeline>, <options> )
```

where `<pipeline>` is usually an array that contains a sequence of aggregation operations including filter, sorting, grouping and transforming, which may look like the following

```
db.<collection>.aggregate([
    {
        $<stage>: {<expression>}
    },
    {
        $<stage>: {<expression>}
    },
    {
        $<stage>: {<expression>}
    }
])
```

An example is given blow.

```
db.orders.insertMany( [
    { _id: 0, name: "Pepperoni", size: "small", price: 19,
        quantity: 10, date: ISODate( "2021-03-13T08:14:30Z" ) },
    { _id: 1, name: "Pepperoni", size: "medium", price: 20,
        quantity: 20, date : ISODate( "2021-03-13T09:13:24Z" )
            },
    { _id: 2, name: "Pepperoni", size: "large", price: 21,
        quantity: 30, date : ISODate( "2021-03-17T09:22:12Z" )
            },
    { _id: 3, name: "Cheese", size: "small", price: 12,
        quantity: 15, date : ISODate( "2021-03-13T11:21:39.736Z"
            ) },
    { _id: 4, name: "Cheese", size: "medium", price: 13,
        quantity:50, date : ISODate( "2022-01-12T21:23:13.331Z"
            ) },
    { _id: 5, name: "Cheese", size: "large", price: 14,
        quantity: 10, date : ISODate( "2022-01-12T05:08:13Z" )
            },
])
```

```

        { _id: 6, name: "Vegan", size: "small", price: 17,
          quantity: 10, date : ISODate( "2021-01-13T05:08:13Z" )
        },
        { _id: 7, name: "Vegan", size: "medium", price: 18,
          quantity: 10, date : ISODate( "2021-01-13T05:10:13Z" ) }
    ] )

db.orders.aggregate( [
    // Stage 1: Filter pizza order documents by pizza size
    {
        $match: { size: "medium" }
    },
    // Stage 2: Group remaining documents by pizza name and
    // calculate total quantity
    {
        $group: { _id: "$name", totalQuantity: { $sum: "
            $quantity" } }
    }
] )

```

where both `$match` and `$group` are commonly used stages.

And the return is a cursor pointing to the following array of documents

```

[
    { _id: 'Cheese', totalQuantity: 50 },
    { _id: 'Vegan', totalQuantity: 10 },
    { _id: 'Pepperoni', totalQuantity: 20 }
]

```

The return of an aggregation pipeline can be returned to the program for further data mining. It can also create new documents or modify existing documents.

Notice that the design of pipeline affects the performance of the query. There are many supported stages, each with a different list of input arguments. A full list of supported stages can be found from MongoDB's user manual at [6] under "Aggregation Stages". Calculations can be performed along with the aggregation pipeline. The calculations are triggered by aggregation operators. A full list of supported aggregation operators can be found from MongoDB's user manual at [7] under "Aggregation Operators".

It is highly recommended that one should read through MongoDB's user manual to learn the aggregation framework. For the convenience of the reader, commonly used stages are introduced as follows.

17.8.2 Stage: `$match`

The `$match` stage filters the documents to pass only the documents that match the specified conditions to the next pipeline stage. A general syntax is given below.

```
{$match: <filter>}
```

The `$match` stage should be used in early steps to make the pipeline more efficient.

17.8.3 Stage: `$group`

The `$group` stage separates documents into groups according to a “group key”. The output is one document for each unique group key. Use `null` as the group key to group all documents into a single group. Consequent aggregations are then performed per group.

A general syntax is given below.

```
{
  $group:
  {
    _id: <expression>, // Group key
    <field1>: { <accumulator1> : <expression1> },
    ...
  }
}
```

where `<accumulator>` indicates what to calculate on the grouped information to put into the `field`. For example, the accumulator can be `$count`.

The following is an example. Let there be a collection “commodity”, inside which each document represents a product, and has the following 3 fields: “item” (item name), “price” (unit price) and “quantity” (the sold number). The target is to calculate the sold total price for each product.

```
{
  $group :
  {
    _id : "$item",
    totalSaleAmount: { $sum: { $multiply: [ "$price", " $quantity" ] } }
  }
}
```

Do notice that to refer to the field name in the documents, use `"$<fieldname>"`.

The return of the above may look like the following

```
[
  { _id: "apple", totalSaleAmount: 50 },
  { _id: "banana", totalSaleAmount: 45 }
]
```

If `null` were used as the group key, the return would have been

```
[
  { _id: null, totalSaleAmount: 95 }
]
```

To count the number of documents in each group and put it as an additional field, consider using `$sum` as follows.

```
{
  $group: {
    _id: <expression>, // Group key
    count: { $sum: 1 } // Accumulator to count documents
  }
}
```

17.8.4 Stage: `$sort`

The `$sort` stage sorts the documents in either ascending order or descending order for further processing in the pipeline. The `$sort` stage is sometimes used before `$limit` stage.

A general syntax is given below.

```
{
  $sort:
  {
    <field1>: 1 // or -1
  }
}
```

where “1” and “−1” indicate the ascending and descending order with respect to the field.

17.8.5 Stage: `$limit`

The `$limit` stage selects and filters the first a few items from the pipeline. The input argument is simply a positive integer, as shown in the below example.

```
{
  $limit: 3
}
```

17.8.6 Stage: `$set`

The `$addFields` or `$set` stage (alias of each other) is used to modify or add fields in the pipeline, using the following syntax.

```
{
  $set:
  {
    <field1>: <new value>,
    <field2>: {<expression>},
    ...
  }
}
```

The assignment can be either a value or an expression, such as rounding, multiplying by a gain, etc. A list of supported arithmetic aggregation functions is given in Table 17.4.

17.8.7 Stage: \$count

The `&count` stage counts the number of the documents passing through the pipeline, and put the number into a field.

```
{
    $count: <field>
}
```

An example is given below

```
db.scores.aggregate([
    {
        $match: {
            score: {
                $gt: 80
            }
        }
    },
    {
        $count: "passing_scores"
    }
])
```

which count the number of students whose score is greater than 80. Notice that when using `$group` stage, `count: {$sum: 1}` can be used to count the number of elements in each group. Though both methods counts the number of documents, they are used in different contexts.

17.8.8 Stage: \$project

The `$project` stage changes the shape of the data in the pipeline. It selects which fields to be displayed. In many occasions it is used near the end of the pipeline, right before the data comes out of the pipeline.

A general syntax looks like the following.

```
{
    $project:
    {
        <field1>: 1,
        <field2>: 0,
        <field3>: <new value>,
        ...
    }
}
```

TABLE 17.4

MongoDB arithmetic aggregation functions.

Name	Description
\$abs	Returns the absolute value of a number.
\$add	Adds numbers to return the sum, or adds numbers and a date to return a new date. If adding numbers and a date, treats the numbers as milliseconds. Accepts any number of argument expressions, but at most, one expression can resolve to a date.
\$ceil	Returns the smallest integer greater than or equal to the specified number.
\$divide	Returns the result of dividing the first number by the second. Accepts two argument expressions.
\$exp	Raises e to the specified exponent.
\$floor	Returns the largest integer less than or equal to the specified number.
\$ln	Calculates the natural log of a number.
\$log	Calculates the log of a number in the specified base.
\$log10	Calculates the log base 10 of a number.
\$mod	Returns the remainder of the first number divided by the second. Accepts two argument expressions.
\$multiply	Multiplies numbers to return the product. Accepts any number of argument expressions.
\$pow	Raises a number to the specified exponent.
\$sqrt	Calculates the square root.
\$subtract	Returns the result of subtracting the second value from the first. If the two values are numbers, return the difference. If the two values are dates, return the difference in milliseconds. If the two values are a date and a number in milliseconds, return the resulting date. Accepts two argument expressions. If the two values are a date and a number, specify the date argument first as it is not meaningful to subtract a date from a number.
\$trunc	Truncates a number to its integer.

where “1” and “–1” are used to either include or exclude a field. It can also create new fields or overwrite an existing field, in which case the value of the new fields needs to be specified.

17.8.9 Stage: \$out

The `&out` stage creates a new collection or overwrite an existing collection using the aggregation pipeline result. If `&out` is used, it should be the last stage.

```
{
    $out:
    {
        db: "<database name>",
        coll: "<collection name>"
    }
}
```

or

```
{
    $out: "<collection name>"
}
```

17.9 MongoDB Atlas

MongoDB Atlas is the MongoDB cloud-based serverless solution. It allows the user to deploy MongoDB on the cloud. The user has the freedom to choose the base cloud service provider including AWS, Azure and Google Cloud. Thanks to Altas, the user has the flexibility to seamlessly change cloud service providers and service tiers without downtime.

In addition to just a host of the database, Atlas provides varieties of tools that helps the developer to develop applications using the database, such as a centralized cloud-based dashboard, autonomous synchronization with edge devices, etc. Some other examples include Atlas data lake which is optimized for analytical queries. Big data can be stored in Atlas data lake, from where analytical information can be retrieved. Atlas federation allows seamlessly query, transform, and aggregate data from one or more MongoDB Atlas databases and cloud object storage offerings. Atlas chart provides rich tools for MongoDB data visualization. There are many more tools.

As of this writing, Atlas allows the user to choose from 3 different storage tiers:

- Shared: the database is stored on shared servers; the user can select the server provider from AWS, Azure and Google Cloud

- M0: free, 512MB
 - M2: 2GB, \$9 / month
 - M5: 5GB, \$25 / month
- Dedicated: the database is stored on dedicated servers with dedicated storage, RAM and multi-core CPUs; 10GB to 4TB; \$0.08 / hour - \$33.26 / hour
 - Serverless: the database is provided as a microservice, and it automatically scales up and down based on use; up to 1TB; the cost depends on the amount of stored data, the number of read and write operations, the computational cost of backups, etc.

17.9.1 MongoDB Atlas Dashboard

To use MongoDB Atlas, register an account with MongoDB Atlas.

Create a database cluster. Select the pricing tier for the database cluster, create a user with admin role and assign him a password, and configure the database cluster access gateway (i.e., a list of IP address that can access the database cluster). Upon creation of the database cluster, the user gains the access to MongoDB Atlas dashboard. The browser-based MongoDB Atlas dashboard can be used to view, add, edit and remove documents, collections and databases.

The newly created database cluster is empty and has no databases, collections or documents. For tutorial purpose, Altas provides a sample database cluster. One can import the data in the sample database cluster into the created empty database cluster.

MongoDB Atlas dashboard provides data explorer tool that allows the user to view and edit the database directly, without using CLI or code. One can also filter documents from a collection by filtering the fields of the documents. Of course, MongoDB Atlas also provides connection string so that the user can connect to it remotely using CLI, from MongoDB Compass, or from third-party applications.

The following screenshot in Fig. 17.2 gives MongoDB Atlas dashboard. The database cluster, user and gateway can be viewed and configured from the dashboard.

To view collections and query for documents, click “Browse Collections” in Fig. 17.2 which gives Fig. 17.3. This sample database cluster contains multiple databases, each database with several collections, as shown by Fig. 17.3.

17.9.2 Connection String

Database connection string allows a client to connect to a database server. MongoDB Atlas provides two connection string formats, namely the standard

The screenshot shows the MongoDB Atlas dashboard for Project 0. The left sidebar includes sections for Overview, Deployment (with Database selected), Services, Security, and more. The main content area displays a summary of Cluster0, including its version (2.0.8), region (AWS Singapore (ap-southeast-1)), cluster tier (MongoDB Sandbox (General)), type (Replica Set - 3 nodes), and status (Inactive). It also shows network metrics (In: 1.2 MB/s, Out: 426.2 B/s) and data size (141.3 MB / 512.0 MB (28%)).

FIGURE 17.2

A demonstration of MongoDB Atlas dashboard.

The screenshot shows the MongoDB Atlas dashboard for Project 0, focusing on Data Services. The left sidebar includes Overview, Deployment (with Database selected), Services, Security, and more. The main content area shows the Cluster0 interface with the Collections tab selected. It lists databases (sample_airbnb, sample_analytics) and collections (customers, transactions, etc.) under sample_analytics. A preview of the sample_analytics.customers collection is shown, displaying 500 documents.

FIGURE 17.3

A demonstration of MongoDB Atlas dashboard where the databases and collections under “Project 0”, “Cluster0” are browsed. Database “sample_analytics”, collection “customers” is selected. There are 500 documents under this collection.

format and the DNS seed list format. The connection string of MongoDB Atlas can be retrieved from the dashboard by clicking “Connect” in Fig. 17.2. It is possible to connect to MongoDB Atlas from:

- MongoDB Shell (MongoDB’s CLI)
- MongoDB Compass (MongoDB’s desktop client GUI)
- User applications

Simply follow the instructions to connect to Atlas from one of the above entries. Notice that as a prerequisite, MongoDB Atlas needs to have the gateway policy to allow such connectivity.

A connection string may look like the following

```
mongodb+srv://<username>:<password>@<server-dns>/?authSource=admin&  
replicaSet=myRepl
```

17.9.3 Atlas Search

Atlas search is a powerful full-text search feature provided by Atlas that can access the documents and search for information based on relevance. It is built on top of Apache Lucene, a high-performance, full-featured text search engine library, and is designed to offer advanced search capabilities beyond the basic query.

Notice that Atlas search is NOT database query based on `find()`, nor AI-encoder-decoder based semantic search. Unlike traditional MongoDB queries which are based on exact matches or range conditions, Atlas Search provides relevance-based search. This means that search results are ranked by how well they match the search criteria, similar to how web search engines work. From this point of view, Atlas search is more business-user friendly, while MongoDB database search is more for developers and applications.

Atlas search, as well as Apache Lucene, uses search index to collect and parse the data to improve search efficiency. Do NOT confuse the search index with the MongoDB collection index introduced in earlier sections, as they are fundamentally different.

The remaining of this section introduces the use of Atlas search. Notice that Atlas search is power and flexible. Only a scratch is covered here.

Create Search Index and Test Atlas Search

Don’t confuse Atlas search index or Apache Lucene index with MongoDB collection index. They function fundamentally differently. The purpose of the search index is to provide advanced full-text search capabilities, as so required by Apache Lucene.

MongoDB Atlas dashboard provides a GUI to create and manage search

indexes for collections, as shown by the demonstration in Fig. 17.4. All configurations such as index analyzer, search analyzer, etc., are left as default in this example. It will take some time when setting up the search index.

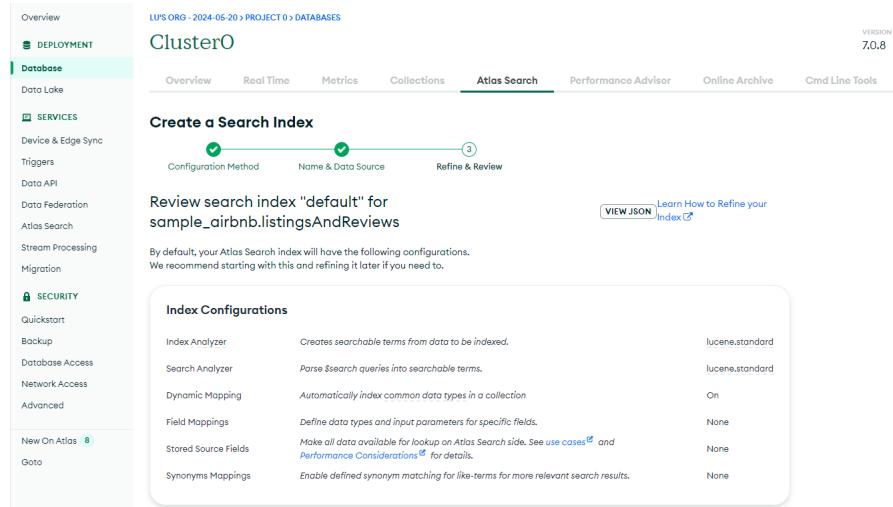


FIGURE 17.4

Atlas search set up search index using the dashboard.

With this setup, we can search the documents as if we were using a web searching engine. An example is given in Fig. 17.5. The most relevant documents with the keyword “big room” is returned. Each returned document corresponds with a score.

The default search index, as shown by the above example, uses dynamic mapping. It index all the fields in the collection. If we have priori knowledge about which field(s) to query, we can use static mapping which usually returns results faster.

To use static mapping, click the button “refine your index” during the search index creation, disable dynamic mapping, and add fields of interest following the instruction.

Use Atlas Search in the Aggregation Pipeline

With Atlas, we can add Atlas search as part in the aggregation pipeline. The associated stage is `$search`. It must be used together with Atlas search. The syntax looks like the following.

```
{
    $search: {
        "index": "<index-name>",
        "<operator-name>" | "<collector-name>": {
            <operator-spec> | <collector-spec>
        }
    }
}
```

The screenshot shows the MongoDB Atlas Search Tester interface. At the top, there are tabs for Overview, Real Time, Metrics, Collections, and **Atlas Search**. Below the tabs, there's a sidebar with a back arrow labeled 'default', 'Index Overview', and three buttons: 'Search Tester' (which is selected and highlighted in green), 'Query Analytics', and 'Quick Start Tutorials'. There's also a link 'View Atlas Search Docs'. The main area is titled 'Search Tester' and contains a search bar with the query 'big room'. Below the search bar, it says 'This search took 0.14 seconds'. Two search results are displayed in cards:

- SCORE: 17.411001205444336** _id: "235651"
 - listing_url: "https://www.airbnb.com/rooms/235651"
 - name: "LARGE ARTSY Room w/ Loft Bed 4 DOGGLERS!"
 - summary: "Large room primarily rented monthly with a 20% to 25% discount depending on the season."
- SCORE: 16.976694107055664** _id: "15771501"
 - listing_url: "https://www.airbnb.com/rooms/15771501"
 - name: "獨立房間 位於市中心 private room in the city"
 - summary: "House with 3 bedrooms and one big living room in Kowloon, 5 mins walk ..."

FIGURE 17.5
Atlas search test.

```
{
    "highlight": {
        <highlight-options>
    },
    "concurrent": true | false,
    "count": {
        <count-options>
    },
    "searchAfter" | "searchBefore": "<encoded-token>",
    "scoreDetails": true | false,
    "sort": {
        <fields-to-sort>: 1 | -1
    },
    "returnStoredSource": true | false,
    "tracking": {
        <tracking-option>
    }
}
```

which can be defined in the Atlas dashboard under “Collections”, “Aggregation”.

17.9.4 Schema Pattern

The schema pattern is the guidance of designing good MongoDB architecture. Schema anti-pattern, on the other hand, refers to the architecture design that causes sub optimal performance.

Commonly seen schema anti-patterns include:

- Massive arrays.
- Massive number of collections.
- Bloated documents.
- Unnecessary indexes.
- Queries without indexes.
- Data that is accessed together but not stored together.

Atlas provides tools to help identify these schema anti-patterns. There are different ways to tackle each of these issues.

18

Non-RDB Example: Redis

CONTENTS

18.1	Brief Introduction to Redis	235
18.2	Installation	236
18.3	Basic Operations	236
18.3.1	Key-Value Pair Operations	237
18.3.2	Array Operations	238
18.3.3	Set Operations	238
18.3.4	Hashes	239
18.4	Redis in Practice	239

Redis, short for *REmote DIctionary Server*, is an open-source in-memory distributed key-value database.

The community version of Redis is free of cost and can be installed on multiple platforms. Redis Cloud, on the other hand, is a paid service with additional features and can be deployed on various cloud platforms such as AWS, Azure, and Google Cloud.

18.1 Brief Introduction to Redis

Redis is an open-source in-memory distributed database. It is a NoSQL database and does not structure and store data in tables. Unlike MongoDB, which stores data using self-contained documents, Redis stores data in key-value pairs. Redis is renowned for its support of in-memory data storage, significantly speeding up data storage, query, and processing. As a result, it is often used as a lightweight data caching tool or a message broker.

The features of Redis are summarized as follows:

- In-memory storage: This speeds up reading and writing operations.
- Persistence: While primarily an in-memory database, Redis offers various ways to persist data on disk without significantly compromising performance.

- Complex data structures and associated atomic operations: Though Redis is a key-value store, it supports more complex data structures than just simple key-value pairs.
- High availability via replicas: Redis can create replicas to ensure high availability.
- Distributed storage via horizontal partitioning: Redis supports horizontal partitioning for distributed storage.
- Lightweight: Redis is designed to be lightweight, making it efficient and easy to use.

A potential drawback of Redis is that when it is used in-memory, the data may be lost in the event of a system shutdown. For this reason, a popular way of using Redis is to let it sit between the user and a persistent database running in the backend. In this architecture, Redis serves as a fast-responding replica to enhance the user experience. When the data to retrieve is in Redis, the system does not need to query the backend database and it can return the result in milliseconds, which otherwise would have cost hundreds or thousands of milliseconds.

18.2 Installation

To install Redis on a Linux machine, simply use the package manager of the machine. For example, for RHEL, do the following

```
$ sudo dnf install redis
```

to install Redis, and use

```
$ redis-server
```

to start the Redis server. Upon starting of Redis, the port number and the process id will pop up. By default, it runs on port 6379.

To login to Redis on the host machine, simply use

```
$ redis-cli
```

to open the CLI to the database. The CLI prompt that looks like

```
127.0.0.1:6379>
```

shall show up, from where the user can type in the commands. In the rest of this chapter, we will use > as the prompt indicator in the CLI.

18.3 Basic Operations

Basic operations such creating, querying and removing key-value pairs, arrays, sets, and hashes are introduced as follows.

18.3.1 Key-Value Pair Operations

Redis utilizes a key-value based data structure in the database. Therefore, the basic operations of Redis include creating, querying and deleting key-pair values. Notice that Redis CLI commands are not case sensitive in general, though in the examples below upper-case commands are used.

To set or overwrite a key-pair value, use

```
> SET <key> <value>
```

where notice that both the key and the value are to be interpreted as strings. If a value is composed of multiple words in its string, use quotation marks to wrap the content.

To retrieve the value of a key, use

```
> GET <key>  
<value>
```

Notice that the value is usually returned as a string datatype, even if it is a number. If the key does not exist, it would return (*nil*).

To check whether a key exists or not, use

```
> EXISTS <key>
```

and it will return either (*integer*) 1 or (*integer*) 0 to indicate whether the key exists or not respectively.

Finally, to remove a key, use

```
> DEL <key>
```

To retrieve all the keys in the database, use

```
> KEYS *
```

To remove all the keys, use

```
> FLUSHALL
```

It is possible to set TTL for a key, so that after a certain amount of time, the key-value pair would be removed automatically. To set TTL for a key, use

```
> EXPIRE <key> <seconds>
```

where *<key>* is an existing key. To check the TTL of a key, use

```
> TTL <key>
```

where notice that a TTL of `-1` means that the key lives indefinitely, and `-2` the key does not exist probably because its TTL has expired.

To set TTL of a key upon it is created, use

```
> setex <key> <seconds> <value>
```

18.3.2 Array Operations

Redis allows storing array as the value of a key, and it supports array-based operations.

To create or append an array, use

```
> lpush <key> <value last> ... <value first>
```

Note that when using `lpush`, the last item in the above input will be treated as the first element. This is because it pushes a new item to the left (beginning) of an array. To push new items from the right (end) of an array, use `rpush` instead as follows.

```
> rpush <key> <value first> ... <value last>
```

To retrieve items from an array, use

```
> lrange <key> <start index> <stop index>
```

which returns elements indexed from the start index to the stop index, both ends included. Notice that the first element in the array is indexed 0. When the stop index is set to `-1`, all items from the start index to the end of the array is to be returned. Hence, to return everything in an array, set the start and stop indexes to be 0 and `-1` respectively.

Finally, to remove an item from an array from its two ends, use

```
> LPOP <key>
<first value>
```

or

```
> RPOP
<last value>
```

which returns and at the same time removes the first or the last item in the array respectively.

18.3.3 Set Operations

It is not convenient to trace whether an item exists or not in an array introduced above. For that use case, consider using sets instead. A set is a collection of elements where each value is unique and has no orders.

To create and add items to a set, use

```
> SADD <key> <value> <value> ...
```

where <key> is the name of the set. Notice that elements in a set cannot duplicate, otherwise it will trigger an exception.

To check whether an item is in a set, use

```
> SISMEMBER <key> <value>
```

which returns either 0 (not exist) or 1 (exist).

To remove an item from a set, use

```
> SREM <key> <value> <value>
```

To retrieve all the elements from a set, use

```
> SMEMBERS <key>
```

18.3.4 Hashes

Hashes allows the user to store objects, i.e., a set of key-value pairs. Note that nested object is not supported.

To create, overwrite or add fields to a hash, use

```
> HSET <key> <field> <value> <field> <value> ...
```

To retrieve a hash or a particular field of a hash, use the following.

```
> HGET <key> <field>
```

returns the value of the selected field of a hash, and

```
> HGETALL <key>
```

returns all field-value pairs of a hash.

To check whether a field has been defined for a hash, use

```
> HEXISTS <key> <field>
```

which returns either 0 (not exist) or 1 (exist).

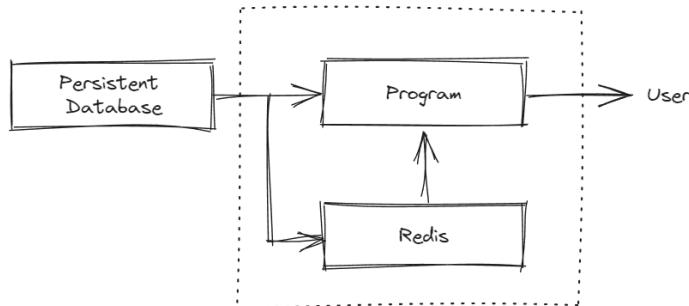
To delete fields from a hash, use

```
> HDEL <key> <field> <field> ...
```

Note that if all the fields of a hash are deleted, the hash itself will disappear.

18.4 Redis in Practice

A commonly seen architecture of implementing Redis is given in Fig. 18.1. The idea is that the program that needs to retrieve the information from the backend persistent database should search for the in-memory Redis database first. If the data is already stored in Redis and has not expired its TTL, the program will not query data from the backend database, but to directly

**FIGURE 18.1**

A commonly seen architecture of implementing Redis.

retrieve data from Redis. Redis plays as a replica of the database to speed up reading.

Other than replica of database for reading, the following gives a list use cases where Redis may be helpful:

- Data caching
- Message brokering
- Distributed locking
- Real-time data streaming, processing and analyzing

In addition to Redis CLI which a user can use to interact with Redis from the console, Redis also supports the following main languages. The program in these languages can connect to Redis via the Redis client.

- Python
- C# / .NET
- Node.js
- Java
- Go

More information is given at [8].

19

Virtualization and Containerization

CONTENTS

19.1	Introduction	241
19.2	Virtualization and Containerization	244
19.2.1	Virtualization	245
19.2.2	Containerization	247
19.3	Docker Container Basics	248
19.3.1	Docker Engine VS Alternatives	248
19.3.2	Docker Installation	249
19.3.3	Docker Container Management	250
19.3.4	An Example: Deploy a Containerized Application	256
19.4	Docker Volume and Bind Mount	257
19.4.1	Volume	258
19.4.2	Bind Mount	259
19.4.3	Multiple Volumes	259
19.5	Container Communication	260
19.5.1	Communication with Host Machine	260
19.5.2	Communication with Other Containers	261
19.6	Docker Image	261
19.6.1	Dockerfile Programming	261
19.6.2	Docker Image Management	268
19.6.3	Docker Image Sharing with Docker Hub	268
19.7	Multi-Container Orchestration	269
19.7.1	Protainer	270

Virtualization and containerization are widely appreciated technologies for distributing multiple instances of applications on single or multiple physical servers. The primary objective of these technologies is to enhance resource utilization efficiency while ensuring isolation between applications.

19.1 Introduction

One of the major differences between a server and a PC is that the former is usually shared among multiple users or applications at the same time. Though working on the same machine, a user would usually want a private working environment not interrupted by other users. In other words, a user would want to “virtually” work on an independent machine with his own CPU, RAM, I/O, OS, drivers and storage, despite that the actual hardware is shared with others. This can be achieved through *virtualization*, which enables running multiple operating systems on a single physical server in an uninterrupted and logically separated manner. The virtually independent computer of such kind is often called a *virtual machine* (VM).

Deploying a new VM generally consumes a considerably large amount of time and resources. This is because different VMs on the same server are separated at the OS level, with each VM requiring its own OS installation. Consider a scenario where there are hundreds of small applications (microservices), each requiring a similar but separate environment. Launching VMs for each and every of them is resource-intensive and can become an unnecessary waste of resource when these applications could have shared the same OS kernel and operated within their own isolated workspace.

In the above scenario, a more efficient approach is to deploy a single VM and place each application in a “container” with its own customized drivers and configurations. A container is similar to a VM in the sense that it provides a degree of isolation from others, but it is typically “lighter” than a VM because it doesn’t need to virtualize or duplicate the whole OS as VMs do. This makes containers cheaper to launch and manage.

The technology used to deploy and manage containers is known as *containerization*. A container contains all the configuration and requirement information of an application. Running a container on different platforms would consistently generate the same expected result. This has made the sharing and rapid deployment of containers remarkably easy and convenient. The similarities and differences of personal PCs, VMs, and container applications are summarized in Fig. 19.1.

As an analogy, think of running an APP as cooking a dish. The hardware corresponds with the physical resources in the kitchen such as the cooktop and gas. The OS corresponds with the cook. The OS requires drivers and libraries to run the APP correctly. The drivers and libraries correspond with the specific skills or cookers for the dish. Finally, the APP is corresponding with the expected dish.

In the most simple configuration, a dedicated machine is used to run an APP. This is like constructing a dedicated kitchen and hiring a dedicated cook for each dish. The cook is trained to master all necessary skills required for that specific dish. This is shown in Fig. 19.2.

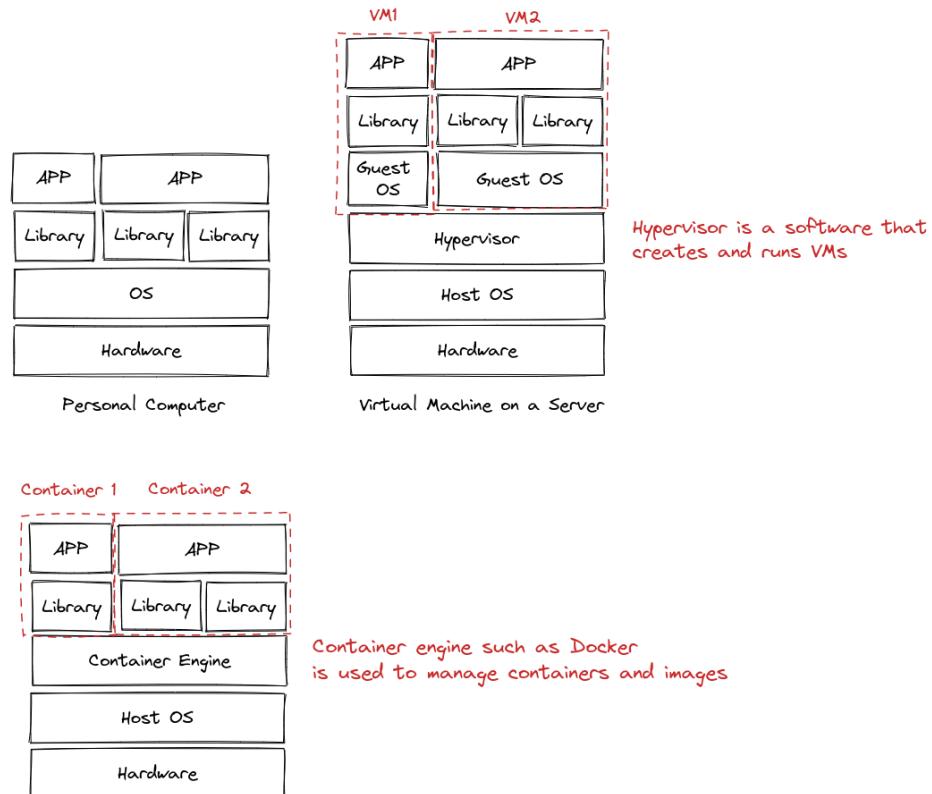


FIGURE 19.1
System architectures of PC, VM and container.

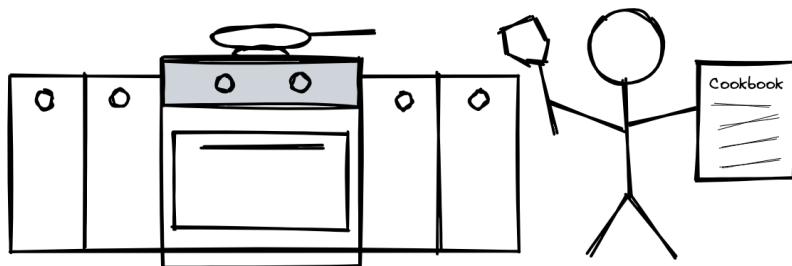


FIGURE 19.2
PC implementation: a cook in a kitchen.

In a VM implementation, a large and capable kitchen is setup in advance as shown in Fig. 19.3. For each dish, a cook is hired. Each cook is trained with the skills necessary for his assigned dish. All cooks share the same kitchen. This implementation is more efficient than Fig. 19.2, as there is no need to scale up the kitchen for a new dish. By sharing the resources among the cooks, the kitchen can be utilized more efficiently.

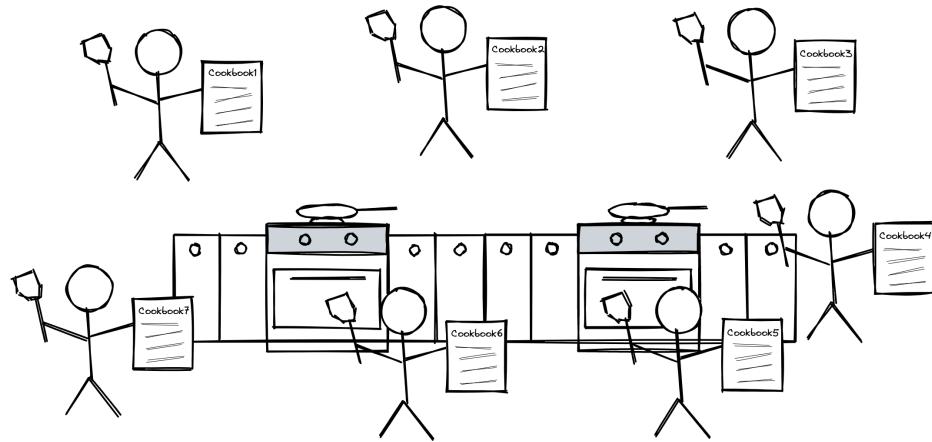
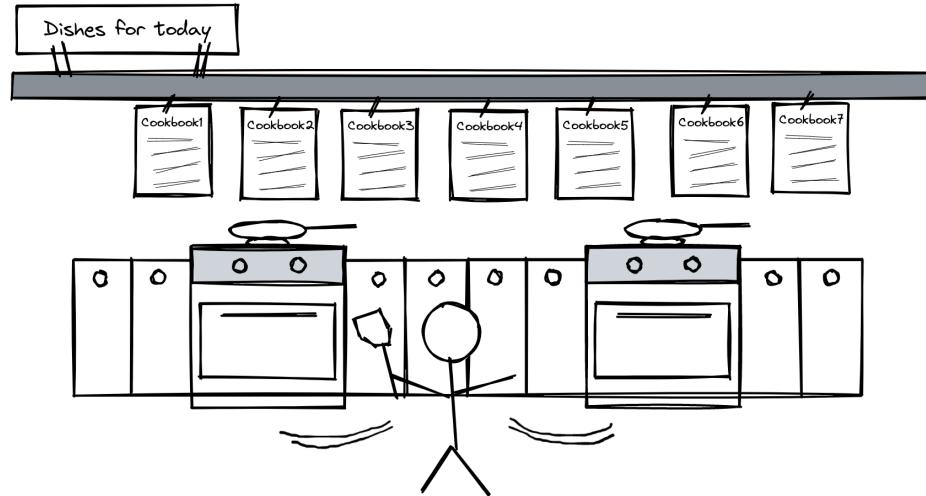


FIGURE 19.3

VM implementation: many cooks in a kitchen, each with a different cookbook.

While Fig. 19.3 might be a popular practice in many restaurants, it is still too costly to hire a new cook for each dish. In a containerization implementation, a cook usually handles a category of dishes, as shown in Fig. 19.4. Of course, each dish will stay in its own fry-pan in an isolated way. For each dish, its recipe is provided that gives all information required to prepare the dish consistently. As long as the cook is good at multi-tasking and has the basic skill sets for general dishes, Fig. 19.4 is usually a more efficient implementation than Figs. 19.2 and 19.3.

Just like a recipe guaranteeing the consistency of dishes, in containerization, an “image” guarantees the consistent behavior of container instances. An image is basically a collection of prerequisites and configurations to start a container efficiently and consistently. Images can be shared among machines to replicate the containers even if the machines adopt different underlying infrastructures such as hardware and OS.

**FIGURE 19.4**

Container implementation: one cook in a kitchen, handling multiple dishes, each has a cookbook and stays in its own pan.

19.2 Virtualization and Containerization

Virtualization and containerization have been studied for decades. This section gives a brief introduction to both technologies.

19.2.1 Virtualization

In a conventional data center, multiple physical servers are deployed, and each server has a single associated application. Servers may share the same LAN and network-attached storage (NAS) for data exchanging. A big issue of this implementation is the utilization efficiency of the servers and the unevenly distributed loads: some of the servers may not be utilized efficiently, while others may be overwhelmed. To deploy a new application, a new server must be purchased, which can take months of time. With more and more servers, the management and IT cost may grow exponentially.

To solve this problem, we need a systematical and automated way of integrating the resources in a data center and re-distributed them to the applications in an efficient manner. Virtualization is one of the most important technology used in this framework. Virtualization is essentially about running a system on a “virtualized machine”, where by saying “virtualized” we mean that the machine is not a physical machine, but a virtual execution environment that is emulated and managed by a special software running on the

actual machine known as the “hypervisor”. The setup should be transparent to the applications in the sense that the applications perform the same way as if it were running on a dedicated physical machine.

Depending on the items to be virtualized, virtualization can be divided into the following categories.

- Virtualization of equipment, mainly network resources and storages. This results in virtual local area network (VLAN), virtual private network (VPN), NAS and storage area network (SAN).
- Virtualization of operating systems. This results in the well-known VM and virtual desktop.
- Virtualization of application running environments. An example is Java virtual machine (JVM) that allows Java to generate consistent result while running on different machines.

There are different virtualization architectures. Commonly seen components shared by different architectures often include

- Host OS. The OS that resides on the physical machine, on which virtualization tools run.
- Virtual Machine Monitor (VMM), also known as the hypervisor. This is a software that runs on the host OS, managing all the guest OSs, providing them interface to the host OS and the hardware.
- VM, the system that runs in an virtualized isolated environment.

Host OS, VMM and VM relationship is shown in Fig. 19.2.

Different virtualization techniques are used in different types of VMMs. They can be widely divided into the following categories.

- Full virtualization. The VMM virtualizes everything including the hardware. The guest OS can run on the VM without modification or adaptation, just like running on any other machine.
- Para-virtualization. The VMM does not virtualize hardware. The guest OS is modified to certain extent to suit the VMM of this type.
- Hardware-assisted virtualization. The processor of the system is specifically designed to provide certain VM functions. These functions, after passing through VMM, are directed and executed on the hardware, thus enhancing system performance.

VMM is able to virtualize hardware resources such as CPU, memory and I/O. For the virtualization of CPU, the key is to virtualize privileged instructions (a set of instructions that can only be executed by software running in privileged mode, usually the OS) for the guest OS, so that the guest OS would

think it is directly talking to a CPU. This is challenging because guest OS usually does not possess the privilege due to the VM architecture. If the requests of the guest OS contain privileged instructions, the CPU will deny the request. When that happens, an exception will be raised to the VMM which will take care of the privileged instructions sequentially. Since hypervisor runs in privileged mode, it can execute those instructions.

For the virtualization of memory, VMM uses shadow page table to assign memory pages to different VMs. The maximum memory allocated to a VM can change dynamically.

For the virtualization of I/O, the development trend is that I/O operations will be less and less rely on software and OS, and more and more on hardware. The VMM directly maps the I/O hardware interface to the VMs.

Hypervisor VS Host OS, Who is the Boss?

Both the hypervisor and the host OS can run in privilege mode (also known as “Ring 0” in x86 architecture). However, notice that at one time there can be only one boss, i.e., there can be only one entity that runs in Ring 0 at the same time.

Depending on the type of the hypervisor configuration, this entity can be either the hypervisor or the host OS. In a type 1 hypervisor configuration, the hypervisor itself runs in Ring 0 and manages all hardware resources directly. In a type 2 hypervisor configuration, the host OS runs in Ring 0, and the hypervisor runs in a lower privileged ring.

19.2.2 Containerization

Containerization is an alternative virtualization approach to VM that can also virtualize independent workspaces for applications. Comparing with VM, containers are often lighter, hence more efficient for massive microservices deployment. Depending on the context, the term “container” may refer to one or more of the following 3 concepts: container runtime, container engine, and container orchestration.

Container runtime refers to the backend software that actually executes the containers. Examples of widely used container runtimes are “containerd”, “runc” and “cri-o”.

It is often inconvenient for users to talk to container runtimes directly. Container engine is the interface for a user or software to manage images and containers. Examples of container engines include “docker” and “podman”.

Finally, container orchestration is the software that strategically deploy, monitor, restart, and terminate the containers on servers. It can scale up and down the number of containers based on the demanding, and balance the API calls each container receives. Container orchestration is very useful in

production environment of large-scale services. One of the most widely used container orchestrations is “kubernetes”.

Notice that though container runtime is always a must-have, container engine and container orchestration are not.

19.3 Docker Container Basics

This section introduces docker. Notice that though being famous for docker container engine, docker as a company or community provides many revolutionary container related tools and services that go far beyond a container engine brand. Many tools and technologies of docker are used in container runtimes and container engines of other brands.

This section focuses mostly on the introduction of docker container engine.

19.3.1 Docker Engine VS Alternatives

Docker engine is the most popular container engine available on the market as of 2023, and it is free of charge for open-source, personal and small business usage. More details of docker can be found at <https://docs.docker.com/>.

But does that mean docker engine is the absolutely best and perfect container engine solution?

Docker has surely revolutionized how we use containerization technology in software development and deployment, and it has been one of the most popular and beloved container engine solutions. However, it is worth mentioning that docker engine is not the only available container engine. For example, as explained earlier podman is an alternative to docker. It supports the same interface as docker in its CLI (as if podman were an alias), and claims to have better performance and security. As a matter of fact, RHEL already started the transition from docker to podman from RHEL 8. Nowadays, installing docker on the latest versions of RHEL is possible but tedious. On the other hand, installation of podman on RHEL, if it had not been pre-installed, can be done simply by

```
$ sudo dnf install podman
```

Kubernetes, a famous container orchestration, is also dropping docker support according to their statement “docker support in the kubernetes is now deprecated and will be removed in a future release”. Some may even argue that “containers are alive, but the role that docker plays is shrinking”. Many open-source initiatives such as podman are gaining popularity.

Docker has some disadvantages indeed. For one thing, docker uses docker server (docker daemon), a single piece of software running in the backend of the system, to support all the services. This creates a single-point-of-failure in

the system. Docker requires root privileges, and it starts a container on behalf of the root user. This means that the program running inside the container and the users in the docker group can potentially bypass the OS access control and gain root access, which introduces security risk. These shortages are to some extent addressed by other container engines such as podman which is daemon-less and does not necessarily need to run on root user's behalf.

This is not to say that Docker is falling behind as a whole. Some key techniques that docker introduced are widely used in all different types and brands of container runtimes and engines. It is just that people do not like some of the features of docker engine, and alternative tools are being developed to fix these problems, the latter of which starting drawing more and more attention. Docker still enjoys widespread usage and support due to its massive community, wealth of online resources, and extensive compatibility with numerous tools and platforms. Nevertheless, for demonstration purpose, for the remaining sections docker is used throughout this notebook. Since podman provides the same interface, it is probable that podman can be used likewise to replicate all the results.

As of 2023, docker is still the dominating container market engine, with a market share of over 80%.

19.3.2 Docker Installation

To install docker on a Linux machine, go to <https://www.docker.com/> to look for the instruction. The installation steps differ depending on the host machine. As explained earlier, RHEL uses podman as its recommended container engine, and podman comes with RHEL installation.

In this section, consider installing docker engine on Ubuntu. Some of the key steps are summarized as follows. Remove existing docker engine, if any.

```
$ sudo apt-get remove docker docker-engine docker.io  
$ sudo apt-get remove containerd runc
```

Add docker's official GPG key and set up the repository.

```
$ sudo apt-get update  
$ sudo apt-get install ca-certificates curl gnupg lsb-release  
$ sudo mkdir -p /etc/apt/keyrings  
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --  
    dearmor -o /etc/apt/keyrings/docker.gpg  
$ echo \  
    "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/  
        docker.gpg] https://download.docker.com/linux/ubuntu \  
        $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.  
        list > /dev/null
```

Install docker.

```
$ sudo apt-get update
```

```
$ sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

where notice that `docker-compose` is a toolkit to manage containers using CLI. It is a handy tool in the testing environment.

To test whether docker is installed correctly, run

```
$ sudo docker run hello-world
```

and if everything is done correctly, a message started with “Hello from Docker!” will be displayed in the console, together with a brief introduction to how docker works.

Notice that to use docker commands, `sudo` privilege is required. To avoid typing `sudo` each time running a docker command, add the user to the docker group as follows. In the rest of the section, `sudo` is neglected for docker commands.

```
$ sudo usermod <user name> -aG docker
```

Docker installs at least two piece of software on the machine, namely docker CLI and docker server. The CLI is the interface to the user, and the server the daemon. Docker runs natively on Linux OS. If docker is installed on non-Linux system such as Windows or macOS, “docker desktop” is used which includes a Linux VM to host the docker daemon and run Linux-based containers.

19.3.3 Docker Container Management

Launch

To create and run a container from an image, simply use

```
$ docker run <configuration> <image>
```

Docker will search the local and remote repositories for the image, download the image if necessary, and start a container from that image. By default, `docker run` starts a container in attached mode (frontend mode) unless flag `-d` is used. After successful execution and completion of all the tasks, the container will enter “Exited” status.

For example, consider running a container of *alpine* using the command below. A screen shot is given in Fig. 19.5.

```
$ docker run -it --name test-alpine alpine
```

where `-i` stands for “interactive”, which keeps the container’s standard input (i.e., the console in this example) open so that the user can actively interact with the container. Option `-t` allocates a pseudo-TTY to the container. TTY stands for “TeleTYpewriter”, which enforces the I/O of the container to follow the typical terminal format and allows the user to interact with the container like a traditional terminal, hence making the interactive interface a bit more user-friendly. Flags `-it` are often used when running a container in attached

mode. Finally, **--name** assigns a name to the container. Without an assigned name, docker will assign a random name to the container.

```
sunlu@sunlu-laptop-ubuntu: $ docker run -it --name test-alpine alpine
/ # ls
bin dev etc home lib media mnt opt proc root run sbin srv sys tmp usr var
/ # pwd
/
/ # whoami
root
root # apk update
fetch https://dl-cdn.alpinelinux.org/alpine/v3.16/main/x86_64/APKINDEX.tar.gz
fetch https://dl-cdn.alpinelinux.org/alpine/v3.16/community/x86_64/APKINDEX.tar.gz
v3.16.0-302-g62bf0b8f5a [https://dl-cdn.alpinelinux.org/alpine/v3.16/main]
v3.16.0-304-g51632b3deb [https://dl-cdn.alpinelinux.org/alpine/v3.16/community]
OK: 17030 distinct packages available
/ # apk upgrade
(1/6) Upgrading alpine-baseLayout-data (3.2.0-r20 -> 3.2.0-r22)
(2/6) Upgrading busybox (1.35.0-r13 -> 1.35.0-r14)
Executing busybox-1.35.0-r14.post-upgrade
(3/6) Upgrading alpine-baseLayout (3.2.0-r20 -> 3.2.0-r22)
Executing alpine-baseLayout-3.2.0-r22.pre-upgrade
Executing alpine-baseLayout-3.2.0-r22.post-upgrade
(4/6) Upgrading libcrypto1.1 (1.1.1o-r0 -> 1.1.1q-r0)
(5/6) Upgrading libssl1.1 (1.1.1o-r0 -> 1.1.1q-r0)
(6/6) Upgrading ssl-client (1.35.0-r13 -> 1.35.0-r14)
Executing busybox-1.35.0-r14.trigger
OK: 6 MiB in 14 packages
/ #
```

FIGURE 19.5

An example of running *alpine* container, with interactive TTY and name *test-alpine*.

It can be seen from Fig. 19.5 that once the container is started, the user can interact with the container via shell and perform actions such as listing items in the current directory in the container. This is because the container is running in attached mode, and flags **-it** map the the containers input and output with the console.

While keeping the container running, open another terminal and use **docker container ls** to check the status of the container. More about **docker container ls** are introduced later. The container **test-alpine** shall appear in the list, as shown in Fig. 19.6. After exiting from Fig. 19.5 (by us-

CONTAINER ID							IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
03b1039d4f4d							alpine	/bin/sh	28 minutes ago	Up 28 minutes		test-alpine

FIGURE 19.6

List the running container *test-alpine*.

ing **exit** in *alpine*), the container will transfer its status from “running” to “exited”, as shown in Fig. 19.7.

CONTAINER ID							IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
03b1039d4f4d							alpine	/bin/sh	28 minutes ago	Up 28 minutes		test-alpine
03b1039d4f4d							alpine	/bin/sh	33 minutes ago	Exited (0) 7 seconds ago		test-alpine

FIGURE 19.7

List the exited container *test-alpine*.

Functional wise, `docker run` executes two commands behind the screen, namely `docker create` and `docker start`, where `docker create` creates a container and `docker start` starts the container by executing the startup script defined in the image. These two commands can be used separately. For example, to start an exited container, use

```
$ docker start <container>
```

Differences between `docker run` and `docker start`

Do note the fundamental differences between `docker run` and `docker start`. First of all, `docker run` starts a container from an image and it always creates a new container, whereas `docker start` starts an existing but exit container. Secondly, `docker run` runs a container in attached mode by default unless `-d` flag is used, while in contrast `docker start` runs a container in detached (backend) mode by default unless `-a` is used.

An example of using `docker run` with `-d` flag to start a container in detached mode is given below.

```
$ docker run -d --name test-background-alpine alpine
```

By changing `-it` to `-d`, the container runs in the backend silently. The status of the container, after executing the above command, will stay running and can be displayed by `docker container ls`.

Commonly used commands regarding launching a container are given in Table 19.1.

It is worth mentioning that file system snapshot and startup commands are defined inside an image. See Section 19.6 for more details. When a container starts, the file system snapshot is pasted to the container file system, and the startup commands executed. It is possible to overwrite the startup commands when starting a container, simply by amending the revised startup commands to the image name as follows.

```
$ docker run <image> <revised command>
```

where `<revised command>` must be defined in the image, or otherwise an error will be raised.

Interaction

For a container running in the backend, use `docker exec` to execute a shell command in that container as follows.

```
$ docker exec <container> <command>
```

To enable the TTY shell of a container running in the backend, use

```
$ docker exec -it <container> <command>
```

TABLE 19.1
Commonly used docker commands to launch a container.

Command	Flag	Description
<code>docker run</code>	—	Launch a container from an image in attached mode by default. If the image cannot be found locally, it downloads the image from the remote repository automatically.
<code>docker start</code>	—	Start an existing but exit container in detached mode by default.
<code>docker run</code>	-i	Keep the standard input of the container open when launching the container.
<code>docker run</code>	-d	Launch the container in the backend and keep it running.
<code>docker run</code>	--rm	Automatically remove the container when exiting. The removed container will not be listed in <code>docker container ls -a</code> . This is usually used with temporary containers during debugging.
<code>docker run</code>	-t	Allocate a pseudo-TTY. The flag usually comes with the flags -i or -d, to form -it or -dt.
<code>docker run</code>	--restart	Enforce restart of the container upon exiting. This is usually used on containers running in the backend. Commonly used restart configurations include --restart no (do not restart), --restart on-failure[:<max retries>] (restart if exits with an error flag), --restart always (always restart when exists).
<code>docker run</code>	--name	Assign a name to the container.

If <command> is replaced by the shell name used in the container, this would open the terminal of the container. Notice that the shell used by the application running inside the container may differ from the one used in the host machine. In the case of an *alpine* image based container, *ash* is the default shell. For a *ubuntu* image based container, *bash* is often used. To exit from the TTY shell while keep the container running in the backend, use shortcut key **Ctrl+p+q**. An alternative way to interact with containers running in the backend is to use

```
$ docker attach <container>
```

to attach local standard input, output, and error streams to a running container. Similar with the previously introduced `docker exec -it` command,

`docker attach` also starts the shell of the application running in the container. Use `Ctrl-C` to quite the shell.

Stop and Removal

To stop, kill or restart a container running in the backend, use

```
$ docker stop <container>
$ docker kill <container>
$ docker restart <container>
```

respectively. The difference between `docker stop` and `docker kill` concerns with how the OS manages process. When `docker stop` is used, a `SIGTERM` signal is sent to the main process that the container runs. The container still has a little bit of time (maximum 10 seconds) to terminate the job and clean up. When `docker kill` is used, `SIGKILL` signal is sent to the process, and the process is terminated immediately. When a container is stopped, it enters exited status.

To remove an exited container, use

```
$ docker (container) rm <container>
```

where `container` can be neglected. Alternatively, use

```
$ docker container prune
```

to remove all exited containers. Notice that there is a more powerful command

```
$ docker system prune
```

which removes not only exited containers but also unused network configurations, dangling images and cache.

Rename

To rename a container (without changing its container ID or anything else), use

```
$ docker rename <container-old-name> <container-new-name>
```

Monitoring

Use

```
$ docker container ls
```

to check the list of running containers, and

```
$ docker container ls -a
```

the list of all containers, running or exited. Alternatively, `docker ps`, `docker ps -a` can also be used to list down containers just like `docker container ls`, `docker container ls -a`.

To check the processes that is running in the container, use

```
$ docker top <container>
```

To quickly check container status including resource consumption (CPU, memory usage, etc.), use

```
$ docker stats [<container>]
```

where the user can choose to list down all containers or a specified container. To show more detailed information of a container, including its status, gateway, IP address, etc., use

```
$ docker inspect <container>
```

Finally, to check the logs of a container such as its standard output or error message, use

```
$ docker logs <container>
```

File Exchange with Host Machine

There are multiple ways and protocols to access the files in a container, depending on the I/O setup of the container. For a container running locally, `docker cp` can be used for file transfer between the container and the host machine as follows. From container to host machine:

```
$ docker cp <container>:<source> <destination>
```

and from host machine to container:

```
$ docker cp <source> <container>:<destination>
```

where `<source>` and `<destination>` refer to the path to the source and destination, respectively, located in the host machine or the container.

Commit

A container is usually generated from an image. It is also possible to do vice versa, i.e., packaging a container into an image. Notice that this is generally not recommended. Images shall be created mostly from Dockerfile, as will be introduced in Section 19.6.

To create an image from a container, use

```
$ docker commit <container> <image>
```

or

```
$ docker commit -c 'CMD ["<startup command>"]' <container> <image>
```

where `docker commit` command saves the container's file changes or settings into a new image, which allows easier populating containers or debugging in a later stage. Notice that `docker commit` does not save everything of the container into the image, and it is not the only way an image is created.

Ports Publishing

By default, containers do not expose any ports to the outside world. A container can be accessed only from its host machine using APIs provided by docker, such as `docker cp` to transfer files and `docker exec` to execute commands, but not network protocols. The user can allow public access from the internet by publishing the ports of the container using

```
$ docker run -p <host machine port>:<container port> <image>
```

when starting a container. To publish multiple ports, use multiple `-p` flags in a command.

More about containerized application connectivity to the internet, to the host machine and to other containerized applications are introduced in later sections.

19.3.4 An Example: Deploy a Containerized Application

An example of setting up a web server in containers from scratch is given in this section. For simplicity, everything happens on a single physical server. Only one container is used and the load balancer and the shared services are not included in the example.

As a first step, create a container from the official *nginx* image as follows. Notice that it is also possible to create a container from *apline*, and install *nginx* on *apline*.

```
$ docker run -dt --name simple-web nginx
```

Next, create the configuration file for *nginx*, and also the *html* files to be used as the static web page. For convenience, the files are created and edited in the host machine, then copied to the container. The following *default.conf* and *index.html* have been created, respectively. The configuration file *default.conf* is given below.

```
server {
    listen 80 default_server;
    listen [::]:80 default_server;
    root /var/www/html/;
}
```

The *html* file *index.html* is given below.

```
<html>
  <body>
    <h1>Hello World!</h1>
  </body>
</html>
```

Use `docker copy` to copy the two files to the designed locations in the container as follows. Notice that for read-only configuration files, unlike what we are doing now, a good practice is to build them into the read-only image layers. Keep in mind that this serves only as a demonstrative example. More about images are given in later sections.

```
$ docker exec simple-web mkdir -p /var/www/html  
$ docker cp default.conf simple-web:/etc/nginx/conf.d/default.conf  
$ docker cp index.html simple-web:/var/www/html/index.html
```

where `mkdir -p` creates the directories along the given path, if not exist. Notice that the file name in the destination can be ignored if it is the same with the source, i.e., the copy commands can be replaced by

```
$ docker cp default.conf simple-web:/etc/nginx/conf.d/  
$ docker cp index.html simple-web:/var/www/html/
```

Change the ownership of the `html` file as follows, so that the current user `nginx` is able to access that file.

```
$ docker exec simple-web chown -R nginx:nginx /var/www/html
```

Finally, reload and configuration file and restart the web server as follows.

```
$ docker exec simple-web nginx -s reload
```

To test the web server running inside the container, obtain the IP address of the container using

```
$ docker inspect simple-web | grep IPAddress
```

and open a browser to key in the obtained IP address. If everything is done correctly, the browser should try to access port 80 of the container, and the “Hello World!” web page shall show up.

For easy sharing and populating of the container, commit the container into a new image using `docker commit` as follows. The new image can be used to populate the web server, just like “`web01`” container given below.

```
$ docker commit simple-web simple-web-image  
$ docker run -dt --name web01 -p 80:80 simple-web-image
```

where `-p <host machine port>:<container port>` is used to map ports. Notice that different from the previous container “`simple-web`”, the new container “`web01`” IP address port 80 is mapped with the port 80 of the host machine. Therefore, the web page hosted in “`web01`” can be accessed not only by the host machine, but also by other machines in the same network with the host machine.

19.4 Docker Volume and Bind Mount

When a container is running, files inside it can be accessed or copied bidirectionally using the `docker cp` command. However, all data created or modified during the container’s life cycle resides in the container’s writable layer. More about the concept of layers are introduced in later section. If the container is removed, all data stored in its writable layer is also lost.

To ensure data persistence, it is best practice to use Docker volumes or bind mounts. Both methods link storage on the host machine to the container, allowing data to persist even if the container is removed.

- Docker volumes are fully managed by Docker. The user does not directly interact with the storage on the host machine. There are two types of docker volumes, the anonymous volume and the named volume.
- Bind mounts, on the other hand, allow the user to specify and manage the storage location on the host machine, providing direct control over the data.

In addition to data persistence, named docker volumes and bind mount can also play as a hub to shared data among multiple containers, which facilitates data exchange and allows containers to work on the same dataset.

19.4.1 Volume

In this section, we only consider named docker volume.

To create a docker volume, use

```
$ docker volume create <volume>
```

Notice that docker volume is fully managed by docker. The user cannot, and does not need to specify where the data should be stored in the host machine.

To list down volumes and to inspect a volume, use

```
$ docker volume ls
$ docker volume inspect <volume>
```

respectively. Notice that when using `docker volume inspect`, docker gives the mount point of the volume. However, the mount point is often a place in a virtual machine that docker creates, therefore, difficult to locate in the host machine. Like said earlier, the user should not access the data of a volume from the host machine directly.

Finally to remove specific an unused volume(s) or all unused volumes, use

```
$ docker volume rm <volume>
$ docker volume prune
```

respectively. When a volume is removed, all the data is lost.

When starting a container from an image, volumes can be mapped with the internal storage inside the container by using `-v` flag as follows

```
$ docker run -v <volume>:<container-path>[:ro] <image>
```

which should mount `<volume>` to `<container internal path>`. If `<volume>` is not created before hand, docker will create a docker volume with the name. The optional `:ro` can be used if it is a read-only volume, i.e., the container can only read from the volume but not write back to the volume. This can

become handy if the volume is shared by multiple containers and only one of them is allowed to writer to the volume while others being only the listener.

Notice that if not specifying the volume name when using flag `-v`, i.e.,

```
$ docker run -v :<container-path>[:ro] <image>
```

an anonymous volume will be created and used. Further more, if `--rm` is used, when the container is stopped, the container together with the anonymous volume will be removed. A use case of anonymous volume is given in the later section.

19.4.2 Bind Mount

Bind mount works similarly as docker volume, except that the user controls the location on the host machine where the data is persisted. Recall

```
$ docker run -v <volume>:<container-path>[:ro] <image>
```

which associate a named docker volume to the path in the container. Instead of `<volume>`, specify the path in the host machine as follows

```
$ docker run -v <host-machine-path>:<container-path>[:ro] <image>
```

in which case docker will persist and synchronize the data on the paths inside and outside the container. Quotation marks can be used "`<host-machine-path>`" if there are spaces or special characters in the host machine path. Notice that the specified host machine path should be accessible by docker. Likewise, the optional `:ro` can be used to prevent the container from overwriting the host machine.

Bind mount is often used in development stage where the developer wants to map application source code or data on his host machine into the container directly. When the application is ready for shipping, the consolidated source code and data should be built into the image. You cannot expect the end user to have a separate copy of the source code and data on his machine, and to use bind mount each time he starts the application.

19.4.3 Multiple Volumes

It is possible to run a container with multiple `-v` flags, each pointing to a different path in the container. Should there be any clashes, the rules with more specific path wins. For example, consider

```
-v <volume or path 1>:/app -v <volume or path 2>:/app/data
```

used together in a `docker run` command. In this example, `/app/data` will be mounted to `<volume or path 2>` and everything else in `/app` to `<volume or path 1>`.

Consider

```
-v <volume or path 1>:/app -v /app/data
```

which mount an anonymous volume to `/app/data` to protect it from being overwritten by any content in `<volume or path 1>`. This is a good use case of anonymous volume.

19.5 Container Communication

In practice, a containerized application needs to communicate with applications running outside the container to request services such as API calls. Commonly seen use cases are as follows.

- Containerized application requests services from the Internet.
- Containerized application requests services from the host machine, for example, from a database deployed on the host machine.
- Containerized application requests services from other containerized applications.

With proper port mapping using `-p`, the communication with the Internet is automatically enabled. Therefore, the focus of this section is mainly on data exchange and API calls between the container and the host machine or other containers.

19.5.1 Communication with Host Machine

The containerized application may request services from the host machine. In this case, the application code needs to be modified to enable communications with the host machine. An example is given below.

Consider a simple scenario where the containerized application needs to send a request to the host machine to trigger an API call of MongoDB. Notice that the MongoDB is installed on the host machine and not in the container.

If the application were running on the host machine instead of in a container, it can communicate with the MongoDB server using a connecting string that looks like the following

```
'mongodb://localhost:27017/<database name>'
```

and it should be part of the application code. However, it would not work if the same code is used in the containerized application. The code needs to be modified as follows.

```
'mongodb://host.docker.internal:27017/<database name>'
```

where `host.docker.internal` is a recognized domain by docker that refers to the host machine. Docker translates the domain to the IP address of the host machine.

Notice that `host.docker.internal` can be used to refer to the host machine not only with MongoDB requests, but also with other requests such as a general HTTP request.

19.5.2 Communication with Other Containers

As a basic solution, cross container communication can be done following the example given below. Notice that this is not a common practice in production environment as it is not robust and efficient.

Consider a simple scenario where there are two containers, one running the application and the other hosting a MongoDB.

The container with MongoDB deployment is started as follows.

```
$ docker run -d --name mongodb mongo
```

where `mongo` is the official docker image for MongoDB on Docker Hub.

The IP address of the MongoDB container can be found using

```
$ docker container inspect mongodb
```

where `mongodb` is the name of the container specified in the earlier command. In the result of the inspection, look for `IPAddress` under `NetworkSettings`.

Then in the application code, replace `localhost` in the connecting string

```
'mongodb://localhost:27017/<database name>'
```

with the IP address of the container.

Notice that each time a container is deployed, its network settings are done by docker and they differ from machine to machine. Therefore, this basic approach only works temporarily in development environment and should not be used in the production environment.

Docker allows the user to configure the container networks. This is a more preferred approach and it makes things easier than the basic approach introduced earlier.

19.6 Docker Image

Images are used to create containers. An image performs like a blueprint that encapsulates all the necessary information needed to spawn a container. It includes initial configurations, requisite libraries, and other pertinent metadata. Docker images are highly portable and can be shared across various machines and platforms. This section delves deeper into the construction and functionality of docker images.

19.6.1 Dockerfile Programming

An image shall contain everything needed to create and initialize a container. This includes but not limited to:

- Necessary steps to create a container
- Files to support the application
- Libraries, tools and dependencies

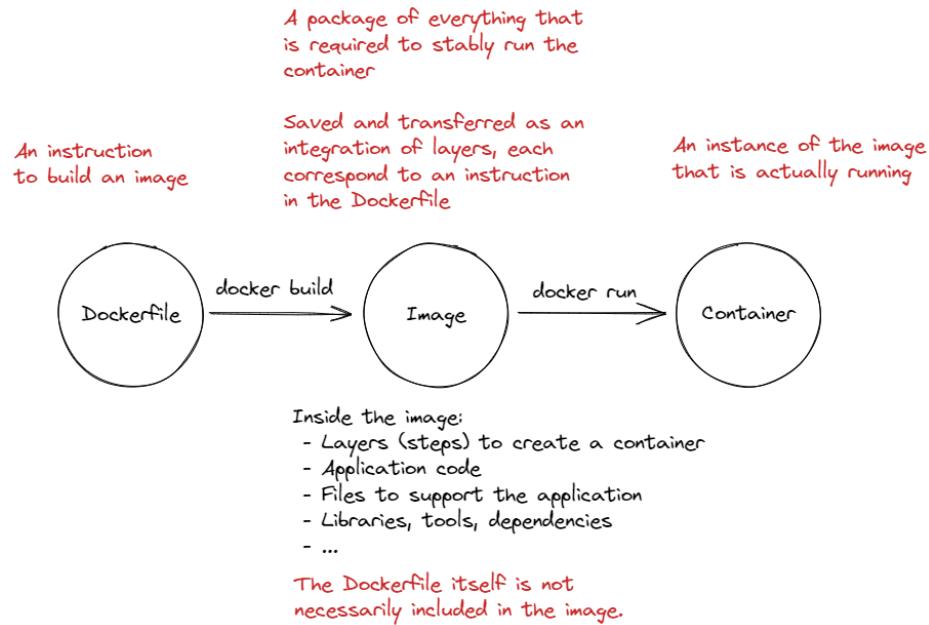
In addition, an image shall be designed and organized in such a way that it is portable, reusable and light, and can be used to easily populate large number of containers. For better inheritability, an image might be based on another existing image, which is called its parent image. An image with no parent, such as the official *hello-world* image from docker hub, is called a base image.

The most common and flexible way to create a docker image is to build from Dockerfile. The Dockerfile is a text document that serves as the blueprint for constructing a docker image. Generally speaking, a Dockerfile follows the following flow:

- (1) Specify base image
- (2) Add additional configurations
 - Setup file system
 - Install dependencies and other programs
 - ...
- (3) Specify startup command

A more detailed step-by-step guidance example is given later. The relationship between Dockerfile, image and container is illustrated in Fig. 19.8. Notably, the Dockerfile itself isn't included within the resulting image.

Since a docker image's primary role is to serve as a template for containers, many Dockerfile commands appear like a “step-by-step” recipe for container creation. Each instruction corresponds to an “image layer”. An image is, in essence, an amalgamation of these layers. It is stored and distributed in this format. If images share layers (for instance, different versions of the same app), these shared layers are not saved or transferred redundantly, hence significantly reducing image sizes. More details can be found at <https://docs.docker.com/storage/storagedriver/>.

**FIGURE 19.8**

A demonstration of how Dockerfile, image and container link to each other.

Docker Image Layers

Docker containers employ a special file system known as the Union File System (UFS), which is well suited to the “layer” concept. UFS facilitates file sharing between the container and the host machine, along with combining read-only upper layers and writable lower layers, among other functions.

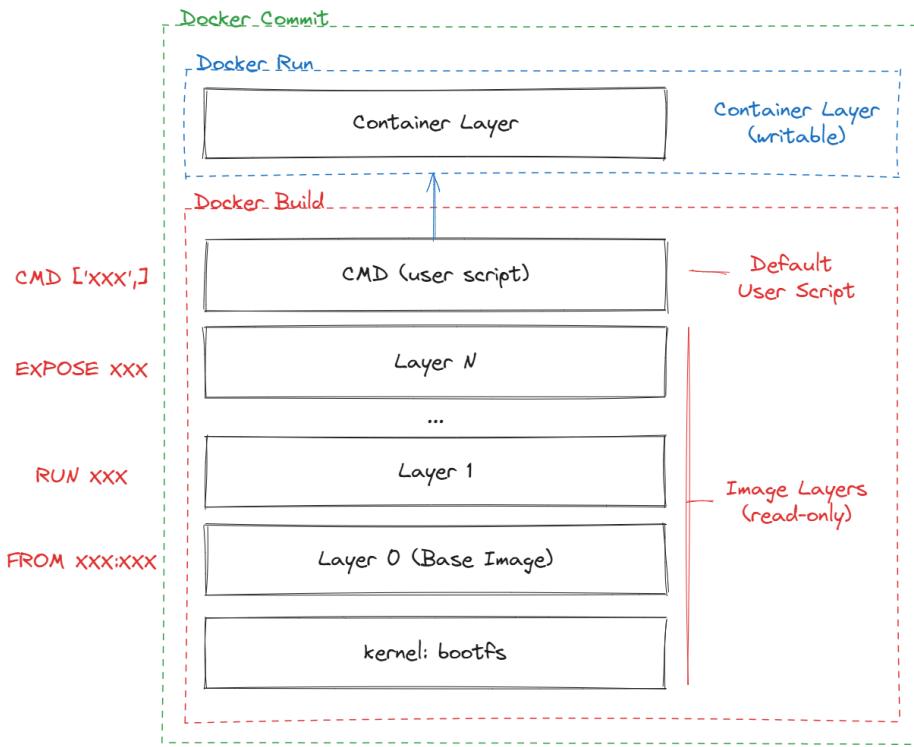
A demonstration to illustrate this concept is given in Fig. 19.9.

Just as a quick example, the Dockerfile to build the official *hello-world* image from docker hub looks like the following.

```
FROM scratch
COPY hello /
CMD ["/hello"]
```

In the example above, **FROM scratch** signifies that this image is a base image without a parent. The **COPY hello /** instruction copies the *hello* binary script from the image to the root directory of the container. Lastly, **CMD ["/hello"]** runs the *hello* binary script.

In general, a typical Dockerfile includes the following instructions to build an image. These instructions allow the image to know how to create a con-

**FIGURE 19.9**

A demonstration of docker image layer structure.

tainer, automatically construct the file system directory structure, install necessary packages, and run the app:

- (1) Define parent image.
- (2) Create filesystem directory.
- (3) Set working directory.
- (4) Copy files.
- (5) Configure registry.
- (6) Install packages.
- (7) Copy more files after package installation. (Create `.dockerignore` file and specifies the files you would rather not to be copied to the container, to prevent any overwriting to the earlier installed packages.)
- (8) Switch to the correct user.

- (9) Expose port.
- (10) Run the application.

The commonly used keywords to be used in a Dockerfile to realize the above instructions, such as `FROM`, `RUN`, are explained in Table 19.2.

It is worth highlighting the distinction between `RUN` and `CMD`. The `RUN` command is executed during the building of the image, which creates a virtual representation of the environment to run the service. Therefore, `RUN` is used to prepare the environment, for example, to install necessary libraries. On the other hand, command `CMD` simply specifies the default command to run when the container starts. It is built together with the image, but not considered as a read-only image layer. Indeed, the contents in `CMD` can even be overwritten if the user wishes to do so when `docker run` executes.

Besides Table 19.2, there are other Dockerfile keywords that can significantly simplify the design and maintenance of the image. For example, `ENV <key>=<value>` assign a value to an environmental variable which can later be referred as `$<key>` in the Dockerfile. An example is given below.

```
ENV PORT=80  
EXPOSE $PORT # this is interpreted as EXPOSE 80
```

Notice that environmental variables are accessible by the application code. They can be overwritten during `docker run`. This means that the user can potentially use environmental variable to dynamically change the behavior of the application in the container when using `docker run`. There are several benefits of doing this, for example, to enhance security. For example, if there is a password that the user needs to key in during the start of the container, the developer may want to put it as environmental variable and let the user to fill in, instead of hardcode the password in Dockerfile.

Another useful features of Dockerfile is the Dockerfile argument. It leaves a “space” in the Dockerfile that the developer can later fill in when he builds the image. An example is given below.

```
ARG DEFAULT_PORT [=80]  
ENV $DEFAULT_PORT  
EXPOSE $PORT
```

The default value `[=80]` is optional. When building the image, use `--build-arg <variable-name>=<value>` to override the argument value. Notice that unlike environmental variables, Dockerfile arguments are not accessible by the application code or by the `CMD` instruction.

Additionally, `LABEL <key>=<value>` assigns a tag to the image, which can be displayed when `docker inspect <container>` is used.

Dockerfile programming can be very flexible and complicated at the same time. The advanced materials goes beyond the scope of this notebook.

Examples of Dockerfiles are given below, one from docs.docker.com and the other from Linux Academy. The docker image layer structure of the second

TABLE 19.2

Critical keywords used in a Dockerfile.

Syntax	Description
<code>FROM <image>[:<tag>]</code>	Define the parent image. A Dockerfile must start with a <code>FROM</code> instruction. A Dockerfile can contain multiple <code>FROM</code> instructions for multi-stage build, in which case the last <code>FROM</code> statement is the base image of the outcome image and the earlier <code>FROM</code> instructions creates intermediate images that can be used by the outcome image. An optional <code>:<tag></code> following <code><image></code> can be used to specify the version of the image to use as the base. By default, the latest version of the image is used.
<code>RUN <command></code>	Execute a shell command using <code>/bin/sh -c</code> .
<code>WORKDIR <path></code>	Set the working directory from the point onward. This prepares the working directory for the upcoming <code>RUN</code> , <code>COPY</code> , etc., commands.
<code>ADD <src> <dest></code>	Add (Copy) <code><src></code> , either a directory/file or URL, to <code><dest></code> . An optional <code>[--chown=<user>:<group>]</code> can be used to specify the owner and group of the added files.
<code>COPY <src> <dest></code>	Copy <code><src></code> , a directory/file, to <code><dest></code> . An optional <code>[--chown=<user>:<group>]</code> can be used to specify the owner and group of the added files. Notice that <code>COPY</code> is similar with <code>ADD</code> . <code>COPY</code> is easier but less powerful than <code>ADD</code> . It cannot handle tar or URL.
<code>USER <user></code>	Switch user for the instructions beyond this point.
<code>EXPOSE <port></code>	Specifies the ports that the container shall listen to. An optional <code>/<protocol></code> following <code><port></code> can be used to specify the protocol for communication.
<code>VOLUME [<path>]</code>	Attaches an anonymous volume to the specified path inside the container.
<code>CMD [<exe>, "p1", ...]</code>	The user-script command. This is the last instruction in the Dockerfile that usually starts the APP. Notice that a Dockerfile can only contain one <code>CMD</code> instruction. The executable command name and the parameters are put into a list. This user-script can be overwritten by <code>docker run <image> <other commands></code> when running the container.

example is given in Fig. 19.9 as a demonstration. Notice that in Fig. 19.9, `bootfs` refers to the “boot file system”, including the bootloader and the Linux kernel. Upon creation of a container using `docker run`, a container layer will be added to the image, as shown by the blue dashed box in Fig. 19.9. In the container, all the changes made is saved into the container layer.

To generate a new image to include the changes made in the container, use `docker commit`, which essentially commits the container layer as the latest image layer in the new image, as shown by the green dashed box in Fig. 19.9.

```
# First Example
FROM golang:1.16
WORKDIR /go/src/github.com/alexellis/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

FROM alpine:latest
RUN apk --no-cache add ca-certificates
WORKDIR /root/
COPY --from=0 /go/src/github.com/alexellis/href-counter/app ./
CMD ["./app"]

# Second Example
FROM node:10-alpine
RUN mkdir -p /home/node/app/node_modules && chown -R node:node /home/
    node/app
WORKDIR /home/node/app
COPY package*.json .
RUN npm config set registry http://registry.npmjs.org/
RUN npm install
COPY --chown=node:node .
USER node
EXPOSE 8080
CMD ["node", "index.js"]
```

With the Dockerfile ready, use `docker build` to build an image. An example is given as follows.

```
$ docker build <path/url> -t <image name>
```

or

```
$ docker build <path/url> -t <user name>/<image name>:<tag>
```

where `<path/url>` is the path or URL to the directory where the Dockerfile locates (does not need to contain “/Dockerfile” in its end), and `-t` gives a tag, in this case an image name, to the image to build. For the convenience of sharing, it is recommended that all images to be pushed to a public registry (such as Docker Hub) shall be tagged with user name and version. Notice that the tag can be changed later after the image is built.

It is worth mentioning that `docker build` builds the images by layers

in the Dockerfile, and it will not re-build the same layer to which point no changes are made. This feature suggests that we should put the common and static layers in the early part of Dockerfile, and customized and user-defined layers in the late part, thus making the building (and also sharing, as will be introduced later) of the images more efficient.

19.6.2 Docker Image Management

The most commonly used image operations can be categorized as follows.

- Create an image.
- Create a container from an image.
- Upload and download an image from a remote server.
- Manage local images, such as listing down all images, deleting an image, etc.

The first two operations have been introduced in earlier sections. The third and last ones are introduced below. Use the following command to search for an image on the default remote repository server (Docker Hub).

```
$ docker search <image name>
```

Use the following command to download or update an image from the default remote repository server as follows. Notice that different from `docker run`, this command will not start a container from the image.

```
$ docker pull <image name>
```

Notice that since images are stored by layers, if two images share common layers, it is unnecessary to pull the shared layers repeatedly when downloading the second image, if the first image already exists in the host machine. Command `docker pull` is smart enough to automatically detect shared layers, and avoid duplicating download of layers.

Use the following commands to list down or remove images.

```
$ docker image ls  
$ docker image rm <image name>  
$ docker rmi <image name> # same as docker image rm  
$ docker image prune # remove all problematic images  
$ docker image prune -a # remove all unused images
```

where `prune` removes all problematic images, and `prune -a` removes all unused images from local.

Use the following command to inspect an image, and list down its metadata details.

```
$ docker image inspect <image name>
```

19.6.3 Docker Image Sharing with Docker Hub

There are two ways to share an image:

- Share the Dockerfile and APP source code so that the receiver can build docker image from his end.
- Share the image directly.

It is often more convenient to share the built image instead of the Dockerfile and the source code. However, a docker image is stored and managed by layers and cannot be shared simply by file transfer.

Docker hub is a commonly used server for storing and sharing docker images. It is also the default remote repository server of docker engine. However, do notice that docker hub is not the only remote docker image server. Some alternatives are Amazon Elastic Container Registry, Red hat Quay, Azure Container Registry, Google Container Registry, etc.

After registering an account on docker hub, use the following command to login to the docker hub from your local machine.

```
$ docker login --username=<user name>
Passowrd:
```

Assume that there is an image in the local machine, and an empty repository on docker hub. In order to push the local image to the docker hub, the first step is to add the remote repository and the “*RepoTags*” in the local image as follows.

```
$ docker tag <image name> <user name>/<repository name>:<version>
```

where *<version>* is a tag usually used to distinguish the different branches or versions of the images on docker hub. For the first image upload, it can simply be *latest*.

Use the following command to push the image to docker hub.

```
$ docker push <user name>/<repository name>
```

Notice that when using `docker run` on an image without a specified version and the image is not stored locally, Docker will automatically search and pull the latest version of that image from the remote repository, usually Docker Hub by default. However, if the image already exists locally, Docker simply uses that image and will not automatically check the remote repository for the latest version. To update the local image, manually use `docker pull` to pull the latest or specific version of the image from the remote repository.

19.7 Multi-Container Ochestration

As introduced earlier, docker engine can be used to build and share images as well as start, monitor, and stop containers. It can be difficult for a user

to manage containers manually when a lot of them are deployed. Container orchestrators such as Protainer and Kubernetes are helpful with managing containers. Many of these tools are able to automatically adjust the number of containers and balance their loads.

19.7.1 Protainer

Portainer is an open-source container management tool. It has a web-based dashboard user interface. Notice that Portainer itself also runs in a container.

Before starting a Portainer container, it is a good practice to first create a docker volume for Portainer to store the database. Use the following command to create such docker volume.

```
$ docker volume create portainer_data
```

Then run a Portainer container using

```
$ docker run -d -p 8000:8000 -p 9000:9000 -p 9443:9443 --name portainer  
--restart=always -v /var/run/docker.sock:/var/run/docker.sock -v  
portainer_data:/data portainer/portainer-ce
```

where ports 8000, 9000 and 9443 are used for hosting HTTP traffic in development environments, hosting web interface, and hosting HTTPS or SSL-secured services, respectively. The `docker.sock` is the socket that enables the docker server-side daemon to communicate with its command-line interface. The image name for Portainer community edition (distinguished from the business edition) is `portainer/portainer-ce`.

Use `https://localhost:9443` to login to the container. The following page in Fig. 19.10 should pop up in the first-time login, asking the user to create an administration user.

After creating the admin user and logging in, the status of images, containers and many more can be monitored via the dashboard, as shown in Figs. 19.11, 19.12 and 19.13. Notice that in Fig. 19.13, using the “quick action” buttons, the user can check the specifics of the container and interact with its console, just like using `docker container inspect` and `docker exec`.

In summary, Portainer is an easy-to-use container management tool with clean graphical interface that a user can quickly get used to without a steep learning curve.

▼ New Portainer installation

Please create the initial administrator user.

Username

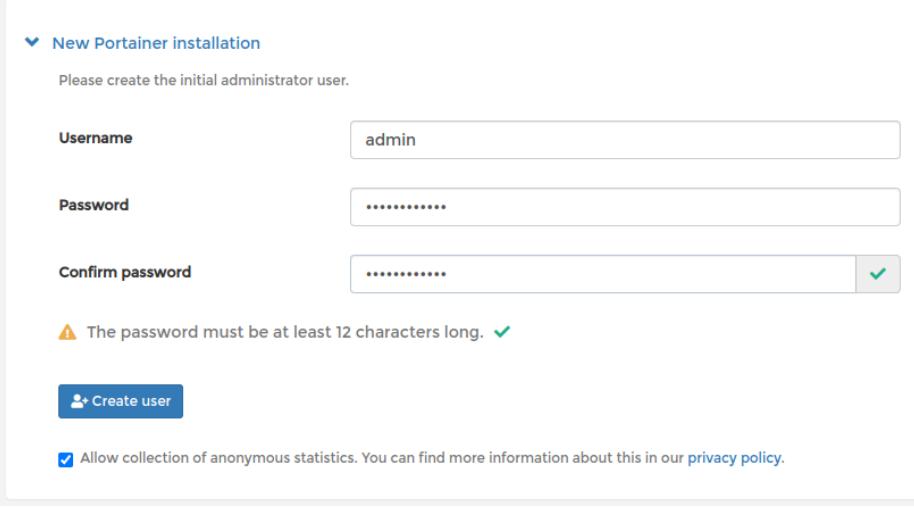
Password

Confirm password

⚠ The password must be at least 12 characters long. ✓

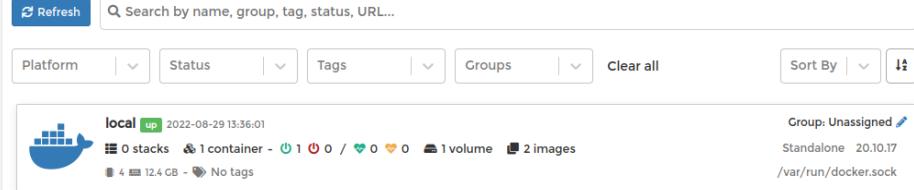
[Create user](#)

Allow collection of anonymous statistics. You can find more information about this in our [privacy policy](#).



This screenshot shows the 'New Portainer installation' setup page. It has fields for 'Username' (admin), 'Password', and 'Confirm password'. A warning message says the password must be at least 12 characters long. There's a 'Create user' button and a checkbox for anonymous statistics collection.

FIGURE 19.10
Portainer login page to create admin user.



This screenshot shows the Portainer dashboard. At the top, there are filters for Platform, Status, Tags, Groups, and Sort By. Below is a summary card for the 'local' host: 0 stacks, 1 container (status up, created 2022-08-29 13:36:01), 0 images, 1 volume, 12.4 GB storage, and 0 tags. To the right, it shows the Docker version (20.10.17) and socket path (/var/run/docker.sock). The 'Group: Unassigned' link is underlined.

FIGURE 19.11
Portainer dashboard overview of docker servers.

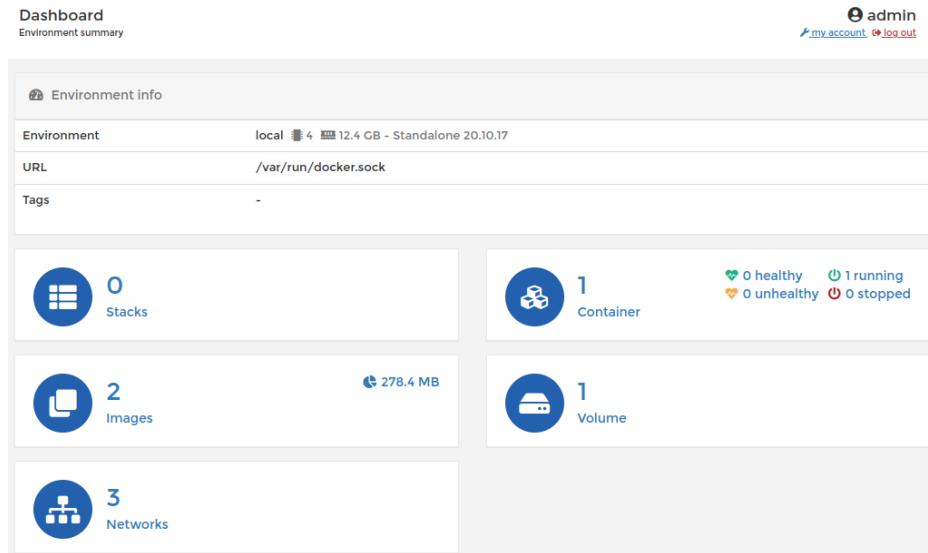


FIGURE 19.12
Portainer dashboard overview in a docker server.

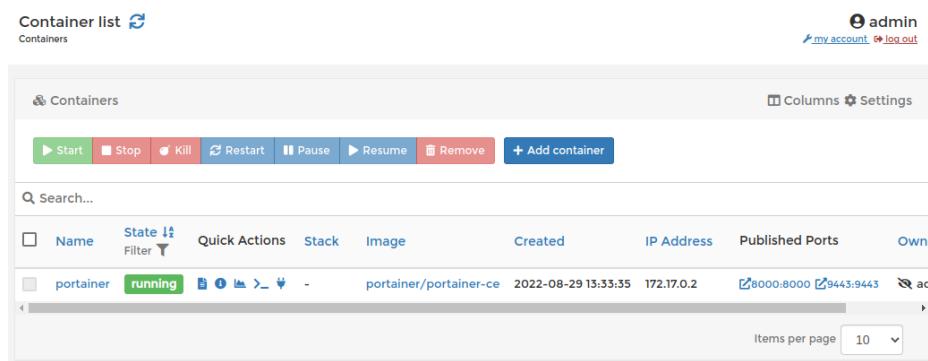


FIGURE 19.13
Portainer dashboard list down of all running containers.

20

Kubernetes

CONTENTS

20.1	Kubernetes Basics	273
20.1.1	Infrastructure	274
20.1.2	Installation	275
20.1.3	Kubernetes Configuration Files	276
20.1.4	Cluster Deployment	277
20.1.5	Cluster Update	281
20.2	Kubernetes Advanced	281
20.2.1	Kubernetes Object: Deployment	281
20.2.2	Kubernetes Object: Service	282
20.2.3	Kubernetes Object: Persistent Volume Claim, Persistent Volume, and Volume	284
20.2.4	Kubernetes Object: Secrets	287
20.2.5	Kubernetes Environment Variables	287
20.3	Container Deployment in Production Environment	288
20.3.1	Setup Cloud Account	289
20.3.2	Configure CI/CD	290
20.3.3	Deploy Containers	290
20.3.4	Manage Secrets	291
20.3.5	Helm	291

Kubernetes is one of the most widely used container orchestration tools. Many cloud platforms provide Kubernetes support. Many opponent container orchestration tools are built on top of Kubernetes.

20.1 Kubernetes Basics

Kubernetes, also known as *k8s*, is an open-source container orchestration system originally developed by Google. It automates the deployment, scaling, and management of containerized applications. While Kubernetes is more flexible than Portainer, it's also more complex and has a steeper learning curve.

Notice that running Kubernetes on a local server differs significantly from

running it on a cloud platform that often offers managed Kubernetes services with additional features and simplified managing interface. For example, AWS has Elastic Kubernetes Service (EKS) and Google cloud Google Kubernetes Engine (GKE). Both of them have developed their own tools and interfaces for interacting with Kubernetes. So, while learning Kubernetes can be beneficial, one might not need to learn all the low-level details if he is using one of the above platforms instead of DIY everything.

The majority part of this chapter focuses on introducing running Kubernetes on a local server, in which case *kubectl* is used as the user interface to Kubernetes and *minikube* the software to manage the host machine. Minikube is an open-source software developed by the Kubernetes community to run a single-node Kubernetes cluster on a local machine, which is suitable for developers to learn and test different things in development environment.

Notice that in the past days, docker is the default container engine built-in to Kubernetes, and Kubernetes uses a special program “dockershim” to talk to the docker engine. As of now, docker support is deprecated in Kubernetes and dockershim is removed from the installation.

20.1.1 Infrastructure

Figure 20.1 demonstrates the key components Kubernetes has inside its cluster. As shown in Fig. 20.1, Kubernetes manages containers in a centralized

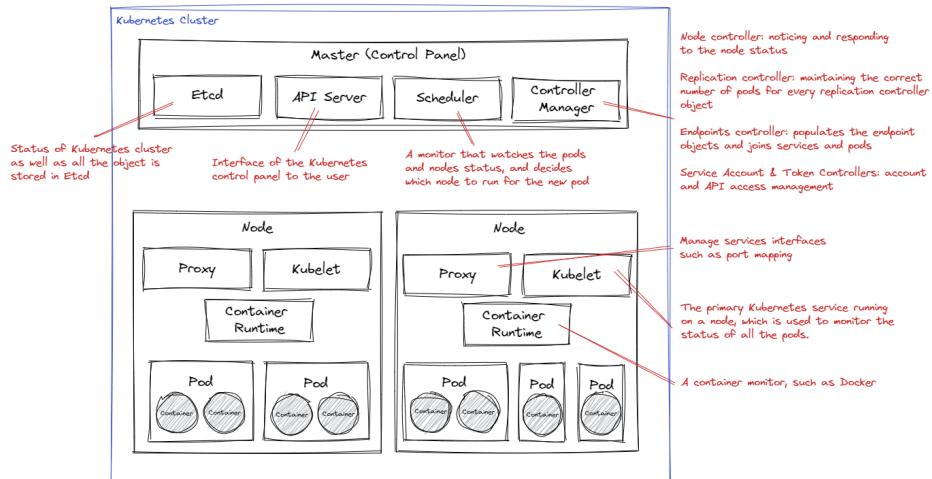


FIGURE 20.1

Kubernetes cluster and its key components.

“master-worker” manner, where the master plays as the control panel to interact with a user, and the worker nodes (nodes, for short) process the data. Each node can host multiple pods. Inside each pod is a container or a group

of containers that work together closely (dependently, in many cases). Notice that in Kubernetes, containers never run directly in a node. They are always grouped into pods. A pod should be the minimal unit that can deliver a basic function.

The master and nodes can run on cross servers or VMs. Kubernetes packages need to be installed on each and every server or VM. Kubernetes provides variety of tools to distribute the loads to different servers or VMs, or to add redundancy to the system for high availability.

20.1.2 Installation

The installation guidance of Kubernetes can be found at its official website [kubernetes.io](https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/). Notice that different OS adopts different ways of installing and using Kubernetes. The installation procedures introduced in this section applies to Linux OS only. For Windows users, Kubernetes can be installed from Docker desktop. For macOS users, other tools are used to install the tools.

As introduced earlier, *kubectl* is used to interact with Kubernetes. In addition, since we are in development environment, we will also install *minikube* which is used to setup a small Kubernetes cluster in the local machine. They can be installed separately. See following links for more details.

```
https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/  
https://minikube.sigs.k8s.io/docs/start/
```

and do not forget to start *minikube* using the following command. Notice that when both *minikube* and *kubectl* are installed, *minikube* needs to be started before *kubectl* can work properly.

```
$ minikube start
```

Upon the start of *minikube*, use the following command to verify the successful running of *kubectl*.

```
$ kubectl cluster-info  
Kubernetes control plane is running at https://192.168.49.2:8443  
CoreDNS is running at https://192.168.49.2:8443/api/v1/namespaces/kube-  
system/services/kube-dns:dns/proxy
```

Notice that when starting *minikube*, it would launch a VM using Virtual Box, and install *kubectl* as a built-in. Therefore, it is probable to use that built-in *kubectl* instead of manually installing one separately. In that case, use the following command to verify the successful running of *kubectl*

```
$ minikube kubectl cluster-info  
Kubernetes control plane is running at https://192.168.49.2:8443  
CoreDNS is running at https://192.168.49.2:8443/api/v1/namespaces/kube-  
system/services/kube-dns:dns/proxy
```

in which case `alias kubectl="minikube kubectl --"` may make things easier.

It is possible to run Kubernetes cluster directly on a host machine OS without VM if that machine is running Linux. However, this is not recommended for reasons pertaining to access control, security, and isolation. It is generally a better practice to deploy Kubernetes clusters in a VM.

The `kubectl` command is a command-line interface (CLI) for interacting with Kubernetes clusters. Its behavior is governed by a configuration file, typically located at `~/.kube/config`, which determines which Kubernetes cluster `kubectl` communicates with. This could be a cluster running on the host machine, or a cluster running in a VM managed by tools like Minikube.

20.1.3 Kubernetes Configuration Files

Kubernetes requires that all images to be used are pre-built. When using Kubernetes, multiple configuration files are required, each file corresponding with an object to be created. Notice that an object is not necessarily a container. It can be a pod, a replica controller, a service, or any item in the Kubernetes framework. There are manual setups of networking. Details are introduced in the remaining of this section.

The following two configuration files are given as examples to demonstrate how the configuration files of Kubernetes look like. This example comes from Udemy course *Docker and Kubernetes: The Complete Guide* by Stephen Grinder [9]. They are both written in YAML. Configuration file to setup a pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: client-pod
labels:
  component: web
spec:
  containers:
    - name: client
      image: <image-name>
  ports:
    - containerPort: 3000
```

Configuration file to setup the networking service:

```
apiVersion: v1
kind: Service
metadata:
  name: client-node-port
spec:
  type: NodePort
  ports:
    - port: 3050
      targetPort: 3000
```

```
nodePort: 31515
selector:
component: web
```

Commonly used Kubernetes object types are summarized in Table 20.1. Some highlights of the above configuration files are as follows.

- **apiVersion** plays as the prefix that decides what configuration types are supported. For example, under the scope of v1, Pod, Service, configMap, Namespace, etc., are supported. In a different apps/v1, a different set of configuration types ControllerRevision, StatefulSet, etc., are supported. It's important to choose the correct apiVersion for the Kubernetes API version to ensure the compatibility and availability of the desired configuration options.
- **kind** This indicates the type of the object that the configuration file describes. For example, Pod represents a pod that is used to host containers, and Service the primary object type that defines networking, with subtypes NodePort (see example above), ClusterIP, LoadBalancer and Ingress.
- **metadata** indicates the name and labels of the object. For example, component: web is defined as a label of the pod. This information is passed to the networking service under selector, so that the networking service knows which object it should link to.
- **port**, **targetPort** and **nodePort** are used to specify ports used in the service. The targetPort indicates which port in the pod should be exposed to the service, and it is consistent with containerPort defined in the pod. Assume that there is another pod in the node who needs to talk to this pod via the service. The other pod's port that communicates with the service is fed into port. Finally, nodePort is the port with value between 30000 and 32767 that is exposed from the service to outside the node. If nodePort is not assigned, a random number within the range will be assigned. This is shown in Fig. 20.2.

Notice that Kubernetes networking using kind: Service is more complicated than shown in the above example. More details of it is given later in a dedicated Section 20.2.2.

20.1.4 Cluster Deployment

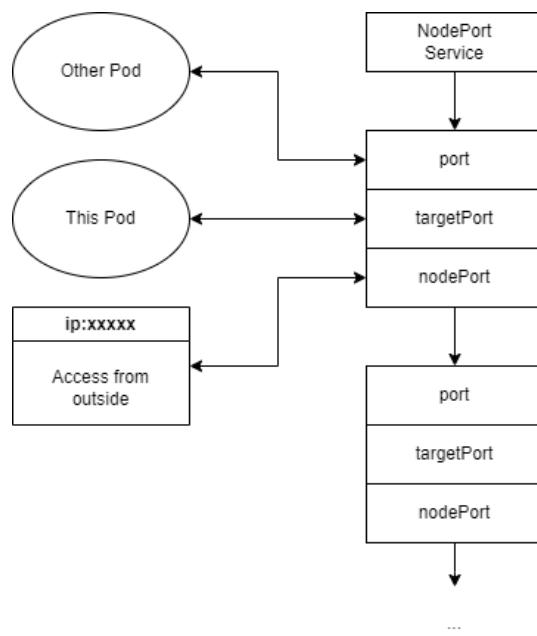
With the image and the configuration files ready, the next step is to deploy the nodes, pods, and containers. *kubectl* command line interface is used to instruct Kubernetes to deploy the objects as follows.

```
$ kubectl apply -f <configuration file>
```

TABLE 20.1

Commonly used Kubernetes object types.

Object Type	Description
Pod	The smallest and most basic unit in the Kubernetes object model. It represents a single instance of a process running on the cluster.
Deployment	Manages the deployment and scaling of a set of identical pods, ensuring the desired number of replicas are running and providing rolling updates for seamless application upgrades.
Service	Enables network access to a set of pods using a stable IP address and DNS name. It provides load balancing across multiple pod replicas and allows external traffic to be directed to the appropriate pods.
ConfigMap	Stores configuration data in key-value pairs, which can be consumed by pods as environment variables, command-line arguments, or mounted as files.
Secret	Similar to a ConfigMap, but specifically designed to store sensitive data, such as passwords, API keys, and TLS certificates. Secrets are encrypted at rest and can be mounted into pods as files or exposed as environment variables.
PersistentVolume	Provides a way to provision and manage persistent storage resources in a cluster. It decouples the storage from the underlying infrastructure and allows data to persist beyond the lifecycle of individual pods.
PersistentVolumeClaim	Requests a specific amount of storage from a PersistentVolume. It acts as a request for a specific storage resource and provides an abstraction layer for managing persistent storage in a cluster.
Ingress	Manages external access to services within a cluster. It acts as a reverse proxy and exposes HTTP and HTTPS routes to route traffic to the appropriate services based on hostnames, paths, or other rules.

**FIGURE 20.2**

The NodePort networking service.

This essentially asks the master node in the Kubernetes cluster to start taking actions according to the configuration files, such as to inform the nodes to start creating pods and containers. The master node also keeps monitoring the status of each work node, to make sure that everything is running as planned. If there is a container failure, etc., the master node will guide the associated node to restart the container.

It is worth mentioning here that by default Kubernetes uses declarative deployment instead of imperative deployment, meaning that the developer does not need to specifically tell Kubernetes what to do in each step. The developer only tells the overall objectives, and Kubernetes master node will try to figure out the steps to realize that goal. It is possible to enforce Kubernetes master node to practice specific details via configuration files, but it is almost always recommended to use the default declarative approach with Kubernetes.

To retrieve information, such as the status, of a group of objects, use

```
$ kubectl get <object type>
```

where `<object type>` can be `pods`, `services`, etc. For more details of a specific object, use

```
$ kubectl describe <object type> <object name>
```

for example, to check the containers running in a pod. If `<object name>` is neglected, Kubernetes returns detailed information of all objects of the given object type. For a running object, use

```
$ kubectl logs <object name>
```

to check the log file of that object.

With the above been done, open a browser and use `<ip>:<port>` to access the application running in the container, where `<ip>` is the IP address of the VM (not `localhost`) that `minikube` created, and `<port>` the port configured in NodePort service under `nodePort`. The IP address can be found by running `minikube ip`.

To apply a group of configuration files all together, provide the directory name of all the configuration files to Kubernetes instead of feeding each configuration file one at a time.

```
$ kubectl apply -f <directory>
```

When directory is given instead of a file, Kubernetes will try to apply all the configuration files in that directory.

It is possible to consolidate the configuration files of objects into one conjunctive configuration file. To do that, use `---` to split the configurations for each object in the conjunctive configuration file as follows. It is of personal preference whether to use conjunctive configuration files or separate configuration files for all objects.

```
<configurations-for-object-1>
---

```

```
<configurations-for-object-2>
---
<...>
```

20.1.5 Cluster Update

Without container orchestration tools such as Kubernetes, one of the most challenging tasks is to update the container for a different configuration, for example, changing the underlying image. With the help of Kubernetes declarative approach, it is possible update the cluster simply by revising the configuration files, and pass them to Kubernetes as if the cluster is to be deployed for the first time. Kubernetes automatically checks the names and kinds of the revised configuration files, comparing them with existing running objects, and update them if necessary.

Check the status of the pods using `kubectl get pods`. After updating, the pods are often restarted, hence it is expected to see increment in the “RESTARTS” tag. To double confirm that updates have been made, use `kubectl describe` to check the details of the relevant objects.

However, there is a limitation to the updating of the Kubernetes deployment. For an existing object, only certain fields in the configuration files can be changed. For example, for a pod that runs containers, the image can be changed, but the container port cannot. Sometimes there can be a walk around. For example, in the case of changing container port of pods, consider using a new object type `Deployment` instead of `Pod`, which allows more flexible updating. The `Deployment` in its backend is consist of one or more monitored and managed identical pods.

To revert `kubectl apply`, i.e., to remove a configuration file, use

```
$ kubectl delete -f <configuration file>
```

Kubernetes treats the above delete command as a specific type of update to the cluster, and will action accordingly.

20.2 Kubernetes Advanced

This section introduces advanced commonly used Kubernetes objects, tools and techniques.

20.2.1 Kubernetes Object: Deployment

As introduced in Section 20.1.5, updating pods has some limitations. It is practically more convenient to setup pods using “Deployment” object instead of “pod”. The Deployment object servers as an additional layer of Kubernetes

infrastructure that manages identical pods. More details of Deployment object is introduced in this section.

As an example, here is a configuration file from Kubernetes manual that deploys a Deployment object.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
```

Some highlights are as follows.

- **replicas** gives the expected number of pods that the Deployment object manages.
- **matchLabels** specifies the pods with which label are to be managed by the Deployment object. In this example, the label is `app: nginx`. When populating pods, the pods would have the same label, as the same label is assigned under `template`, `metadata`, `labels`.
- **template** specifies the template that is used to create the pods.

When a new version of an image becomes available, we may want to update the containers accordingly. Re-apply the same configuration file would not help, as Kubernetes would reject apply request if no change is detected in the configuration file. It would not check whether the image is in its latest version. Kubernetes uses the following imperial command to update images as a walk around, and the developer needs to run this command manually.

```
$ kubectl set image deployment/<Deployment name> <container name>=<
  image name>
```

For example,

```
$ kubectl set image deployment/nginx-deployment nginx=nginx:1.25.1
```

20.2.2 Kubernetes Object: Service

There are 4 service types defined in Kubernetes. So far “NodePort” service type has been introduced in earlier examples. More types are introduced here.

A summary of different service types are given below.

- ClusterIP: Exposes the service object on a cluster-internal IP. Objects in the cluster can access to the object that a ClusterIP is pointing at.
- NodePort: Assigns a static port with the cluster IP, and exposes the Service object to the internet. This is used mostly used in development environment, not in production environment.
- LoadBalancer: Exposes the service object externally using an external load balancer. Kubernetes does not provide built-in load balancer.
- ExternalName: Maps the service object to the contents of the `externalName` field, such as a host name. This is related to cluster DNS server.

ClusterIP

A ClusterIP configuration file may look like the following.

```
apiVersion: v1
kind: Service
metadata:
  name: client-cluster-ip-service
spec:
  type: ClusterIP
  selector:
    <target tag>
  ports:
    - port: <port for internal comm>
      targetPort: <port for internal comm>
```

where the first `port for internal comm` is the port in the ClusterIP service object that opens to other objects in the cluster, and the second the port the target object opens to the ClusterIP service object. They can be set differently, but usually they are just set to the same value.

NodePort

NodePort has already been introduced earlier. It exposes the object to the internet with a static port, and it is used more in a development environment than a production environment.

LoadBalancer

LoadBalancer is a legacy way of getting network traffic into the pods. It is essentially an interface or a tool to bridge an Kubernetes Deployment with

an external load balancer. It will try to automatically configure the external load balancer using the configuration provided by the developer, while compromising the external load balancer rule.

Ingress

Ingress is a more commonly used Service type than LoadBalancer to get traffic into the Kubernetes containers. There are different types of Ingress, for example, Nginx Ingress by github.com/kubernetes/ingress-nginx. A demonstrative example of ingress service realization is given in Fig. 20.3. In this implementation framework, the configuration file (mainly a set of routing rules) of the object is used to define an “Ingress Controller” which manages the runtime that controls inbound traffic. In some applications such as [kubernetes/ingress-nginx](https://kubernetes.io/docs/concepts/services-networking/ingress/), the ingress controller and the runtime are integrated together.

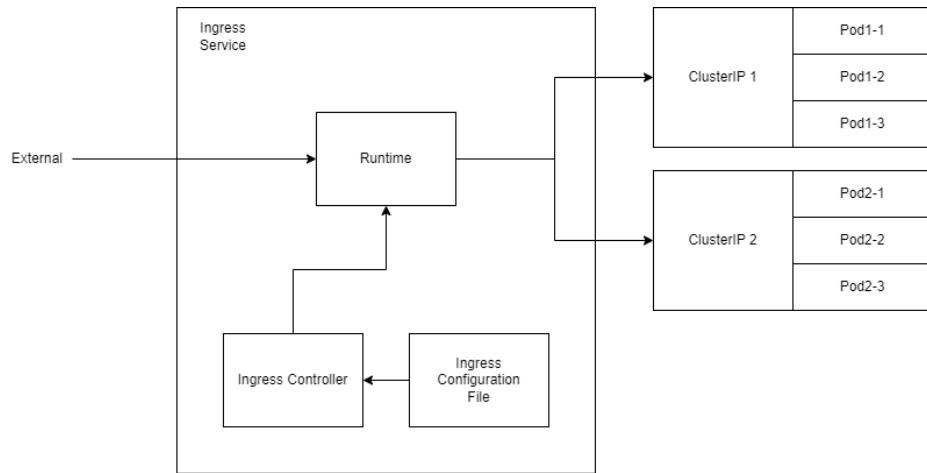


FIGURE 20.3

An example of ingress service framework.

The ingress configuration differs depending on the ingress service type and the platform. Details are not given here.

20.2.3 Kubernetes Object: Persistent Volume Claim, Persistent Volume, and Volume

Docker engine uses volumes to maintain persistent data and share data among containers. Details have been introduced in Section 19.4. Kubernetes volume framework is similar in the sense that it makes sure that the data is saved and managed by the host machine, so that when the pods or containers are shutdown or restarted, the data can be restored safely.

Do notice that when comes to data sharing using volume, it is dangerous to have multiple containers or the host machine accessing the same files simultaneously, without knowing the existence of each other. Usually additional steps need to be setup to ensure data consistency.

It is worth emphasizing the differences of “volume” technology in containerization and Kubernetes volume-related objects: Persistent Volume Claim (PVC), Persistent Volume (PV), and Volume. As a matter of fact, Kubernetes Volume object is usually not what we want. Kubernetes Volume creates the volume tied to a pod, not to the host machine. It survives container failure in the pod, but not pod failure. In summary:

- Kubernetes Volume: A volume tied to the pod. It survives container failure inside the pod, but not pod failure.
- Kubernetes PV: A volume tied to the host machine. It survives pod failure. It can be provisioned either automatically by a StorageClass, or manually by the developer and administrator.
- Kubernetes PVC: It is essentially a request sent from a pod or a container, asking for specific amount of storage from a PV. Kubernetes will find that amount of PV from either existing provisioned static PV, or dynamically provision new ones for the pod or container.

Notice that it is not necessary to claim PV in order to use PVC, as PVC can provision PV dynamically. There is a one-to-one relationship between the provisioned PV and the PVC. If there are multiple pods, each requiring a dedicated PV, then multiple PVCs must be used. The developer can either create those PVCs manually, or use a Volume Claim Template to claim them if they are similar.

An example of claiming Kubernetes PV and PVC is given below. In the remaining part of this section, we will be mostly using PVC instead of PV.

```
# persistent-volume.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  storageClassName: standard
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /data/my-pv
---
# persistent-volume-claim.yaml
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  storageClassName: standard
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

To check the PV objects, use `$ kubectl get pv` and `$ kubectl get pvc`.

To add the above Kubernetes PVC to a Kubernetes Deployment, add volumes information to the specs as given in the following example.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app-container
          image: my-app-image
          ports:
            - containerPort: 8080
          volumeMounts:
            - name: data-volume
              mountPath: /data
              subPath: data-from-container
          volumes:
            - name: data-volume
          persistentVolumeClaim:
            claimName: my-pvc
```

where `volumes` defines which Kubernetes PVC is used, and `volumeMounts` tells how it is mounted in the container. The `mountPath` is the path in the container whose data is mounted by the volume. If a `subPath` is given, a sub-folder of its specified name will be created in the host machine in the volume to contain the data.

There are different types of access modes:

- **ReadWriteOnce**: Allow one node to read and write at a time.
- **ReadOnlyMany**: Allow many nodes to read at a time.
- **ReadWriteMany**: Allow many nodes to read and write at a time.

The developer can specify the place for Kubernetes PVs. This is usually the hard drive on a local server, a virtual storage space in the VM. Use the following command to check Kubernetes possible choice of storage.

```
$ kubectl get storageclass
```

and

```
$ kubectl describe storageclass
```

When deploying Kubernetes on the Cloud, the developer needs to do additional configurations as there would be many storage options. Usually, each Cloud provider will have its own default storage space for Kubernetes, such as AWS Elastic Block Store for AWS.

20.2.4 Kubernetes Object: Secrets

Kubernetes Secrets object is used store confidential information, such as the database password, API key, etc. It is often a piece of information that is necessary for the containers, but the developer does not want to present as plain text in the configuration file.

Secrets are not created from configuration files, which is the recommended way of creating other Kubernetes objects. Instead, it is created from one-time imperative command, inside which the confidential information needs to be told to Kubernetes. Use the following command to create a Secret object.

```
$ kubectl create secret <type-of-secret> <secret-name> --from-literal <key>=<value>
```

There are 3 types of Secrets: `generic`, `docker-registry` and `tls`.

20.2.5 Kubernetes Environment Variables

Kubernetes environment variables are used to pass or share information among Deployments. Depending on the features of the information, such as whether it is a constant global configuration or a dynamic value, whether it is plain text or confidential encoding, etc., it might be handled differently.

To define constant environment variables in containers, simply specify them in the Deployment configuration file as given in the example below.

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: my-app-deployment
```

```
spec:
replicas: 1
selector:
matchLabels:
app: my-app
template:
metadata:
labels:
app: my-app
spec:
containers:
- name: my-app-container
image: my-app-image
ports:
- containerPort: 8080
volumeMounts:
- name: data-volume
mountPath: /data
subPath: data-from-container
env:
- name: <name1>
value: <value1>
- name: <name2>
value: <value2>
- name: <name3>
valueFrom:
secretKeyRef:
name: <secret-name>
key: <key>
volumes:
- name: data-volume
persistentVolumeClaim:
claimName: my-pvc
```

where a new tag `env` is defined under template, specifications, containers. Under the `env` tag, a list is defined containing names and values of the environment variables. The value must be a string, not a numerical number.

20.3 Container Deployment in Production Environment

With the tools and methodologies introduced so far, we are able to deploy containers in development environment. This is good enough for testing purpose or for small individual projects. However, when comes to enterprise tier projects or collaborative projects, there is often a CI/CD pipeline that stan-

dardize the integration and delivery of the containers in production environment. Container orchestration tools such as Kubernetes is often a must have.

This section briefly introduces the steps to develop and deploy containers in production environment with Kubernetes. Figure 20.4 gives an example of overview of what such deployment may look like. Notice that this example is more towards a community project but not an enterprise project.

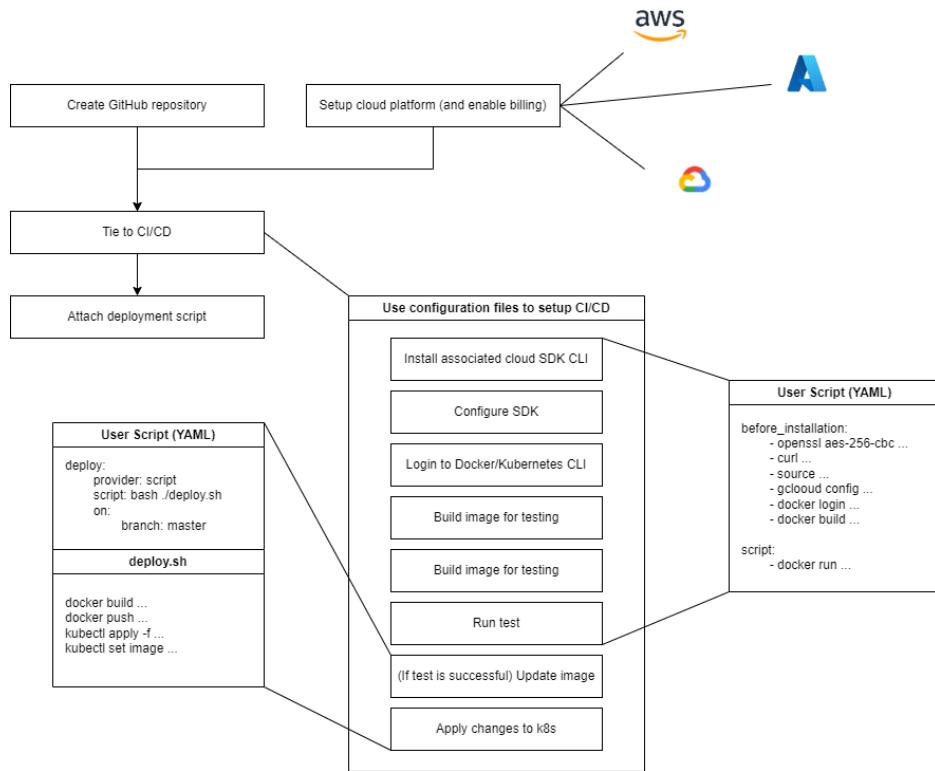


FIGURE 20.4

An example workflow of creating a production environment with Kubernetes.

The example used in this section to demonstrate Kubernetes CI/CD on a cloud platform in production environment is taken from [9]. Following [9], Google Cloud Platform is used as the cloud platform provider.

20.3.1 Setup Cloud Account

Many cloud platforms nowadays have a very good support of Kubernetes. The deployment of containers using Kubernetes can be done via their UIs easily. The developer needs to decide the resources to be used for the deployment. The more nodes and more power machine, the higher the charge. In most

cases, the developer does not need to start a VM and install Kubernetes on it by himself. The cloud provider shall have dedicated Kubernetes engine service that would automatically configure the VM per required.

20.3.2 Configure CI/CD

Travis CI is a continuous integration service tool written. It can be deployed on the cloud and linked to a Github repository and a CI/CD platform such as a Kubernetes cluster on Google Cloud Platform.

To run Travis, a machine supporting Ruby programming language is required. For that, a separate container with Ruby installation is deployed to run Travis. Use Github credentials to login to Travis, so that it can link to the Github repositories.

Travis has built-in file encryption function. This function is mainly used to encrypt login credentials and service account credentials (in this example, the service account information to link to Kubernetes clusters on Google Cloud Platform) locally, so that later the unencrypted original credential files can be deleted, and only the encrypted credentials uploaded to the Github repository. When encrypting a file, Travis will also guide the user on how to call the encrypt information in the build script.

Travis uses configuration files to setup CI/CD pipeline.

20.3.3 Deploy Containers

In the Travis configuration file, `deploy` is used to specify the script to run when the testing is successful. A separate bash script `deploy.sh` is defined for this purpose, inside which is a sequence of commands that builds and publishes images, and configures Kubernetes using `kubectl`.

It is particularly worth mention that in `deploy.sh`, when building and applying the latest version of the docker image, tagging using `<image-name>:latest` alone is not going to work for the same reason explained earlier in Section 20.2.1: when the same configuration with `<image-name>:latest` is applied, the system would simply acknowledge it as “no change” and would not actually download the latest version of the image.

The walk around introduced in Section 20.2.1 was to use version number in the configuration file and/or as an imperial command as follows.

```
$ kubectl set image deployment/<Deployment name> <container name>=<
    image name>:<version>
```

so what when the version name changes, Kubernetes would notice the differences and apply the new image. When working with CI/CD using *Git*, this can be further automated. Just use the `$GIT_SHA` as part of the tag as follows.

```
$ docker build -t <docker-id>/<image-name>:latest -t <docker-id>/<image
    -name>:$GIT_SHA -f <dockerfile> <save-directory>
$ docker push <docker-id>/<image-name>:latest
```

```
$ docker push <docker-id>/<image-name>:$GIT_SHA
```

Notice that in addition to `:latest`, `:$GIT_SHA` is used as a secondary tag. When pushing the built image to Docker hub, both `:latest` and `:$GIT_SHA` are pushed (although they have identical content). When setting image, the `$GIT_SHA` is used to identify the image just like the version number.

It is recommended not to remove `:latest` in the building command. This is because if someone wants to pull and test the latest image in his server (without knowing the value of `$GIT_SHA` for the latest commit), he is still able to do so using only the image name.

Notice that `$GIT_SHA` is not a built-in environment variable. The developer needs to set that environmental variable manually in the configuration YAML file as follows. It is possible to replace `$GIT_SHA` with a different name.

```
env:  
global:  
- GIT_SHA=$(git rev-parse HEAD)
```

With the above setup, `$GIT_SHA` can be used in `deploy.sh` as an environmental variable.

20.3.4 Manage Secrets

Notice that when CI/CD tool is tied to the cloud platform provider, service account authentication is required. It is a good habit to NOT to put the authentication information in the CI/CD configuration file as plain text, or to upload the unencrypted file that contains the authentication information to the public workspace. It is possible that some CI/CD tools provide encryption tools that can be used to encrypt the authentication file. In such case, the developer may need to install the required CLI for that CI/CD tool.

In Section 20.2.4, it has been introduced that Kubernetes uses Secrets object to encrypt secret files. The encrypted secret files can then be safely published online. In the Kubernetes configuration file, an environmental variable can be created to call the secret information.

Many cloud platform providers including Google Cloud Platform provides services to manage secrets.

20.3.5 Helm

To install a software in a Kubernetes pod, the most intuitive way is to commit the installation in the image, and call the image in the Kubernetes configuration file. For commonly used services such as `ingress-nginx`, its installation configuration file is available online as part of the manual. It essentially starts and initializes a branch of Kubernetes objects to enable the service.

Helm is designed as an alternative to manage software installation in Kubernetes clusters. In many occasions, it is more convenient (or even

only possible) to use Helm to install a software. More details are given in github.com/helm/helm.

We need to first install Helm from script. Helm installation used to contain two parts, the CLI (referred as Helm client) and the server (referred as Tiller server). We could then use Helm CLI to install other third-party software and tools.

Access control is important on cloud platforms. In practice, user accounts are used to identify users, and service accounts to identify pods and programs. Associated role bindings are used to manage what resources can be accessed by a user or program. For example, administrative role over the entire account can be used to bind with the administrative user. The same applies to Helm. The Tiller server required some extent of administrative control over the resources in an account. In many occasions, Tiller server was given the administrative permission to access the entire account, which introduced security risks.

As of Helm version 3, a major change was carried out where Tiller server was removed completely. Helm architecture is more secure and simpler today. The concerns related to Tiller's permission in the Kubernetes cluster are no longer relevant. Helm 3 of course requires permissions to use the resources, which is now managed by Kubernetes role-based access control mechanisms.

21

HTTP Server

CONTENTS

21.1	Brief Introduction to Apache HTTP Server	293
21.2	Installation of Apache HTTP Server	294
21.2.1	Apache HTTP Server Installation on Host Machine	294
21.2.2	Apache HTTP Server Execution in Container	294
21.3	Apache HTTP Server Configuration and Deployment	295
21.3.1	Virtual Host Configuration	296
21.3.2	Kerberos Authentication Configuration	297
21.4	A Brief Introduction to Website Development	297
21.5	Other Web Servers	297

A commonly seen Linux application is to host a web service. “LAMP”, an acronym that stands for Linux, Apache, MySQL, and PHP, is a classic architecture for that application. In this structure, Linux serves as the host OS for the server or for the container orchestration, Apache the web service provider, MySQL (or its alternatives) the backend database, and finally PHP the programming language for web development.

Database services, both RDB and NoSQL, have been introduced in earlier chapters. Apache HTTP server is introduced in this chapter. Programming languages for web application such as PHP, Java, .NET framework, c#, JavaScript and CSS are not covered in this notebook.

Notice that Apache is not the only web service provider in the market. Alternative choices include Nginx, Node.js and many more. They are briefly introduced in the end of the chapter.

21.1 Brief Introduction to Apache HTTP Server

Web server is a network service that serves contents to a client over the internet. The client can be a web browser or any software that supports the pre-agreed communication protocol, one of the most popular ones being the hypertext transport protocol (HTTP). Details about HTTP protocol is not introduced in this notebook.

Apache HTTP server, also known as *httpd* in RHEL, is one of the web service providers in the market. It is an open-source web server project developed by the Apache Software Foundation. A full list of projects managed by the Apache Software Foundation can be found at [10]. Some of the widely appreciated applications in the list include Apache Spark, Apache Iceberg, and many more.

This chapter introduces the basic configuration and usage of Apache HTTP server on RHEL. For more details or Apache HTTP server on other platforms, visit the Apache HTTP Server website at <https://httpd.apache.org/>.

21.2 Installation of Apache HTTP Server

Apache HTTP server can be installed and executed on the host machine or deployed in a container.

21.2.1 Apache HTTP Server Installation on Host Machine

The package for Apache HTTP server on RHEL repositories is *httpd*. To install Apache HTTP server on RHEL, use

```
$ sudo dnf install httpd
```

To start and enable *httpd*, use

```
$ sudo systemctl start httpd  
$ sudo systemctl enable httpd
```

A testing page is provided. Upon installation, retrieve the testing page by

```
$ curl localhost
```

and the test page HTML will show up in the console. Alternatively, use a web browser to visit the host server at port 80 to see the web page visually that looks like Fig. 21.1. Notice that firewall may need to be configured to allow HTTP TCP access on port 80, which can be done by

```
$ sudo firewall-cmd --permanent --add-port=80/tcp  
$ sudo firewall-cmd --reload
```

Firewall configurations will be introduced in more details in later part of the notebook under Linux security.

21.2.2 Apache HTTP Server Execution in Container

Apache HTTP server also has a Docker image, and it can run in a container. More information can be found at [11]. In short, one can build the image using the following Dockerfile

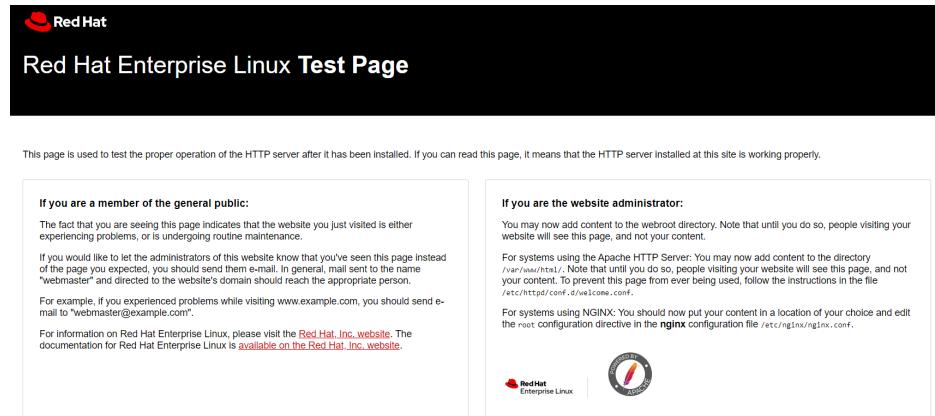


FIGURE 21.1
Apache HTTP server test page on RHEL.

```
FROM httpd:latest
COPY ./public-html/ /usr/local/apache2/htdocs/
```

followed by

```
$ docker build -t <image name> .
$ docker run -dit --name <container name> -p 8080:80 <image name>
```

where `./public-html/` is a directory on the host machine that contains all the HTML files. Alternatively, download and run the image from Docker Hub by

```
$ docker run -dit --name <container name> -p 8080:80 -v "$PWD":/usr/
local/apache2/htdocs/ httpd:latest
```

In the remaining of this chapter, we assume that Apache HTTP server is installed on the host machine. With that being said, all the configurations and functions should work similarly if it were run in a container.

21.3 Apache HTTP Server Configuration and Deployment

Configuration files setup the environmental variables (such as the location of the web pages, the port number, etc.) and control the behavior of the web server. Apache HTTP server configuration files include

- `/etc/httpd/conf/httpd.conf`: the main configuration file

- `/etc/httpd/conf.d/`: an auxiliary directory of configurations files included in the main configuration file
- `/etc/httpd/conf.modules.d/`: an auxiliary directory that contains configuration files for modules

Commonly used configurations are introduced in this section.

21.3.1 Virtual Host Configuration

Default Host VS Virtual Host

By default, an HTTP server deploys one website and all the website contents such as HTML files, JavaScript files and CSS files shall be stored in `/var/www/html/`. The idea of virtual host is to enable one HTTP server to deploy multiple websites, each stored in their associated subdirectory `/var/www/html/<virtual host name>/`. The virtual host information such as the port number, document root, etc., needs to be configured in `httpd.conf` the main configuration file.

Even if deploying only one website, it is still rather a good practice to use a virtual host than to use the default host for consistency and ease of management.

To deploy a virtual host, the main configuration file `httpd.conf` shall be appended with the following content.

```
<VirtualHost *:80>
    ServerAdmin <username>@example.com
    ServerName example.com
    DocumentRoot /var/www/html/example
    ErrorLog /var/log/httpd/example_error.log
    CustomLog /var/log/httpd/example_access.log combined
</VirtualHost>
```

where `example` and `example.com` can be replaced with the website name. Note that this is only a simple configuration with only the document root location, server name, and log locations. A practical virtual host configuration is usually more complex and filled with more details such as security policies, etc.

It is possible to add a configuration file for the particular virtual host at `/etc/httpd/conf.d/`. In this example

```
/etc/httpd/conf.d/example.conf
```

can be created and included as part of the main configuration file. This is optional and can become useful when the configuration is complicated.

With above setup, use

```
http://<server name>
```

to browse the virtual host, where

- The <server name> matches the `ServerName` or `ServerAlias` in the configuration file
- A DNS needs to be setup as a prerequisite, otherwise <server name> cannot be resolved.

21.3.2 Kerberos Authentication Configuration

Kerberos is a network authentication protocol. User registration and authentication have become a widely adopted feature in modern websites so that they can distinguish and provide different contents for different users. Kerberos can be used to back up this feature. Details of Kerberos is not given in this notebook.

RHEL uses GSS-Proxy module to provide support for Apache HTTP server running on it to perform Kerberos authentication. For that, make sure that `gssproxy` package is installed.

21.4 A Brief Introduction to Website Development

21.5 Other Web Servers



Part IV

Linux Security



22

Introduction to OS Security

CONTENTS

22.1	Basics	301
22.1.1	Risks and Attacks	301
22.1.2	General Security Architecture	302
22.1.3	Standards and Requirements	303
22.2	Elements of Security	304
22.2.1	Security Policy	305
22.2.2	Security Mechanism	305
22.2.3	Security Assurance	306
22.2.4	Trusted Computing Base	307
22.3	Access Control	309
22.3.1	Discretionary Access Control	310
22.3.2	Mandatory Access Control	311

Linux, as well as other OSs, uses variety of methods to protect the system and the data. The chapters in this part of the notebook introduce some of these security methods. As cloud computing is getting more and more popular, virtualization security is also discussed.

22.1 Basics

The background, motivation, and basic concepts of computer security are introduced in this section.

22.1.1 Risks and Attacks

No computer or OS are absolutely safe. Risks can be introduced by the subjects listed below.

- Software bugs. The OS and application software may have bugs which leave backdoors to malware.

- Malicious users. A user may perform illegal actions that damage other users sharing the same servers or services.
- Unauthorized access. The user or a program may intentionally or accidentally try to access confidential data that they should not view.

A risk will most likely not turn into an actual disaster by nature. However, when a hacker or a malicious user initiate an attack deliberately taking advantage of the risk, it may cause troubles.

The attacks can be divided into the following categories.

- Malware. The attacker disguises a piece of malware code as a legitimate software. When the software is executed, the malware carries out harmful activities.
- System Penetration. The attacker accesses a protected system bypassing security checks.
- Man-in-the-Middle Attack (MitM). The attacker intercepts communications between legitimate entities, and steal or modify the contents of the communication.
- Denial of Service (DoS). The attacker overwhelms a system and paralyzes its service by sending a lot of requests to the system, more than it can handle.
- Network Sniffing. The attacker passively logs information from the internet, and use them for future attacks.
- TEMPEST (Van Eck phreaking). The attacker collects and analyses data measured from electromagnetic emissions of devices such as mobile phones, and decode information from the measurements.
- Social Engineering. The attacker gathers information of the victims by cheating, phishing emails, etc.

To protect the users from the attacks, we need both computer security and communication security. This notebook focuses on computer security, specifically OS security.

22.1.2 General Security Architecture

It is impractical to secure every component of a system (hardware, OS kernel, OS services, application services, user interface, users, etc.) with a singular universal protection method. Therefore, a more common approach is to implement a layered security architecture as shown in Fig. 22.1. In this paradigm, the system is segregated into distinct layers such as the hardware layer, the OS layer, and the application layer. Each layer employs its own security mechanisms targeting specific vulnerabilities.

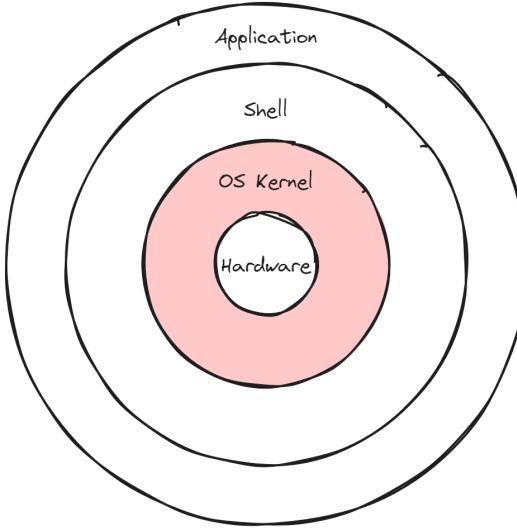


FIGURE 22.1
Layer structure of an operating system.

Among all these layers, securing the OS is particularly crucial for several reasons:

- Breaches in both hardware and application software often exploit vulnerabilities within the OS. By securing the OS, threats to other layers can be substantially mitigated. Even if certain applications are compromised, a robust OS can limit the extent and spread of the damage.
- If the OS is compromised, safeguarding other layers becomes extremely difficult. Most components across layers interact frequently with the OS and they typically operate under the assumption that the OS is trustworthy.

A system is considered secure if the following criteria are met.

- It is secure upon booting, and
- It never performs an action so that it can become insecure from a secure condition.

Notice that just for clarification, there are some slight differences between “secure” and “trusted” as follows. System security is the ultimate goal. A system is either secure or insecure, and we want it to stay secure all the time. Trustworthy, on the other hand, is a graded feature that we use to describe an entity in the system. We can, for example, say which parts (services, users, etc.) of the system is trustworthy, and to what extent they can be trusted.

22.1.3 Standards and Requirements

There are many official standards on computer security. For example, Trusted Computer System Evaluation Criteria (TCSEC) published in 1985, also known as the “orange book”, is one of the earliest standards in this domain. It divides system into different tiers in terms of security, including

- A: Verified protection
- B: Mandatory protection
 - B3: Security Domains
 - B2: Structured Protection
 - B1: Labeled Security Protection
- C: Discretionary protection
 - C2: Controlled Access Protection
 - C1: Discretionary Security Protection
- D: Minimal protection

Notice that TCSEC is considered outdated due to the rapid advancement of technology. Nowadays, commercialized PCs and OS such as Windows 11 pro, MacOS, RHEL, implement robust security mechanisms that align with various aspects of TCSEC criteria of different tiers, some of which required by Tier B and even Tier A. While TCSEC remains a classic and milestone, newer standards have been developed and adopted globally by various countries and organizations.

In the scope of Linux, there is an open-project, “Security Enhanced Linux (SELinux)” that enables mandatory access control in Linux. It started as an add-on module of Linux kernel, and today it has become a default module.

In general, the requirements of secure computer systems include

- Confidentiality. Data, as well as the existence of the data, is not leaked to unauthorized entities.
- Integrity. The data can be trusted, and it cannot be modified by unauthorized entities.
- Accountability. It is possible to trace and audit the actions performed by users and programs.
- Availability. The system should be resistant to attacks and consistently provide services.

The primary objective of studying computer security is to ensure that the aforementioned requirements are consistently met and upheld. Regrettably, there is no systematic approach guaranteeing that these requirements are met at all times.

22.2 Elements of Security

Key components in security schema include:

- Security policy. It defines what needs to be protected and what the desired security outcomes are.
- Security mechanism. It defines the tools, methods, and procedures employed to enforce the security policy.
- Security assurance: It defines the means by which we evaluate the efficacy of the security mechanism.

22.2.1 Security Policy

A security policy establishes the standards and objectives that a system must adhere to, outlining the rules that both users and programs are expected to follow. Deviations from these stipulations or breaches of the rules can compromise the security of the system. A security policy often comprises a set of sub-policies, which may be categorized into areas like confidentiality policies, integrity policies, and so on.

Different systems may implement different policies. There are two commonly seen security policies that concern most of the systems. They are

- Confidentiality policies: preventing the unauthorized disclosure of information.
- Integrity policies: preventing the unauthorized alteration of information.

For the sake of clarity and to ensure that no misunderstandings arise, it is crucial for the security policy to be articulated in a precise and consistent manner. Instead of relying on colloquial or vague terminology, we use security policy model and policy language to formally and precisely describe the security policy. Security policy model should be ambiguity-free and easy to comprehend. Though it does not assume or restrict the security mechanism to be used to fulfill the policy, it should give some guidance to how the mechanisms can be designed. At the minimum, it should make sure that the policy is reachable.

More details of security policy, especially security model, is introduced in later sections. There are classic security models such as HRU model that has been proved useful and inspiring.

22.2.2 Security Mechanism

Security mechanisms are the means and tools to fulfill the security policies. Security mechanisms can be widely divided into 3 types:

- Prevention: (most commonly seen) protect system from being damaged.
- Detection: detect potential risks and damages.
- Recovery: recover a compromised system back to a secure system.

Different systems uses different security mechanisms to fulfill different security policies. Some important concepts are introduced below.

One of the most commonly used security mechanisms is access control, which monitors and controls the accessibility of a resource (known as objects) from users or programs (known as subjects). Access control is managed by reference monitor. Reference monitor refers to the combination of hardware and software that practices access control using the architecture given in Fig. 22.2.

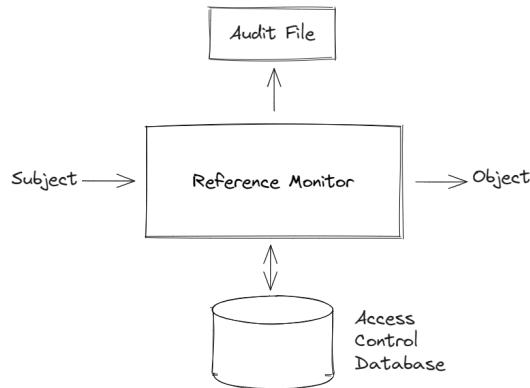


FIGURE 22.2
Reference Monitor Architecture.

Another important concept is security kernel. Security kernel refers to a small piece of code running in the kernel of OS that addresses system security.

Both reference monitor and security kernel shall have the following features:

- Completeness. They must be active all the time, and no process can bypass them.
- Isolation. Their content cannot be modified by unauthorized personals.
- Verification. They can be audited, and their effectiveness can be proved and verified. (This implies that their realization cannot be too complicated.)

22.2.3 Security Assurance

Security assurance refers to the degree of confidence in the security features, practices, procedures, and architecture of an information system. It ensures that the system enforces the security policy effectively. Below are key aspects of security assurance:

- **Verification and Validation:**

- Verification: Checking that the system complies with specifications and is correctly implemented.
- Validation: Ensuring the system meets user's needs and its intended purpose.

- **Assessment and Evaluation:**

- Evaluating the effectiveness of security controls.
- Assessing compliance with security standards.

- **Testing:**

- Conducting tests to identify security vulnerabilities.
- Penetration testing and automated vulnerability scanning.

- **Certification and Accreditation:**

- Certification: Evaluation of security features of a system.
- Accreditation: Approval to operate in a secure environment.

- **Risk Management:**

- Identifying, assessing, and mitigating risks.

- **Audit and Compliance:**

- Regular audits for policy and standard compliance.
- Maintaining logs and records for security events.

- **Continuous Monitoring:**

- Real-time threat detection and response.

- **Documentation:**

- Maintaining records of security policies, procedures, and changes.

- **Training and Awareness:**

- Training users and administrators in security best practices.
- Creating organizational awareness of security threats.

Security assurance is an ongoing process that is critical for ensuring that a system is secure by design, in implementation, and in deployment, adapting to new vulnerabilities and attack vectors over time.

22.2.4 Trusted Computing Base

System boundary refers to the boundary between the system and outside world. Everything in the system is protected by the system following the requirements specified by the security policy.

Security perimeter refers to the imaginary boundary that distinguishes security-relevant components and non-security-relevant components in the system. Security-relevant components include OS, system files, administrators and his files and programs, etc. Non-security-relevant components include user program, user profiles, I/O devices, etc. A demonstrative figure from [1] is given in Fig. 22.3.

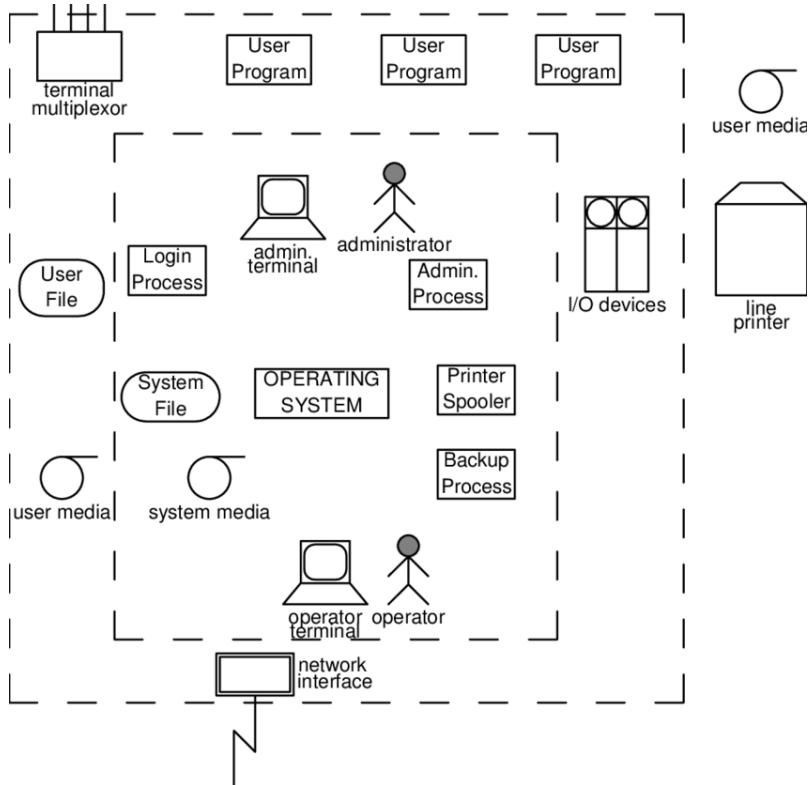


FIGURE 22.3
Reference Monitor Architecture [1].

The Trusted Computing Base (TCB) encompasses all the hardware and software components that are crucial for enforcing a system's security policy. This dual perspective implies that:

- Security Assurance. Efforts must be made to ensure that the TCB components are as secure as possible. Any vulnerability within the TCB could

potentially compromise the security of the entire system. Therefore, the integrity, confidentiality, and availability of the TCB are of utmost importance.

- Assumption of Trust. When designing security mechanisms for the system, the TCB is assumed to be inherently secure and trustworthy. Security mechanisms rely on the TCB to operate correctly and to enforce security policies effectively. This assumption is fundamental to the system's architecture, as the TCB underpins the security of all other components.

The TCB can be likened to a police department in a city. Just as citizens rely on the police force to uphold the law and protect public safety, a system relies on the TCB, the “police security bureau” of the system, to enforce its security policies and maintain the integrity of its operations. Similarly, the well-being and readiness of the police officers are paramount to the security of the city. In the case of the TCB, ensuring the security of these trusted components is crucial because a compromise to any part of the TCB could undermine the security of the entire system. The system defenses are only as robust as the TCB’s integrity and resilience.

Non-trusted components, while not central to the system security architecture, are analogous to the various elements of a city that the police department does not directly oversee. If these elements are compromised, the immediate risk is localized. However, securing non-trusted components is also important. We always want each and every component in the system to work properly. Maybe more importantly, we want to prevent them from becoming weak links that could be exploited to attack the TCB.

The goal is to keep the TCB, our “police department”, as streamlined and strong as possible. A smaller, well-protected TCB simplifies the task of maintaining system security and reduces the potential attack surface.

22.3 Access Control

In the realm of computer security, a “subject” refers to an active entity that initiates an action, typically a user who takes an active role. On the other hand, an “object” refers to a passive entity that receives or is acted upon by the action. In most contexts, objects are files, data, or programs. Processes and threads can simultaneously act as both subjects and objects.

The term “access” is used to describe when a subject performs actions on an object. This can encompass various activities such as creating, reading, executing, editing, or deleting the object. Access control is a mechanism that assists systems in determining whether to grant or deny specific permissions, dictating which subjects can access particular objects and what actions they can undertake with those objects.

Typically, a user who plays subject is required to first identify himself by logging into an account of the system using a valid authentication methods. A user or a program that plays subject needs to be assigned with a “role”. The program can only execute if its role possesses the necessary permissions to do so, including access to required resources like CPU, memory, disk space, and databases.

We use access control matrix to describe the association of subject and object. A demonstrative graph is given in Fig. 22.4. In this demonstration, like introduced in Section 4.3, “read”, “write” and “execute” permissions are controlled, each using 3-bit “r”, “w” and “x”, together forming the nine permission bits.

	File/Program 1	File/Program 2	File/Program 3
User 1	rwx	rx	r
User 2	rwx	r	-
User 3	r	r	-

FIGURE 22.4

A demonstration of access control matrix.

The most popular ways to manage the access control matrix include discretionary access control (DAC) and mandatory access control (MAC).

22.3.1 Discretionary Access Control

In the DAC model, the owner of the data, or the group to which the owner belongs, determines the access control matrix for the data. This means that the owner has complete authority over who can access the data and in what manner. Additionally, the owner has the flexibility to delegate full access privileges to others if desired. This level of flexibility makes DAC a popular choice in many operating systems, including Linux.

Under DAC, the operating system can track the capability of each user’s access to data. One method involves using a capability list, which outlines all the data a user can access. The users cannot modify their own capability lists. However, they can extend their access capabilities to other users. When a capability list is associated with each subject rather than individual users, it is referred to as a profile. Conversely, an object can be assigned an ACL as introduced in Section 4.3 that records which users or subjects have access. This latter method of assigning ACLs to objects is more widely adopted.

As mentioned in Section 4.3, in Linux access control is managed through a system of “9 permission bits”. These bits define the accessibility for three

distinct user categories: the owner, the group (the primary group of the owner, or a group the owner specifies), and others. Each category's permissions are represented by three bits, indicating whether the data is readable (r or -), writable (w or -) and executable (x or -).

While this method is straightforward, it does have limitations, notably its simplistic division of subjects into only three categories, which may not offer sufficient flexibility for all use cases. But it suits most of our needs on a personal computer.

22.3.2 Mandatory Access Control

Though flexible, DAC adds risks to the system. A malware such as a Trojan horse is able to change ACL of files on behalf of the data owner without his notice or permission. To tackle this issue, consider MAC instead. MAC is popular on machines with sensitive data, such as government servers.

Unlike DAC where the owner can change the ACLs of his files, when MAC is implemented, the owner cannot change the ACLs of the files. Only the system can change the ACLs of the files according to predefined security policies.

As a special case of mandatory access control, consider multi-label security mechanism. In this implementation, security labels are assigned to both subjects and objects. Only the subject with a equal or higher security level than the object can access that object.



23

Security of Services and Applications

CONTENTS

23.1	Database Security	313
23.1.1	Database Security Risks Categories	314
23.1.2	Database Access Control	315
23.1.3	Security-Enhanced DBMS Solutions in a Glance	317
23.1.4	Outsourced Database Security	318
23.1.5	Big Data Security	319
23.2	Virtualization Security	319
23.2.1	Security Concerns	320
23.2.2	VMM Security	320

The previous chapter introduced general OS security mechanisms. This chapter discusses security mechanisms of services and software running on the OS such as database. Notice that network and communication security is a stand alone topic not considered in this chapter.

23.1 Database Security

Many Apps, both local and online, heavily rely on database to manage information. For example, online shopping APP uses database to record transaction history. Online banking uses database to store and manage customer capitals. The data stored in a database is often critical and confidential, and the database service providers need to try their best to prevent data loss and leaking, and to ensure the integrity and availability of the database.

When a database is under attack, the worst scenario may go beyond data damage. Examples are given below.

- Paralyze database service.
- Change and remove data illegally.
- Steel data.

TABLE 23.1

DB security risks categories and associated security methods.

	With Authentication	Without Authentication
Internal	Developer, manager, etc. Access control and audit	Irrelevant employee, etc. Encoding
External	Customer, vendor, etc. Outsourced DB security methods	Hacker, visitor, etc. SQL injection prevention

- Attack and gain unauthorized access to the underlying OS, and damage or control the entire server.
- Deploy Trojan horse program for other servers connected to the database server.

It is possible that the attacker hides and disguises the attacking command into SQL injection to open a back door or to retrieve unauthorized data such as confidential information.

Secure DBMS and Secure OS

It is worth introducing the relationship between a security-enhanced OS and security-enhanced DBMS. In short, they make a better each other, forming a “security chain” together. A secure OS boosts the security of the database. A poor OS, on the other hand, harms the DBMS security level because data is essentially stored on hard drive which can be penetrated if the OS is down.

There are two common ways of forming a security-enhanced DBMS from a normal DBMS. The first way is to upgrade the DBMS kernel codes for additional security features. This can be complicated and requires a high-level mathematics, databases and programming skills, but can provide a safe DBMS. Obviously, this applies to only open-source databases. The second way is to build a “wrapper” for the DBMS to interface with the users and API calls. This usually requires less skill sets and can be applied to both open-source and proprietary databases, but can only provide mediocre security.

23.1.1 Database Security Risks Categories

Depending on the identity and the access level of the attacker, database security risks can be divided into the following categories as shown in Table 23.1. Different database security risks categories may tell completely different stories on how an attacker plans his attack. For example, a developer may leave a backdoor in the program which he can use to access confidential data. A ven-

dor may bring the hard drive of the database outside the managed premises, after which he can use variety of tools to crack the database and obtain the data.

To tackle the challenges, different security methods need to be applied to prevent each and every risks category. A high-level summary is given in Table 23.1. Details are given in later sections.

23.1.2 Database Access Control

There are different types of access control, some of which widely adopted to all different types of databases, while others may apply to only high-level secure databases. Some access control schema apply to both database and OS, the most popular ones being DAC and MAC which have already been introduced in earlier sections when discussing OS security. They are re-addressed for database as follows.

DAC restricts access to objects based on the identity of the subject, i.e. the user or the group of the user. The accessibility of an object is determined by the owner of the object. The same idea has been adopted by Linux in file management.

As introduced earlier in database chapters, in DBMS, use syntax that looks like the following to grant and revoke access of an object from a subject.

```
GRANT <privilege> ON <table/view> TO <subject>
REVOKE <privilege> ON <table/view> FROM <subject>
```

In practice, it is common that the database manager sets up different set of views for different user groups, each set of views containing everything that the user group requires. Grant access to only the associated views to the user groups. This can hopefully prevent a user from accessing data confidential to him.

The problem of DAC is that it can “lose control” sometimes, making a user bypassing the restriction. For example, a user who has been revoked from access may still be able to access the data if he had created a procedure that reads the data, and his access to that procedure is not revoked. Many DBMS tries to provide some protection against this, for example, by integrating security labels into the SQL that the user injects. When a user execute an SQL command, in the backend the SQL command is “reformulated” to contain user authentication information. In a good implementation, this security mechanism should be made transparent to the user.

Another challenge is that sometimes the user legitimately asks for aggregated information which, however, can only be derived if he has access to data confidential to him. For example, consider a table that stores the scores of a class. A student wants to check his score as well as the average score of the class, which makes perfect sense. However, the scores of all the students are required to derive the average score. If the access of the student is limited to his score alone, he will not be able to get the average score of the class.

Different from DAC where the owner of an object choose his preference of who can access the data and the preference can change case by case, in MAC the rules are enforced by the system administrator consistently and globally. The user cannot overwrite security policy even on his own data. This reduces the chance of human error, hence providing a higher level of security. The cost is the flexibility in the user experience, and the complexity of setting up global rules especially on a large database. MAC is often used in government database where huge amount of confidential and sensitive data is managed with heavy responsibility.

In multi-level security DBMS (MLS), also know as multi-layer DBMS, each piece of data in a database is associated with a security label that reads like “unclassified”, “confidential”, “secret”, “top secret”, etc. This security label is a compulsory attribute to all the data. Users also have security labels that determines the level of accessibility. When querying the same database, different users will get different results based on the security level of each piece of data. The higher the security label of the user, the more information he is able to retrieve.

MLS is often used as part of MAC which is introduced earlier in this section.

In addition to query, the security labels also affect how `INSERT` and other database manipulation commands work. Obviously, the user needs to have a higher layer (or at least equal) security label than the data, in order to insert, edit or remove it.

There might be an interesting case where a row with higher security label already exists in the table, and a user with a lower security label who cannot detect that data wants to insert into the table a new row with the same primary key. In a normal database, this operation would have been rejected due to the duplication of primary key. However, in MLS, this action is granted. Otherwise, the user with lower security label would sense the existence of the higher security label data. This technology is known as polyinstantiation, a method used to avoid covert channels by allowing multiple rows with the same primary key but with different data, based on different security levels. Polyinstantiation occurs when multiple rows in table appear to have the same primary key when viewed at different security levels.

Covert channel refers to a “disguised” channel that transfer information between entities while violating the security policy. In many cases, the channel is built from a list of operations, all of which legitimate by itself alone. These operations, when combined together, creates this unexpected bug outbound designer’s intention. An example of a convert channel is as follows.

1. Entity A, with higher privileges, encodes secret information in binary format.
2. Periodically, entity A change the permission of a file that can be sensed by entity B. The encoded binary format is used to setup the permissions.

3. Entity B listens to the permission of that file to obtain the binary format data.
4. Entity B decodes the binary format to obtain the secret information.

By doing the above, entity A is able to transfer a secret information to entity B who has a lower layer security label and should not touch the data. Notice that entity A may not be a human traitor, but a malware program.

Covert channel uses unintended system mechanisms for communication, which is often low efficient. As a result, the bandwidth of the covert channel is often much lower than a regular communication channel. Besides, a fast covert channel is likely to be easier to detect, which is something that the hackers want to avoid.

23.1.3 Security-Enhanced DBMS Solutions in a Glance

To wrap up, different security-enhanced DBMS solutions are now available on the market as follows.

Normal DBMS on Security-Enhanced OS

In the early stage, no additional security features is added to the DBMS. It is just that the DBMS is running on a secure OS with MAC enabled. The database is often put into the group with highest sensitivity level. The problem of this implementation is that all users who legitimately access the database have to be in the same highest sensitivity group, which violates the principle of access control. The output of the database, whatever it might be, is considered generated from the highest sensitivity group, and needs to be audited each time before release. This severely adds human cost.

MLS

MLS, also known as “trusted database”, adopts security-enhanced DBMS that uses security labels to mark all the data and users. It is secure and flexible, and the details have been introduced in the earlier section. The only obvious problem for MLS is that it is difficult to realize. For third-party MLS, the customer never know whether there is a backdoor, unless he check all the codes (millions of lines of codes) that realize the DBMS by himself, which is enormously tedious and sometimes impossible.

Security Wrapper

An alternative to MLS is use normal OS and DBMS, and add a “filter” between DBMS and the users. This filter serves as a wrapper to the DBMS, and it uses security stamps to manage data transmission. All the data stored in the database can be encoded, and only users with the correct keys can decode them. This forms an encoded database with security wrapper.

Distributed Database

Till this point, it can be seen that the key to database security is to prevent data leakage from high security tier databases to low security tier databases. MLS tries to label the data carefully to isolate high security tier data from low security tier data, while the secure OS and secure wrapper try to prevent low security tier user from accessing high security tier data. Distributed database is another approach trying to further isolate the data of different security tiers: use different database, or even run them on different machines, in the first place.

Some system runs multiple DBMS kernels concurrently on a single server, each kernel managing a security layer of data. The compromise of one kernel does not necessarily mean that other layers are compromised. This may make the DBMS safer, but it will create high computational burden to the system. High security sometimes means low efficiency, and low efficiency can be bad for commercialized databases. It is also a challenge how to balance the trade-off between security, efficiency and cost.

Other system runs multiple DBMS kernels on concurrently on multiple servers, each server in charge of a security tier. Low security tier databases are synchronized to high security tier servers (bot not wise versa). This architecture design is robust to covert channel, but the data consistency and availability becomes a challenge.

23.1.4 Outsourced Database Security

Some enterprise outsources the database management and maintenance to IT companies such as Microsoft, Oracle or other database service providers. Security challenges introduced by cloud/vendor-based database differ largely from on-premises databases mainly because the DBMS itself is not reliable.

User-Encrypted Data: Prevent Data Leakage

There is a risk that the third-party database service provider may leak the data intentionally or unintentionally.

An intuitive solution of this problem is to encrypt the data in the user-end before saving into the database. A downside of this is that when we want to retrieve data from the database using an SQL query, the DBMS may have a difficult time interpreting and filtering the data. “Searchable encryption” is required in this case. It allows filtering of data without decrypting it first.

The result from the outsourced database needs to be audited, mainly to check the authenticity, completeness and freshness of the returned information. Part of the reason is that searchable encryption may fail to return the correct and complete result. More importantly, the underlying assumption is that we cannot entirely trust the DBMS in the first place.

Watermark: Prevent Data Modification

Watermark can be used to identify the owner and authenticity of the data.

It does not affect the normal usage of data, and it is hardly detectable by a third-party, but can be checked and audited conveniently by the party who assigned the watermark. Watermark shall be difficult to remove. The watermark is often added to the least significant bits of a numerical data.

Problems of watermark include

- It allows multiple entities to assign watermark to the same database. It is difficult to tell who the original owner of the database is.
- If a user wants to remove the watermark, he can simply remove all the LSBs of the data.

23.1.5 Big Data Security

As Industry 4.0 and IoT become more and more popular, we are collecting more data than ever in the history. Many activities such as building data driven models rely heavily on the big data. The major cloud service providers offer enterprise-tier databases optimized for big data management, both SQL and NoSQL such as Dynamo DB by Amazon and Cosmos DB by Azure, Microsoft.

There are some unique challenges when comes to the security of big data. This is not surprising as big data is generated, distributed and utilized very differently from the conventional data.

- Access control.
 - Data generated in big data comes from variety of sources by different types of users. It is difficult to track all the sources and all the users to determine the data accessibility and security tiers using traditional method.
 - AI model is used to assist categorizing sources and users, and assigning security tiers.
- Control and maintenance of data when it is distributed.
 - It is difficult to protect the privacy of the owner of the user when his data is published into a big data pool. It is possible to use AI model to find the owner of anonymously published data.
 - The key is to deny suspicious query and prevent a single entity from querying aggregated information.

23.2 Virtualization Security

Virtualization has been introduced in Chapter 19. While having many beneficial features, the use of virtualization brings new challenges to system security.

23.2.1 Security Concerns

Some major security concerns of VMs are listed below.

- Isolation between VMs.
- Migration of VMs.
- VM monitoring and supervisory.

VMs are naturally isolated due to the virtualization mechanism. Ideally, though running on the same physical server, a VM cannot bypass the monitored I/O to talk to another VM. When two VMs need to talk, VMM builds special communication channels for the VMs, usually a message queue or a piece of shared memory space. However, this does not guarantee the absolute isolation of information between two VMs. They may fail to wipe out all the data when handing over the hardware resources to another VM. There are potential covert channels where two VMs may communicate with each other, for example via the utilization rate of a hardware resource.

Migration happens frequently in VM applications due to its frequent scaling up and down. When migrating VMs from one server to the other, it is possible that the undercover malware is also migrated. In such cases, the malware can spread across servers and it will be difficult to trace back to its original source.

Conventionally, the MAC address of the hardware can be used as the unique identity of a machine. This does not apply to VMs. This makes some of the security measures difficult to implement.

When running a VM, the user who deploys the VM usually have administrative access to the VM guest OS. As introduced earlier in Chapter 19, VMM may allow VM guest OS to run some instructions in privileged mode. If there is a flaw in the VMM, the software running in the VM may take advantage of that and use it to attack other VMs hosted on the same server.

23.2.2 VMM Security

The security of VMM is key to the security of VMs as VMM has full control of the hardware and it manages and interacts with all the VMs. If VMM is compromised, all the VMs running on the system is exposed to high risk. In this sense, VMM is the single-point-of-failure to the entire system, hence needs to be monitored at all time.

A secure VMM architecture is helpful. Such architecture can be designed on top of an existing VMM. In practice, the secure VMM architecture may separate critical (“over-powered”) functions of VMM into different domains, and provide additional interface for security monitoring of the VMM.

|||||| HEAD ====== ||||| HEAD

A

Scripts

CONTENTS

A.1	<i>Vim</i> Configuration <code>vimrc</code> Used in Section 3	323
A.2	<i>Neovim</i> Configuration Files	324

This appendix chapter collects the example scripts used in this notebook.

A.1 *Vim* Configuration `vimrc` Used in Section 3

```
call plug#begin()
Plug 'vim-airline/vim-airline'
Plug 'joshdick/onedark.vim'
call plug#end()

inoremap jj <Esc>
noremap j h
noremap k j
noremap i k
noremap h i

noremap s <nop>
noremap S :w<CR>
noremap Q :q<CR>

syntax on
colorscheme onedark

set number
set cursorline
set wrap
set wildmenu

set hlsearch
exec "nohlsearch"
```

```
set incsearch
set ignorecase
noremap <Space> :nohlsearch<CR>
noremap - Nzz
noremap = nzz

noremap sj :set nosplitright<CR>:vsplit<CR>
noremap sl :set splitright<CR>:vsplit<CR>
noremap si :set nosplitbelow<CR>:split<CR>
noremap sk :set splitbelow<CR>:split<CR>
noremap <C-j> <C-w>h
noremap <C-l> <C-w>l
noremap <C-i> <C-w>k
noremap <C-k> <C-w>j
noremap J :vertical resize-2<CR>
noremap L :vertical resize+2<CR>
noremap I :res+2<CR>
noremap K :res-2<CR>

set scrolloff=3
noremap sc :set spell!<CR>
```

A.2 *Neovim* Configuration Files

Neovim is fork from *Vim* and it has gained popularity in the past years. Just like *Vim*, *Neovim* can be customized by user-defined configuration files. These configuration files are usually packed into *lua* file tree, where *lua* is a computer programming language that *Neovim* supports by nature.

B

Continuous Integration and Delivery

CONTENTS

B.1	Agile VS Waterfall	325
B.1.1	Waterfall	326
B.1.2	Agile	327
B.1.3	Roles in Agile-based Development	327
B.2	Continuous Integration Continuous Delivery	329
B.2.1	Pipeline	329
B.2.2	Continuous Integration	329
B.2.3	Continuous Delivery	331
B.3	<i>Github</i> Action (Part I: Basics)	333
B.3.1	Framework	334
B.3.2	<i>Actions</i> Triggers and Types	335
B.3.3	<i>Actions</i> Marketplace	336
B.3.4	Costs	336
B.4	<i>Github Actions</i> (Part II: Practice)	336
B.4.1	Trigger	337
B.4.2	Workflow	338
B.4.3	Execution	340
B.4.4	Very Complicated Actions	340

This notebook is mainly about Linux. In this appendix chapter, the boundary of the notebook is slightly expanded to software development, which is quite often how Linux is used in practice.

Continuous integration and delivery (CI/CD) is both a philosophical concept and a bunch of technology that speeds up the development, testing and deployment cycle of a software. It has become a common and beneficial practice that collaborative projects with rapid update implement the practice.

This chapter introduces CI/CD as well as tools to carry out CI/CD. In particular, GitHub, a widely appreciated CI/CD integrated platform to manage and share collaborative projects, is briefly introduced. The introduction of GitHub focuses on GitHub Action, the tool GitHub uses for CI/CD.

Some contents of this chapter come from [12].

B.1 Agile VS Waterfall

Agile and waterfall are both project management methodologies. Both of them are introduced here, starting with the more conventional waterfall model then Agile.

B.1.1 Waterfall

Speaking of project proposal, development, testing and delivery cycle, it is fairly intuitive to follow the procedures below:

1. Understand requirements from the user.
2. Design the architecture of the solution.
3. Develop the solution.
4. Test the solution.
5. Deliver the solution and close the project.
6. (Follow-up) maintain the solution.

The philosophy behind waterfall, as its name indicates, is to “follow the procedures and do not turn back”. When a previous step is considered completed, it is completed and should not be revoked or revised. This is demonstrated by Fig. E.1.

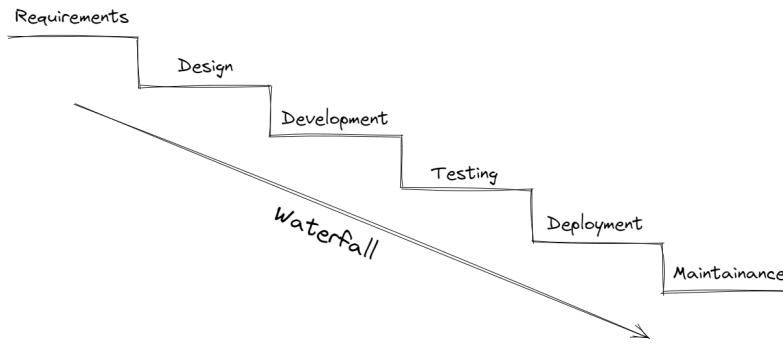


FIGURE B.1
Waterfall model.

With waterfall, a project can be designed, developed and deployed in a relatively efficient manner. However, there is one limitation: everything done in earlier steps cannot be changed in later steps. This sets up a high bar on

both the user and the developer. For example, in step 1 “understand requirements from the user”, the user needs to illustrate all the requirements to the developer as they cannot be modified in later steps. Similarly, in step 2 “design the architecture”, the developer needs to optimize the design to his best, as the architecture cannot be changed later.

Regrettably, with the rapid change in the market and the aggressive advent in technology today, it is challenging even for the smartest user and developer to determine all the requirements and designs in the beginning stage of the project. More likely, the requirements of the user have to change align with the market trend, and so does the design of the solution.

For a new feature to be added into the existing system, it is possible to simply start a new project flow for the new feature, and integrate it into the existing system later. However, integrating the new feature into the existing system can also be challenging when the design is complicated and coupled. The integration often introduces a blackout period of the system. If the system is already deployed, the customer experience would be affected by the blackout.

B.1.2 Agile

Agile is the counterpart of waterfall. It is proposed to tackle the aforementioned issue: rapid change of requirements and adaptations to new technology and tools. It allows continuous integration and deployment of new features into the system in a convenient and systematic way, without introducing blackout.

In agile architecture, each feature is separately modularized. Each feature, before deploying and integrating into production environment, circulates in its own “development and testing circle”, where it can be tested and reviewed iteratively by the developers and the users audit team, as shown below in Fig. E.2.

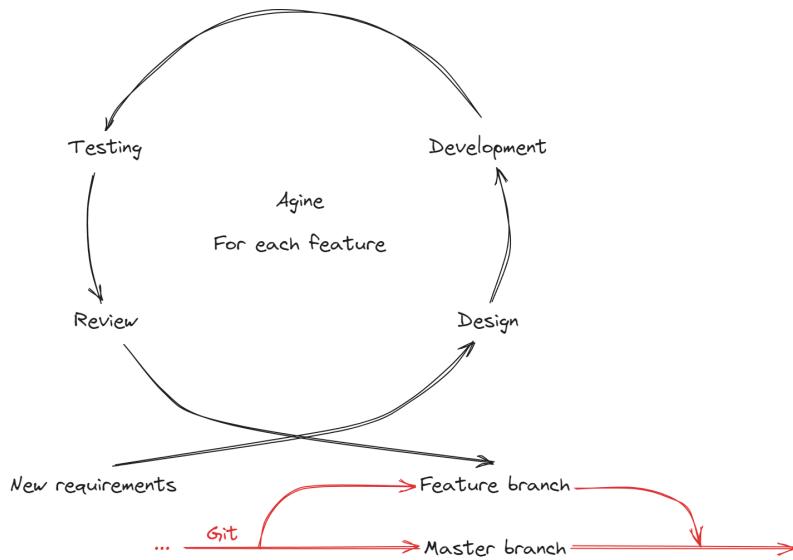
Agile allows rapid change to be made to the requirements and realizations of a feature. Should there be a change, just keep cycling in Fig. E.2 until the change is implemented and tested, before it is pushed back to the master branch.

In parallel development where there are multiple features running in their associated circles, the developer can easily choose which feature branch and circles to prioritize. This gives the developer a clearer overview of what is happening and how to best response to the customers immediate requirements.

B.1.3 Roles in Agile-based Development

The following roles are defined in agile, each role coming with a responsibility. The roles may slightly differ for each project. The most commonly seen approach, namely **scrum**, is introduced in Table. E.1.

In this role assignment, the product owner and scrum master come up with

**FIGURE B.2**

Agile model.

TABLE B.1
Roles in Agile model.

Role	Description
Product owner	Manage the entire program. He understands all the user requirements and progression of all features. He also signs off each feature when they are deployed.
Scrum master	Lead the developer team as team manager or chief developer.
Developer	Based on requirements, program the features.
Tester	Design test cases to verify the efficacy of the developed feature.
Operator/Supporter	Maintain the software

the product backlog, which clarify the sprints (tasks) and their priorities. The team then knows which sprints they shall work on first.

In the case where the features concerns a professional domain (such as economics, medical, etc.) that the developers cannot understand, business analysts are involved who bridge the user and the developer.

For each sprint, sprint planning and sprint backlog are proposed that describes the schedule of the sprint. The team works on the sprint and host daily scrum meetings until the sprint is solved. Upon finishing of a sprint, sprint review is hosted for audition.

B.2 Continuous Integration Continuous Delivery

CI/CD is introduced in this section. As a start, pipeline is introduced. Pipeline is an important concept used in CI/CD.

B.2.1 Pipeline

Pipeline is a set of data processing elements in a queue, where the output of the upper stream process is the input of the down stream process. An example is shown in Fig. E.3. By using a pipeline and let multiple pipelines at different execution phase run in parallel, the efficiency of the system is increased.



FIGURE B.3
Pipeline.

Pipeline is widely used in OS for process and thread management (recall Fig. 6.1). It is also used in CI/CD.

B.2.2 Continuous Integration

In the development of a sprint, new codes are rapidly developed, and they are rapidly compiled, packaged and tested.

Conventionally, the integration of a new feature requires involvement from multiple parties. An example is given in Fig. E.4. It includes the developer who program the software following users (or business analysts) requests, the integration team who integrates the new feature with the existing system and compile the code into packages, and the operations team who upload the new system into the pre-prod environment for real data testing, and the testers who audit the output of the program running in the pre-prod environment.

Should there be any error along the way, the code is roll back the developer team for trouble shooting. When the new system with the updated code survives pre-prod environment, it is then pushed to the production environment.

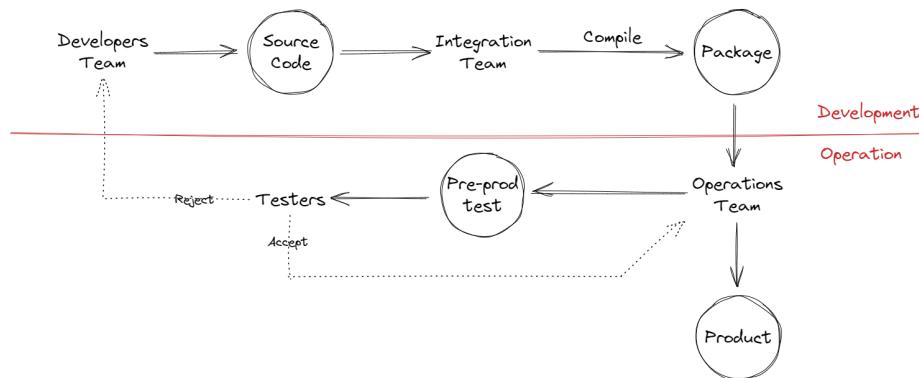


FIGURE B.4
CI of a new feature.

In practice, each cycle in Fig. E.4 can take a few days or even weeks as so many teams have to cooperate to make it happen. It takes time for the integration team to integrate different branches in the source code together, making sure that components from different branches function properly. When there is a defect, the flaws can be spotted only in the last stage of the iteration, i.e., testing. This practically disallows very frequent update of the system in response to the rapid changes in requirements.

CI (together with CD which is introduced in later section) tries to solve the above problems. CI automates the “development” portion in Fig. E.4, while CD automates the “operation” portion.

To speed up the development of new features, CI mainly adopts the following methods.

- Use *Git* to manage features. This simplifies the procedure of managing

multiple under-development features and integrating them together. Integration is now managed by *Git* following developers' intention.

- Use a build server to automatically compile the code into ready-for-delivery packages. The scrum master and senior developers can access the server and monitor the progression. Should there be a compiling error, the developer is notified immediately.
- The code, after compiling, is immediately tested in the build server using pre-defined test cases. If the code fails the test cases, the developer is immediately notified.

CI effectively removed “integration team” from the picture. The integration related tasks are split into small pieces and processed by automation tools, feature integration by *Git*, and compiling by build server. During CI, the developer not only generates the codes, but also supervises *Git* and build server. Should there be any error, the developer is notified immediately by the automation tools.

CI speeds up the developing cycle, from the receiver receiving requests from the user, to the point they have new system in the package ready for pre-prod testing.

B.2.3 Continuous Delivery

The package received from the development side contains the latest version of the system where a new feature is integrated. It is, however, not certain at this point whether the new feature works properly and how the system would behave as a whole with this new feature. The sophisticated testing and deployment of the software features are done by the operations team.

Conventionally, operations team and the testers receive the updated package together with an instruction from the developers team. The instruction describes how the package shall be installed, and what test cases to use for auditing. The operations team and the testers need to understand the instructions, and configure the pre-prod environment accordingly for the testing. The testers then uses varieties of scenarios to test the performance of the software. Bugs, if any, are reported to the developers team. If no bugs are spotted, the testers notify the operation team to release the package into production environment.

There are some obvious drawbacks to the conventional approach. The developers team needs to give detailed and precise instruction to the operations team, and the developers may make mistakes or missing something in the instruction, especially regarding environment configuration. Besides, there are too many human interactions, which slow down the process and generates human error. The entire procedure usually take about a whole day.

CD is a software development practice that allows software to be released to production at any time. The idea behind CD is to deploy the code for

testing automatically anytime CI provides a new package by adopting the following methods.

- Use machine-readable instruction files for packages installation, and let the server virtualize the execution environment and install the packages automatically.
- Use machine-readable testing scripts, and let the server execute tests and analyze the results automatically.
- The aforementioned machine-readable instruction files and testing scripts are managed the same way as the source code by the developers.

Ideally, as soon as a version of packages is released by CI, CD can automatically have it deployed and tested, and return the testing results to CI without human interaction. This is shown by Fig. E.5. In this CI/CD implementation, the developer is playing a more comprehensive role than what is shown in Fig. E.4.

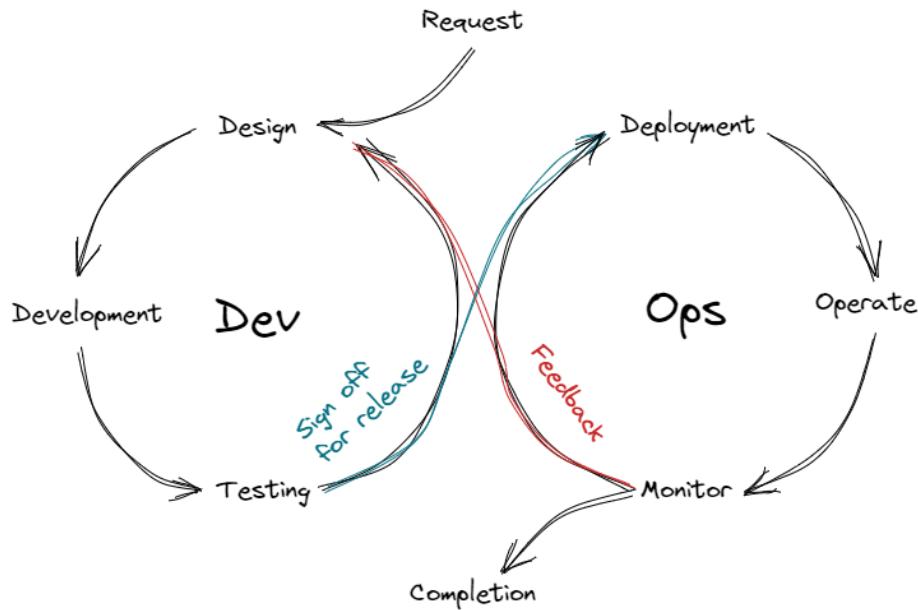


FIGURE B.5
CI and CD.

Under the framework of CI/CD in Fig. E.5, the developer not only develops the new feature, but also integrates the features with the help of version control and branch management tool *Git*, compiles the packages using build server, deploys the new packages in the testing environment by preparing

machine-readable instruction files, and finally tests the new features using pre-configured test cases in the machine-readable testing scripts.

Since the operation team joins the developers team, they are now called the DevOps team. The CI/CD framework shown in Fig. E.5 is known as the CI/CD pipeline which represents an end-to-end software development lifecycle (SDLC) within its ecosystem.

This CI/CD framework enables fast deployment of new packages. Large IT companies can make up to dozens of new releases everyday.

B.3 GitHub Action (Part I: Basics)

GitHub has been an amazing platform for managing software projects, especially open-source collaborative projects.

In the early days when CI/CD was not enabled in *GitHub*, developers used third-party CI/CD tools such as Jenkins and Travis CI in conjunction with *GitHub* for automatic integration and deployment. Lately, *GitHub* introduced *GitHub Actions* (for short, *Actions*), its own CI/CD solution, as a response to developers' requests.

Actions is essentially a pipeline tool, and it is very useful as part of the practice of CI/CD on GitHub. In short, it allows the developers to create automatic workflows tied to *GitHub* events. *Actions* is widely used to automate the following tasks:

1. Compile the source code. The developer can define how the source code should be compiled.
2. Virtualize the environment to run the compiled packages. The developer can define how to set up the environment.
3. Test the packages on testing scenarios. The developer can define the testing scenarios.
4. Raise an issue if anything occurs during the above procedures.

Machine-readable instruction files are used to guide the automation. They are managed together with the source code in the repository.

Notice that *Actions* might be chargeable for private repositories depending on its computational cost. It is, however, often free of charge for community and public repository projects.

A commonly used scenarios of *Actions* for community projects is that when someone submits a pull request, a series of checks are automatically done, and emails are sent to the project maintenance team to notify them the request. Sometimes a “thank you” email is sent to the contributor.

Pull Request

When an individual developer wants to contribute to an open-source community project of others, the following is the general flow.

1. The developer forks the project to his own *GitHub* account.
2. The developer clones a copy of the project from his own *GitHub* account to his own machine.
3. The developer creates a new branch about the feature he wants to add or improve.
4. The developer develops the feature on the branch.
5. The developer commits the development and pushes the commit to the forked repository.
6. The developer submits a pull request to the maintenance team of the community project.
7. The maintenance team receives the pull request and review the changes made to the code in the developer's forked repository.
8. If no objection, the maintenance team pull the changes from the developer's forked repository, and merge the new feature branch with the existing branches.

With the above, the developer becomes an official contributor to the community project.

Another use case for *Actions* for individual projects is that when a new code is pushed to the repository, the code is automatically compiled and tested. If the test returns an error, create a *GitHub* issue. Otherwise, put out a new release.

B.3.1 Framework

In *Actions*, the developer needs to prepare CI/CD pipelines known as workflows, either by himself or from *Actions* Marketplace (a platform where commonly used workflows are shared). The backbone of a workflow is YAML files that describe the environment, the trigger, and the list of actions to execute. When an event such as a pull request happens, the associated workflow is triggered and executed. A demonstrative plot is given in Fig. E.6.

As illustrated in Fig. E.6, a workflow can contain multiple jobs that can run in parallel (by default) or sequence (by defining dependencies). Each job can contain multiple steps that run in sequence.

For each job, a “runner” is assigned which serves as the environment to run

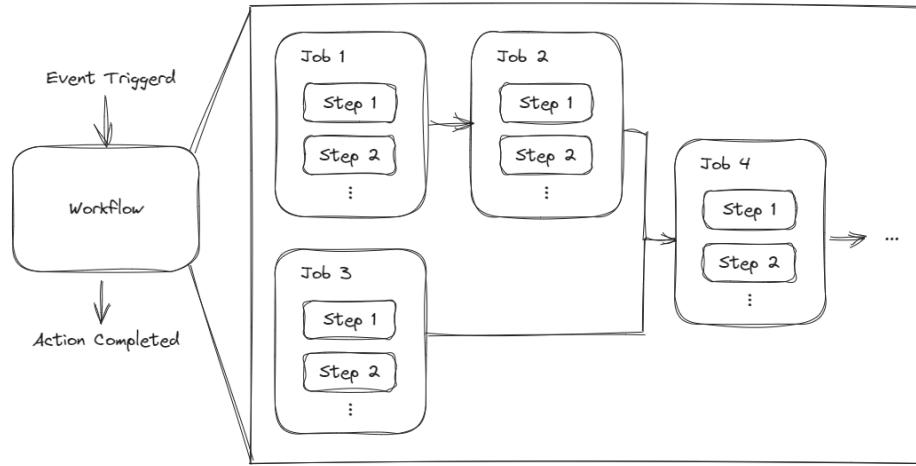


FIGURE B.6
Actions framework.

the job. The developer needs to specify how the runner should be configured such as what software needs to be installed in order to run the associated job.

B.3.2 *Actions* Triggers and Types

It is worth mentioning that *Actions* can be triggered not only by *Github* events but also external events, schedules, etc. Commonly seen triggers include

- *Github* Events, such as push, pull request, etc. This is the most common setup for CI/CD.
- External events.
- Schedule.
- Manual.

Actions provides workflow templates of different task types. There are at least the following workflow types as of this writing. They are not limited to CI/CD.

- CI
- Deployment
- Testing
- Code quality
- Code review

- Dependency management
- Monitoring
- Automation
- Utilities
- Pages
- Hugo

Some of them will be covered in later sections.

B.3.3 *Actions* Marketplace

Actions marketplace provides commonly seen actions for free, and they can be easily integrated into a workflow. The developer can look for action items in *Actions* marketplace before trying to prepare the entire workflow by himself from scratch.

B.3.4 Costs

Actions may generates additional costs. This is because executing workflows usually consumes *GitHub* servers' storage and CPU. As of this writing, additional cost is generated only if all the following criteria are met:

- Private repository. This is often not the case for community and individual projects, but often true for enterprise repositories.
- The workflows are executed on *GitHub* servers. Notice that it is possible to execute workflows using on-premises servers, in which case *Actions* becomes a free service.
- Storage and/or computational cost goes beyond the free-tier threshold.

When all the criteria is met, the developer pays *GitHub* by the storage and computational consumption. He can select either prepaid mode (fixed bill, limited consumption per month) or postpaid mode (unfixed bill, unlimited consumption per month).

The running environment also affects the cost. It is often more expensive if the workflow needs to run on Windows or MacOS than if it can run on Linux due to the licensing fee. As of this writing, Windows and MacOS introduce a computational cost multiplier of 2 and 10 respectively comparing with Linux.

B.4 GitHub Actions (Part II: Practice)

As introduced in the previous section, the key to an *Actions* includes defining the triggers and preparing the associated workflows for the triggers. Of course, the developer also needs to map the triggers with the workflows. More details are introduced in this section.

The workflow files are used to define the triggers as well as the associated workflows. They are YAML format files following specific syntax that illustrates environment configurations requirements, jobs, and job steps. They are stored under `.github/workflows` and are managed by the developers just like any other source code. Notice that there are certain events that would trigger only if their associated workflow files exist in the project repository default branch (usually `main` or `master` branch).

As an example, here is a piece of workflow file from *GitHub* document.

```
name: GitHub Actions Demo
run-name: ${{ github.actor }} is testing out GitHub Actions
on: [push]
jobs:
  Explore-GitHub-Actions:
    runs-on: ubuntu-latest
    steps:
      - run: echo "The job was automatically triggered by a ${{ github.event_name }} event."
      - run: echo "This job is now running on a ${{ runner.os }} server hosted by GitHub!"
      - run: echo "The name of your branch is ${{ github.ref }} and your repository is ${{ github.repository }}."
      - name: Check out repository code
        uses: actions/checkout@v4
      - run: echo "The ${{ github.repository }} repository has been cloned to the runner."
      - run: echo "The workflow is now ready to test your code on the runner."
      - name: List files in the repository
        run: |
          ls ${{ github.workspace }}
      - run: echo "This job's status is ${{ job.status }}."
```

YAML and its syntax are introduced in more details in Appendix F.

B.4.1 Trigger

It is most common that an event triggers a workflow. An event can be defined in several ways such as

- A person or a process does some operations to the *GitHub* repository, such as submitting a pull request or pushing a new commit to the repository.
- Something happened outside *GitHub* which serves as an external trigger that triggers a workflow.
- A schedule that runs a workflow at particular timestamps or periodically.
- Manually starting a workflow.

In a workflow YAML file, the trigger is defined by keyword `on`. For example, one may see

```
on:
  push:
    <...>
```

or

```
on:
  pull_request:
    <...>
```

or

```
on:
  scheduled:
    <...>
```

which define different trigger types. It is possible to include multiple trigger types in the statement using a list, such as

```
on:
  [push, pull_request]:
    <...>
```

B.4.2 Workflow

A workflow contains multiple jobs that run in parallel or in sequence. A job contains multiple steps executed in sequence. Each step is something like a line of command in shell.

Each job is associated with a “runner” which is a physical or a virtual computer or container that execute the job. All the steps defined under a job need to run in the same environment. Of course, each step can call a different software program.

The following is an example given by *GitHub* starter workflows template. Upon pushing a new commit, the workflow create a *ubuntu* environment, install conda, python and associated packages, and finally run `pytest` to check the code quality.

```
name: Python Package using Conda
on: [push]
jobs:
  build-linux:
    runs-on: ubuntu-latest
    strategy:
      max-parallel: 5
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python 3.10
        uses: actions/setup-python@v3
        with:
          python-version: '3.10'
      - name: Add conda to system path
        run: |
          # $CONDA is an environment variable pointing to the root of the
          # miniconda directory
          echo $CONDA/bin >> $GITHUB_PATH
      - name: Install dependencies
        run: |
          conda env update --file environment.yml --name base
      - name: Lint with flake8
        run: |
          conda install flake8
          # stop the build if there are Python syntax errors or undefined
          # names
          flake8 . --count --select=E9,F63,F7,F82 --show-source --
          statistics
          # exit-zero treats all errors as warnings. The GitHub editor is
          # 127 chars wide
          flake8 . --count --exit-zero --max-complexity=10 --max-line-
          length=127 --statistics
      - name: Test with pytest
        run: |
          conda install pytest
          pytest
```

This workflow example defines one job, `build-linux`. The job defines five steps, `Set up Python 3.10`, `Add conda to system path`, `Install dependencies`, `Lint with flake8` and `Test with pytest`. Each step is associated with one `run` that executes a command in the shell, for example,

```
name: Install dependencies
run: conda env update --file environment.yml --name base
```

executes a `conda` command that updates the packages defined in `environment.yml` in the base environment.

In this example, it can be seen that many things besides the commands are configured in the workflow, such as the name of the jobs and steps, the OS, the *Actions* reference, the software version, etc.

Notice that `uses: actions/checkout@v3` as the first step in the list of steps pulls in a predefined action defined at `github.com/actions/checkout`. The domain name `github.com/` can be neglected.

Further details can be configured that are not shown in this example. Just to give one example, by using

```
on:
  push:
    branches:
      - main
      - branch-1
      - branch-2
```

it can be specified which branch(es) to trigger the workflow when push a commit.

B.4.3 Execution

The workflow file needs to be saved under `.github/workflows/` in the repository, with an arbitrary name but it has to end with suffix `.yml` or `.yaml`.

Push the repository with the workflow file to *GitHub*. On *GitHub* repository dashboard under “Actions”, select and enable the action so that the workflow would be triggered once the trigger event occurs.

B.4.4 Very Complicated Actions

Some *Actions* workflows can be as simple as a few lines of shell commands, for example, [GitHub Action Demo](#) given on page 363. Others, however, can be extremely complicated that includes a large set of implementation code, test cases, building, checking for vulnerabilities, packaging, etc. An example of such complicated *Actions* flow is `actions/checkout@v3` which was used in the other example [Python Package using Conda](#) given in page 364.

More details of the action can be found at `github.com/actions/checkout`. As of this writing, the latest version of the code is v4.1.0, and in its repository it has a long list of files in TypeScript (an enhanced superset of JavaScript), YAML, JSON, shell scripts and other languages, and it is still under development. A screenshot is given in Fig. E.7. Like other actions, it stores the workflow files (for this action, there are many of them) under `.github/workflows`. The scripts that does all the actual jobs, in this action, are written in TypeScript and are stored under `src`. Supporting materials such as license, reference to contributors, etc., are also included.

To package such a complicated action, there must be an `action.yml` or `action.yaml` that contains metadata of the action. This file specifies the inputs, outputs and configurations of the action.

It would be too detailed to go through all the files in `github.com/actions/checkout`. The message here is that an action can be designed to be very complicated and powerful.

 .github/workflows	Add support for partial checkout filters (#1396)	3 weeks ago
 .licenses/npm	Improve checkout performance on Windows runners by upgrading @ac...	6 months ago
 __test__	Add support for partial checkout filters (#1396)	3 weeks ago
 adrs	Fix typos found by codespell (#1287)	6 months ago
 dist	Add support for partial checkout filters (#1396)	3 weeks ago
 src	Add support for partial checkout filters (#1396)	3 weeks ago
 .eslintignore	Convert checkout to a regular action (#70)	4 years ago
 .eslintrc.json	update dev dependencies and react to new linting rules (#611)	2 years ago
 .gitattributes	Add Licensed To Help Verify Prod Licenses (#326)	3 years ago
 .gitignore	Inject GitHub host to be able to clone from another GitHub instance (#...)	last year
 .licensed.yml	Add Licensed To Help Verify Prod Licenses (#326)	3 years ago
 .prettierignore	Convert checkout to a regular action (#70)	4 years ago
 .prettierrc.json	Convert checkout to a regular action (#70)	4 years ago
 CHANGELOG.md	Prepare 4.1.0 release (#1496)	3 weeks ago
 CODEOWNERS	Update CODEOWNERS to Launch team (#1510)	last week
 CONTRIBUTING.md	Replace datadog/squid with ubuntu/squid Docker image (#1002)	last year
 LICENSE	Add docs (#2)	4 years ago
 README.md	Correct link to GitHub Docs (#1511)	3 days ago
 action.yml	Add support for partial checkout filters (#1396)	3 weeks ago
 jest.config.js	Convert checkout to a regular action (#70)	4 years ago
 package-lock.json	Prepare 4.1.0 release (#1496)	3 weeks ago
 package.json	Prepare 4.1.0 release (#1496)	3 weeks ago
 tsconfig.json	update dev dependencies and react to new linting rules (#611)	2 years ago

FIGURE B.7
CI and CD.



C

A Brief Introduction to YAML

CONTENTS

C.1	Overview	343
C.2	Syntax	344
C.2.1	Key-Value Pair	344
C.2.2	Object and Nested Object	344
C.2.3	List	345
C.2.4	Multi-line String	346
C.3	Commonly Seen YAML Use Cases	347

YAML is widely used as configuration files and workflow description files. Under the scope of this notebook, YAML is used at least in *GitHub Actions* and *Kubernetes*. A brief introduction to YAML and its syntax is given here.

C.1 Overview

YAML is a human-readable data serialization language widely used for writing configuration files. The name YAML was initially interpreted as “Yet Another Markup Language” because the motivation for the authors to develop YAML was to simplify XML. However, in the later stage the authors pointed out that YAML is more of a data serialization language (like JSON) than a markup language. Hence, it is now more often known as the recursive acronym “YAML Ain’t Markup Language”.

Markup VS Serialization Languages

Markup languages focus on the marking up of various elements in a text document. In the early days, the editor of a book often needed to put marks on the manuscripts to show where each line should go, etc., before sending it to the publisher. These marks inspired markup languages, and hence the name.

Data serialization languages, on the other hand, focus on using texts

to represent data structures. Data serialization languages like JSON and YAML are able represent “objects” such as Python dictionaries, JavaScript objects, etc., using its textual syntax. Data serialization languages are useful as they maintain the data structures, allowing a machine to decode easily. Therefore, they are widely used in configuration files, data storage, and machine-to-machine data transfer.

It is possible that markup and serialization languages overlaps in some applications. For example, XML, a typical markup language, can also be used to serialize data.

C.2 Syntax

Commonly used YAML file extensions include `.yaml` and `.yml`. When learning YAML syntax, it is helpful to compare it side-by-side with JSON, as both of them are serialization languages and can be translated from one to the other.

Like Python, YAML uses line separation and indentation as part of its syntax. This makes it reader friendly.

YAML uses `#` to lead a comment. YAML is case-sensitive. Use `---` in a new line to separate a single YAML file into multiple logical sectors.

C.2.1 Key-Value Pair

Key-value pair is the most basic syntax in YAML as follows.

```
<key>: <value>
```

For example,

```
name: myApp
port: 9000
version: 1.1
tested: true
```

YAML is able to automatically interpret the data types of different variables. In the above example, for instance, port number 9000 is identified as an integer, while application name `myApp`, a string. To enforce it to interpret values as strings, use quotation marks. Notice that `true/yes/on` and their associated `false/no/off` are regarded as boolean values.

C.2.2 Object and Nested Object

Use indentation to indicate object trees. See the example below. Object can be nested.

```
<object name>:
```

```
<key1>: <value1>
<key2>: <value2>
<key3>:
  <key31>: <value31>
  <key32>: <value32>
```

where object `object name` contains 3 key-value pairs. The third key `key3` is associated with a value who is a nested object that contains another 2 key-value pairs.

C.2.3 List

It is possible that the value of a key being a list of items. See the example below. Be careful with the indentation when items in the list are nested objects.

```
<list1>:
  - <item1>
  - <item2>
  - <item3>
<list2>:
  - <item1 key>: <item1 value>
  - <item2 key>: <item2 value>
  - <item3 key>: <item3 value>
<list3>:
  - <item1 key1>: <item1 value1>
    <item1 key2>: <item1 value2>
    <item1 key3>: <item1 value3>
  - <item2 key1>: <item2 value1>
    <item2 key1>: <item2 value2>
    <item2 key1>: <item2 value3>
  - <item3 key1>: <item3 value1>
    <item3 key1>: <item3 value2>
    <item3 key1>: <item3 value3>
```

In the example above, the value of `<list1>` is simply a list of 3 items, `<item1>`, `<item2>`, `<item3>`. The value of `<list2>` is a list of 3 key-value pairs in the form `<itemN key>: <itemN value>`. The value of `<list3>` is a list of 3 objects, each object containing 3 key-value pairs, in the form of `<itemN keyM>: <itemN valueM>`.

Items in a list in YAML do not need to have the same data type or object structure.

For a list whose items are primitive data types, such as integer, float, boolean, string, etc., or a mix of them, it is also possible to use `[item1, item2, ...]`. For example, the following two expressions are equivalent.

```
port:
  - 9000
```

```
- 9001
- 9002
```

versus

```
port: [9000, 9001, 9002]
```

where the former and later expressions are known as the block style and flow style respectively.

It is possible to have a list of only one item. Examples are given below.

```
<list1>:
  - <item1>
<list2>: [<item2>]
<list3>:
  - <item3 key>: <item3 value>
<list4>:
  - <item4 key1>: <item4 value1>
    <item4 key2>: <item4 value2>
    <item4 key3>: <item4 value2>
```

In the above example, all 4 lists, `<list1>` to `<list4>`, have 1 items. The first two lists `<list1>` and `<list2>` have primitive items `<item1>` and `<item2>` respectively. List `<list3>` has one item which is a key-value pair. List `<list4>` has one item which is an object that contains 3 key-value pairs.

C.2.4 Multi-line String

Sometimes the value to be stored in YAML can be a long or multi-line string, for example, a command. Use `|` to indicate a multi-line string. An example is given below.

```
<key>: |
  this is the first line of a string
  this is the second line of a string
  this is the third line of a string
```

A practical example is given below.

```
run: |
  ls \
  -la \
  > files.txt
```

where a bash command `ls -la > files.txt` is stored as the value of key `run`.

Do not confuse a multi-line string with a wrapped single-line string. For example,

```
<key>: |
  abc
  def
  ghi
```

is different from

```
<key>: abcdefghi
```

This is because when | is used, YAML automatically adds line break characters between different rows.

To express the wrap of a single string, use > instead of | as follows.

```
<key>: >
  abc
  def
  ghi
```

which is equivalent with

```
<key>: abc def ghi
```

because > adds a space instead of a line break character to each row.

Notice that both | and >, if used alone, will generate a line breaker in the end of the entire text. If no such line breaker is designed, use |- and >- instead.

To express a very long continuous string without even a space, consider using

```
<key>: "abc\
  def\
  ghi"
```

which is equivalent to

```
<key>: abcdefghi
```

Sometimes for better readability, one may consider storing long strings into lists, then concatenate them later using the program that takes in the YAML file.

C.3 Commonly Seen YAML Use Cases

YAML has gained its popularity among configuration files, especially when containers, cloud service and CI/CD are involved. Some commonly seen services that support YAML are given below. This is only a small portion of all the services and programs that use YAML.

- *GitHub Actions* configuration files. *GitHub Actions* uses YAML as the workflow configuration files.
- *Kubernetes* configuration files. *Kubernetes* pods, images, services, deployments, etc., have to be configured by the developer using YAML.

- AWS CloudFormation. AWS CloudFormation is the manuscript using which AWS can start and configure a service automatically so that the user does not need to do everything using the dashboard. It is useful when the user needs to run similar and repetitive services. The AWS CloudFormation instruction file is written in YAML.
- Azure pipeline, and many other CI/CD services. Many CI/CD services uses YAML for the configuration of pipelines.

===== 9eeb46352eb2580dff8745d93d78e28721fb793c

D

Scripts

CONTENTS

D.1	<i>Vim</i> Configuration <code>vimrc</code> Used in Section 3	349
D.2	<i>Neovim</i> Configuration Files	350

This appendix chapter collects the example scripts used in this notebook.

D.1 *Vim* Configuration `vimrc` Used in Section 3

```
call plug#begin()
Plug 'vim-airline/vim-airline'
Plug 'joshdick/onedark.vim'
call plug#end()

inoremap jj <Esc>
noremap j h
noremap k j
noremap i k
noremap h i

noremap s <nop>
noremap S :w<CR>
noremap Q :q<CR>

syntax on
colorscheme onedark

set number
set cursorline
set wrap
set wildmenu

set hlsearch
exec "nohlsearch"
```

```
set incsearch
set ignorecase
noremap <Space> :nohlsearch<CR>
noremap - Nzz
noremap = nzz

noremap sj :set nosplitright<CR>:vsplit<CR>
noremap sl :set splitright<CR>:vsplit<CR>
noremap si :set nosplitbelow<CR>:split<CR>
noremap sk :set splitbelow<CR>:split<CR>
noremap <C-j> <C-w>h
noremap <C-l> <C-w>l
noremap <C-i> <C-w>k
noremap <C-k> <C-w>j
noremap J :vertical resize-2<CR>
noremap L :vertical resize+2<CR>
noremap I :res+2<CR>
noremap K :res-2<CR>

set scrolloff=3
noremap sc :set spell!<CR>
```

D.2 *Neovim* Configuration Files

Neovim is fork from *Vim* and it has gained popularity in the past years. Just like *Vim*, *Neovim* can be customized by user-defined configuration files. These configuration files are usually packed into *lua* file tree, where *lua* is a computer programming language that *Neovim* supports by nature.

E

Continuous Integration and Delivery

CONTENTS

E.1	Agile VS Waterfall	351
E.1.1	Waterfall	352
E.1.2	Agile	353
E.1.3	Roles in Agile-based Development	353
E.2	Continuous Integration Continuous Delivery	355
E.2.1	Pipeline	355
E.2.2	Continuous Integration	356
E.2.3	Continuous Delivery	357
E.3	<i>GitHub Actions</i> (Part I: Basics)	358
E.3.1	Framework	360
E.3.2	<i>Actions</i> Triggers and Types	361
E.3.3	<i>Actions</i> Marketplace	362
E.3.4	Costs	362
E.4	<i>GitHub Actions</i> (Part II: Practice)	362
E.4.1	Trigger	363
E.4.2	Workflow	364
E.4.3	Execution	366
E.4.4	Very Complicated Actions	366

This notebook is mainly about Linux. In this appendix chapter, the boundary of the notebook is slightly expanded to software development, which is quite often how Linux is used in practice.

Continuous integration and delivery (CI/CD) is both a philosophical concept and a bunch of technologies that speeds up the development, testing and deployment cycle of a software. It has become a common and beneficial practice for collaborative projects with rapid updates.

This chapter introduces CI/CD as well as tools to carry out CI/CD. In particular, GitHub, a widely appreciated CI/CD integrated platform to manage and share collaborative projects, is briefly introduced. The introduction of GitHub focuses on GitHub Actions, the tool GitHub uses for CI/CD.

Some contents of this chapter come from [12].

E.1 Agile VS Waterfall

Agile and waterfall are both project management methods. Both of them are introduced here, starting with the more conventional waterfall model then Agile.

E.1.1 Waterfall

Speaking of project proposal, development, testing and delivery cycle, it is fairly intuitive to follow the procedures below:

1. Understand requirements from the user.
2. Design the architecture of the solution.
3. Develop the solution.
4. Test the solution.
5. Deliver the solution and close the project.
6. (Follow-up) maintain the solution.

The philosophy behind waterfall, as its name indicates, is to “follow the procedures and do not turn back”. When a previous step is considered completed, it is completed and should not be revoked or revised. This is demonstrated by Fig. E.1.

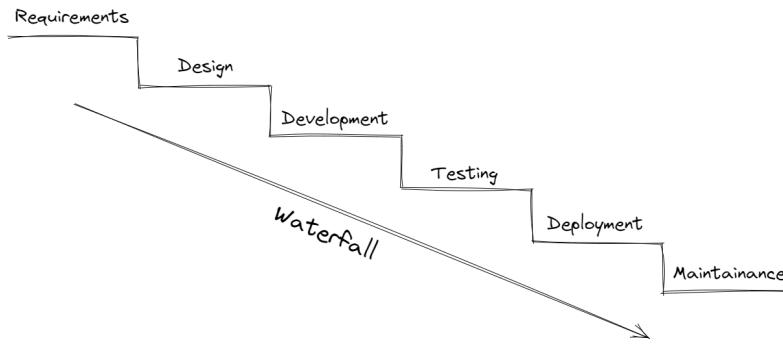


FIGURE E.1
Waterfall model.

With waterfall, a project can be designed, developed and deployed in a relatively efficient manner. However, there is one limitation: everything done in earlier steps cannot be changed in later steps. This sets up a high bar on

both the user and the developer. For example, in step 1 “understand requirements from the user”, the user needs to illustrate all the requirements to the developer as they cannot be modified in later steps. Similarly, in step 2 “design the architecture”, the developer needs to optimize the design to his best, as the architecture cannot be changed later.

Regrettably, with the rapid change in the market and the aggressive advent in technology today, it is challenging even for the smartest user and developer to determine all the requirements and designs in the beginning stage of a big project. More often, the requirements of the user have to change to align with the market trend, and so does the design of the solution.

For a new feature to be added into the existing system, it is possible to simply start a new project flow for the new feature, and integrate it into the existing system later. However, integrating the new feature into the existing system can also be challenging when the design is complicated and coupled. The integration often introduces a blackout period of the system. If the system is already deployed, the customer experience would be affected by the blackout.

E.1.2 Agile

Agile is the counterpart of waterfall. It is proposed to tackle the aforementioned issue: rapid change of requirements and adaptations to new technology and tools. It allows continuous integration and deployment of new features into the system in a convenient and systematic way, without introducing blackout.

In agile architecture, each feature is separately modularized. Each feature, before deploying and integrating into production environment, circulates in its own “development and testing circle”, where it can be tested and reviewed iteratively by the developers and the users audit team, as shown below in Fig. E.2.

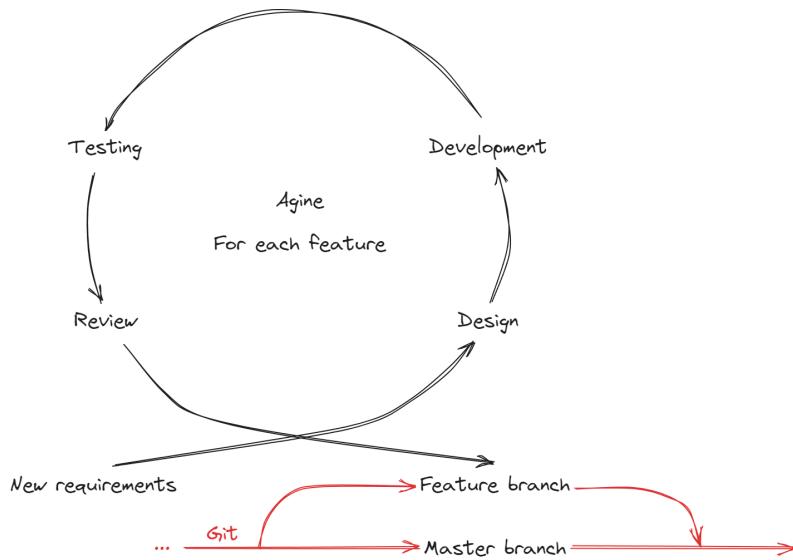
Agile allows rapid change to be made to the requirements and realizations of a feature. Should there be a change, just keep cycling in Fig. E.2 until the change is implemented and tested, before it is pushed back to the master branch.

In parallel development where there are multiple features running in their associated circles, the developer can easily choose which feature branch and circles to prioritize. This gives the developer a clearer overview of what is happening and how to best response to the customers immediate requirements.

E.1.3 Roles in Agile-based Development

The following roles are defined in agile, each role coming with a responsibility. The roles may slightly differ for each project. The most commonly seen approach, namely **scrum**, is introduced in Table. E.1.

In this role assignment, the product owner and scrum master come up with

**FIGURE E.2**

Agile model.

TABLE E.1
Roles in Agile model.

Role	Description
Product owner	Manage the entire program. He understands all the user requirements and progression of all features. He also signs off each feature when they are deployed.
Scrum master	Lead the developer team as team manager or chief developer.
Developer	Based on requirements, program the features.
Tester	Design test cases to verify the efficacy of the developed feature.
Operator/Supporter	Maintain the software

the product backlog, which clarify the sprints (tasks) and their priorities. The team then knows which sprints they shall work on first.

In the case where the features concerns a professional domain (such as economics, medical, etc.) that the developers cannot understand, business analysts are involved who bridge the user and the developer.

For each sprint, sprint planning and sprint backlog are proposed that describes the schedule of the sprint. The team works on the sprint and host daily scrum meetings until the sprint is solved. Upon finishing of a sprint, sprint review is hosted for audition.

E.2 Continuous Integration Continuous Delivery

CI/CD is introduced in this section. As a start, pipeline is introduced. Pipeline is an important concept in CI/CD.

E.2.1 Pipeline

Pipeline is a set of data processing elements in a queue, where the output of the upper stream process is the input of the down stream process. An example is shown in Fig. E.3. By using a pipeline and let multiple pipelines at different execution phase run in parallel, the efficiency of the system is increased.

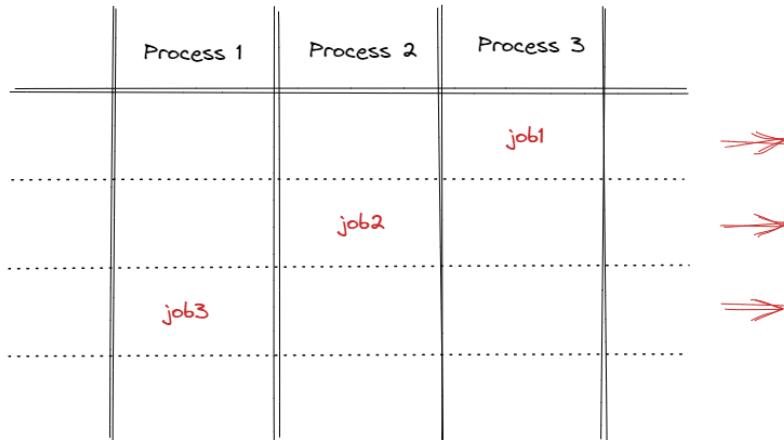


FIGURE E.3
Pipeline.

Pipeline is widely used in OS for process and thread management (recall Fig. 6.1). It is also used in CI/CD. In the context of CI/CD, think of different

jobs as features developed in parallel, and different processes the phases during the development including programming, integration, testing and deployment. In the remaining of this chapter, for simplicity, we assume only one job in the pipeline, i.e., only one feature is being developed. However, always have it in mind that multiple features can be developed simultaneously.

E.2.2 Continuous Integration

In the development of a sprint, new codes are rapidly developed, and they are rapidly compiled, packaged and tested.

Conventionally, the integration of a new feature requires involvement from multiple parties. An example is given in Fig. E.4. It includes the developer who program the software following users (or business analysts) requests, the integration team who integrates the new feature with the existing system and compiles the code into packages, and the operations team who uploads the new system into the pre-prod environment for real data testing, and the testers who audit the output of the program running in the pre-prod environment.

Should there be any error along the way, the code rolls back to the developer team for trouble shooting. When the new system with the updated code survives pre-prod environment, it is then pushed to the production environment.

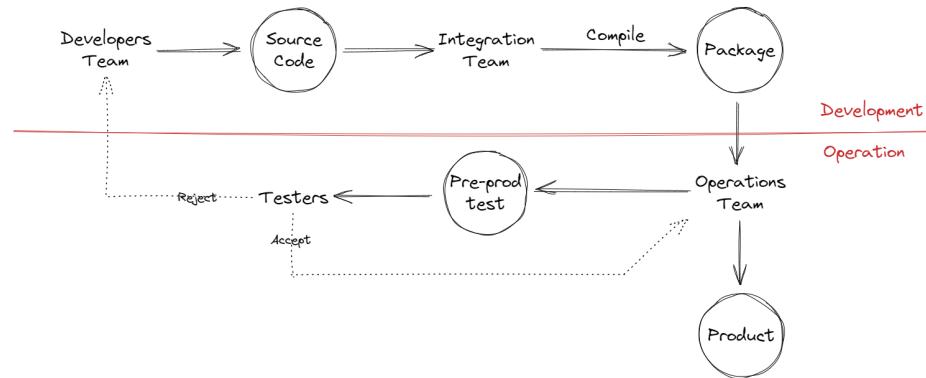


FIGURE E.4

Integration and deployment of a new feature in a conventional manner.

In practice, each cycle in Fig. E.4 can take a few days or even weeks as so many teams have to cooperate to make it happen. It takes time for the integration team to integrate different branches in the source code together, making sure that components from different branches function properly. When there is a defect, the flaws can be spotted only in the last stage of the iteration, i.e., testing. This practically disallows very frequent update of the system in response to the rapid changes in requirements.

CI (together with CD, which is introduced in a later section) addresses the challenges outlined previously. CI automates the “development” portion shown in Fig. E.4, while CD automates the “operation” portion.

To expedite the development of new features, CI primarily employs the following strategies:

- Utilize *Git* for version control, managing code branches that reflect different features or development efforts. This streamlines the integration process, allowing developers to merge features.
- Employ a build server to compile the code into delivery-ready packages. The scrum master and senior developers can access this server to monitor progress. If a compilation or test error occurs, the developer is immediately notified through preferred communication channels.
- After compilation, the code is rigorously tested on the build server using predefined test cases. Failing these tests triggers immediate notifications to developers. This testing phase is distinct from the pre-production environment tests that are part of CD, which are generally more comprehensive and may include performance and security assessments.

CI effectively removed “integration team” from the picture. The integration related tasks are split into small pieces and processed (almost) automatically with the help of *Git* and the build server. During CI, the developers program the software, use *Git* to manage branches and merge features and guide the build server to compile and test the code. Should there be any error, the developer is notified immediately.

E.2.3 Continuous Delivery

The packages received from CI already passed the tests in the build server. These tests mainly focus on the code quality. At this point of time, it is not clear yet how the system with the new packages would perform in the realistic environment. The major task for the operation team, as illustrated by Fig. E.4, is to deploy the system in the pre-production environment and have it tested with more complex and realistic scenarios, ensuring that the new packages work properly as part of the whole system.

Conventionally, operations team and the testers receive the updated package together with an instruction from the developers team. The instruction describes how the package shall be installed, and what test cases should be arranged in the pre-production environment. The operations team and the testers need to understand the instructions, and configure the pre-prod environment accordingly for the testing. The testers then uses varieties of scenarios to test the performance of the software. Bugs, if any, are reported to the developers team. If no bugs are spotted, the testers notify the operation team to release the package into the production environment.

There are some obvious drawbacks to the conventional approach. The developers team needs to give detailed and precise instruction to the operations team, and the developers may make mistakes or missing something in the instruction, especially regarding environment configuration. Besides, there are too many human interactions, which slow down the process and generates human error. The entire procedure usually take about a whole day.

CD is a software development practice that allows software to be released to production at any time. The idea behind CD is to deploy the code for testing automatically anytime CI provides a new package by adopting the following methods.

- Use machine-readable instruction files for packages installation, and let the server virtualize the pre-production environment and install the packages automatically.
- Use machine-readable testing scripts, and let the server execute tests and analyze the results automatically.
- The aforementioned machine-readable instruction files and testing scripts are managed the same way as the source code by the developers.

Ideally, as soon as a version of packages is released by CI, CD can automatically have it deployed and tested, and return the testing results to CI without human interaction. This is shown by Fig. E.5. In this CI/CD implementation, the developer is playing a more comprehensive role than that in Fig. E.4.

Under the framework of CI/CD in Fig. E.5, developers not only program the code to include new features but also integrate the features with the help of version control and branch management tools like Git. They compile the packages using the build server, deploy the new packages in the testing environment by preparing machine-readable instruction files, and finally test the new features using pre-configured test cases in machine-readable testing scripts. Developers now take on the roles of the original developers, the integration team, the operations team, and the testers. They are referred to as the DevOps team to reflect this change. The CI/CD framework shown in Fig. E.5 is known as the CI/CD pipeline, which represents an end-to-end software development lifecycle (SDLC) within its ecosystem.

This CI/CD framework enables fast deployment of new packages. Large IT companies can make up to dozens of new releases everyday.

E.3 *GitHub Actions* (Part I: Basics)

GitHub has been an amazing platform for managing software projects, especially open-source collaborative projects.

In the early days when CI/CD was not enabled in *GitHub*, developers used

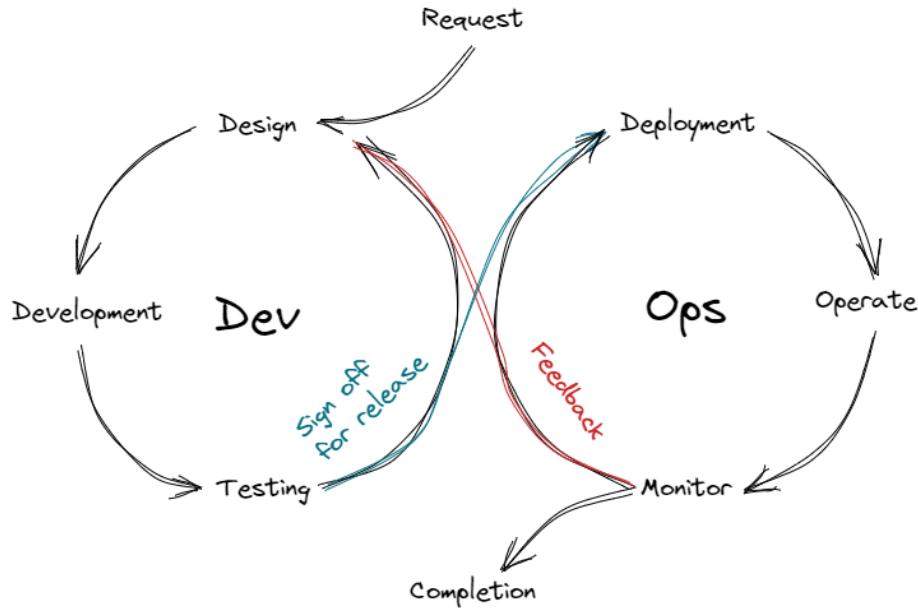


FIGURE E.5
CI and CD.

third-party CI/CD tools such as Jenkins and Travis CI in conjunction with *GitHub* for automatic integration and deployment. Lately, *GitHub* introduced *GitHub Actions* (for short, *Actions*), its own CI/CD solution, as a response to developers' requests.

Actions is essentially a pipeline tool, and it is very useful as part of the practice of CI/CD on GitHub. In short, it allows the developers to create automatic workflows tied to *GitHub* events. *Actions* is widely used to automate the following tasks:

1. Compile the source code. The developer can define how the source code should be compiled.
2. Virtualize the environment to run the compiled packages. The developer can define how to set up the environment.
3. Test the packages on testing scenarios. The developer can define the testing scenarios.
4. Raise an issue if anything occurs during the above procedures.

Machine-readable instruction files are used to guide the automation. They are managed together with the source code in the repository.

Notice that *Actions* might be chargeable for private repositories depending

on its computational cost. It is, however, often free of charge for community and public repository projects.

A commonly used scenarios of *Actions* for community projects is that when someone submits a pull request, a series of checks are automatically done, and emails are sent to the project maintenance team to notify them the request. Sometimes a “thank you” email is sent to the contributor.

Pull Request

When an individual developer wants to contribute to an open-source community project of others, the following is the general flow.

1. The developer forks the project to his own *GitHub* account.
2. The developer clones a copy of the project from his own *GitHub* account to his own machine.
3. The developer creates a new branch about the feature he wants to add or improve.
4. The developer develops the feature on the branch.
5. The developer commits the development and pushes the commit to the forked repository.
6. The developer submits a pull request to the maintenance team of the community project.
7. The maintenance team receives the pull request and review the changes made to the code in the developer’s forked repository.
8. If no objection, the maintenance team pull the changes from the developer’s forked repository, and merge the new feature branch with the existing branches.

With the above, the developer becomes an official contributor to the community project.

Another use case for *Actions* for individual projects is that when a new code is pushed to the repository, the code is automatically compiled and tested. If the test returns an error, create a *GitHub* issue. Otherwise, put out a new release.

E.3.1 Framework

In *Actions*, the developer needs to prepare CI/CD pipelines known as workflows, either by himself or from *Actions* Marketplace (a platform where commonly used workflows are shared). The backbone of a workflow is YAML files that describe the environment, the trigger, and the list of actions to execute.

When an event such as a pull request happens, the associated workflow is triggered and executed. A demonstrative plot is given in Fig. E.6.

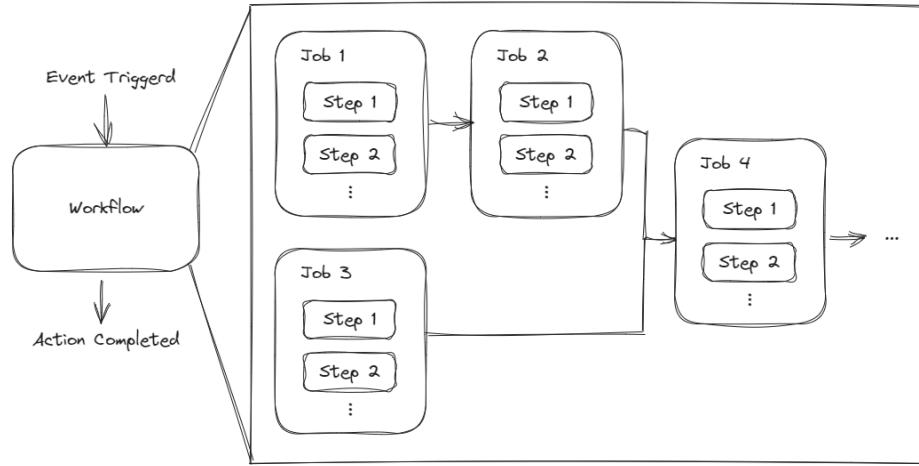


FIGURE E.6
Actions framework.

As illustrated in Fig. E.6, a workflow can contain multiple jobs that can run in parallel (by default) or sequence (by defining dependencies). Each job can contain multiple steps that run in sequence.

For each job, a “runner” is assigned which serves as the environment to run the job. The developer needs to specify how the runner should be configured such as what software needs to be installed in order to run the associated job.

E.3.2 *Actions* Triggers and Types

It is worth mentioning that *Actions* can be triggered not only by *Github* events but also external events, schedules, etc. Commonly seen triggers include

- *Github* Events, such as push, pull request, etc. This is the most common setup for CI/CD.
- External events.
- Schedule.
- Manual.

Actions provides workflow templates of different task types. There are at least the following workflow types as of this writing. They are not limited to CI/CD.

- CI

- Deployment
- Testing
- Code quality
- Code review
- Dependency management
- Monitoring
- Automation
- Utilities
- Pages
- Hugo

Some of them will be covered in later sections.

E.3.3 *Actions* Marketplace

Actions marketplace provides commonly seen actions for free, and they can be easily integrated into a workflow. The developer can look for action items in *Actions* marketplace before trying to prepare the entire workflow by himself from scratch.

E.3.4 Costs

Actions may generates additional costs. This is because executing workflows usually consumes *Github* servers' storage and CPU. As of this writing, additional cost is generated only if all the following criteria are met:

- Private repository. This is often not the case for community and individual projects, but often true for enterprise repositories.
- The workflows are executed on *Github* servers. Notice that it is possible to execute workflows using on-premises servers, in which case *Actions* becomes a free service.
- Storage and/or computational cost goes beyond the free-tier threshold.

When all the criteria is met, the developer pays *Github* by the storage and computational consumption. He can select either prepaid mode (fixed bill, limited consumption per month) or postpaid mode (unfixed bill, unlimited consumption per month).

The running environment also affects the cost. It is often more expensive if the workflow needs to run on Windows or MacOS than if it can run on Linux due to the licensing fee. As of this writing, Windows and MacOS introduce a computational cost multiplier of 2 and 10 respectively comparing with Linux.

E.4 GitHub Actions (Part II: Practice)

As introduced in the previous section, the key to an *Actions* includes defining the triggers and preparing the associated workflows for the triggers. Of course, the developer also needs to map the triggers with the workflows. More details are introduced in this section.

The workflow files are used to define the triggers as well as the associated workflows. They are YAML format files following specific syntax that illustrates environment configurations requirements, jobs, and job steps. They are stored under `.github/workflows` and are managed by the developers just like any other source code. Notice that there are certain events that would trigger only if their associated workflow files exist in the project repository default branch (usually `main` or `master` branch).

As an example, here is a piece of workflow file from *GitHub* document.

```
name: GitHub Actions Demo
run-name: ${{ github.actor }} is testing out GitHub Actions
on: [push]
jobs:
  Explore-GitHub-Actions:
    runs-on: ubuntu-latest
    steps:
      - run: echo "The job was automatically triggered by a ${{ github.event_name }} event."
      - run: echo "This job is now running on a ${{ runner.os }} server hosted by GitHub!"
      - run: echo "The name of your branch is ${{ github.ref }} and your repository is ${{ github.repository }}."
      - name: Check out repository code
        uses: actions/checkout@v4
      - run: echo "The ${{ github.repository }} repository has been cloned to the runner."
      - run: echo "The workflow is now ready to test your code on the runner."
      - name: List files in the repository
        run: |
          ls ${{ github.workspace }}
      - run: echo "This job's status is ${{ job.status }}."
```

YAML and its syntax are introduced in more details in Appendix F.

E.4.1 Trigger

It is most common that an event triggers a workflow. An event can be defined in several ways such as

- A person or a process does some operations to the *GitHub* repository, such as submitting a pull request or pushing a new commit to the repository.
- Something happened outside *GitHub* which serves as an external trigger that triggers a workflow.
- A schedule that runs a workflow at particular timestamps or periodically.
- Manually starting a workflow.

In a workflow YAML file, the trigger is defined by keyword `on`. For example, one may see

```
on:
  push:
    <...>
```

or

```
on:
  pull_request:
    <...>
```

or

```
on:
  scheduled:
    <...>
```

which define different trigger types. It is possible to include multiple trigger types in the statement using a list, such as

```
on:
  [push, pull_request]:
    <...>
```

E.4.2 Workflow

A workflow contains multiple jobs that run in parallel or in sequence. A job contains multiple steps executed in sequence. Each step is something like a line of command in shell.

Each job is associated with a “runner” which is a physical or a virtual computer or container that execute the job. All the steps defined under a job need to run in the same environment. Of course, each step can call a different software program.

The following is an example given by *GitHub* starter workflows template. Upon pushing a new commit, the workflow create a *ubuntu* environment, install conda, python and associated packages, and finally run `pytest` to check the code quality.

```
name: Python Package using Conda
on: [push]
jobs:
  build-linux:
    runs-on: ubuntu-latest
    strategy:
      max-parallel: 5
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python 3.10
        uses: actions/setup-python@v3
        with:
          python-version: '3.10'
      - name: Add conda to system path
        run: |
          # $CONDA is an environment variable pointing to the root of the
          # miniconda directory
          echo $CONDA/bin >> $GITHUB_PATH
      - name: Install dependencies
        run: |
          conda env update --file environment.yml --name base
      - name: Lint with flake8
        run: |
          conda install flake8
          # stop the build if there are Python syntax errors or undefined
          # names
          flake8 . --count --select=E9,F63,F7,F82 --show-source --
          statistics
          # exit-zero treats all errors as warnings. The GitHub editor is
          # 127 chars wide
          flake8 . --count --exit-zero --max-complexity=10 --max-line-
          length=127 --statistics
      - name: Test with pytest
        run: |
          conda install pytest
          pytest
```

This workflow example defines one job, `build-linux`. The job defines five steps, `Set up Python 3.10`, `Add conda to system path`, `Install dependencies`, `Lint with flake8` and `Test with pytest`. Each step is associated with one `run` that executes a command in the shell, for example,

```
name: Install dependencies
run: conda env update --file environment.yml --name base
```

executes a `conda` command that updates the packages defined in `environment.yml` in the base environment.

In this example, it can be seen that many things besides the commands are configured in the workflow, such as the name of the jobs and steps, the OS, the *Actions* reference, the software version, etc.

Notice that `uses: actions/checkout@v3` as the first step in the list of steps pulls in a predefined action defined at `github.com/actions/checkout`. The domain name `github.com/` can be neglected.

Further details can be configured that are not shown in this example. Just to give one example, by using

```
on:
  push:
    branches:
      - main
      - branch-1
      - branch-2
```

it can be specified which branch(es) to trigger the workflow when push a commit.

E.4.3 Execution

The workflow file needs to be saved under `.github/workflows/` in the repository, with an arbitrary name but it has to end with suffix `.yml` or `.yaml`.

Push the repository with the workflow file to *GitHub*. On *GitHub* repository dashboard under “Actions”, select and enable the action so that the workflow would be triggered once the trigger event occurs.

E.4.4 Very Complicated Actions

Some *Actions* workflows can be as simple as a few lines of shell commands, for example, `GitHub Action Demo` given on page 363. Others, however, can be extremely complicated that includes a large set of implementation code, test cases, building, checking for vulnerabilities, packaging, etc. An example of such complicated *Actions* flow is `actions/checkout@v3` which was used in the other example `Python Package using Conda` given in page 364.

More details of the action can be found at `github.com/actions/checkout`. As of this writing, the latest version of the code is v4.1.0, and in its repository it has a long list of files in TypeScript (an enhanced superset of JavaScript), YAML, JSON, shell scripts and other languages, and it is still under development. A screenshot is given in Fig. E.7. Like other actions, it stores the workflow files (for this action, there are many of them) under `.github/workflows`. The scripts that does all the actual jobs, in this action, are written in TypeScript and are stored under `src`. Supporting materials such as license, reference to contributors, etc., are also included.

To package such a complicated action, there must be an `action.yml` or `action.yaml` that contains metadata of the action. This file specifies the inputs, outputs and configurations of the action.

It would be too detailed to go through all the files in `github.com/actions/checkout`. The message here is that an action can be designed to be very complicated and powerful.

 .github/workflows	Add support for partial checkout filters (#1396)	3 weeks ago
 .licenses/npm	Improve checkout performance on Windows runners by upgrading @ac...	6 months ago
 __test__	Add support for partial checkout filters (#1396)	3 weeks ago
 adrs	Fix typos found by codespell (#1287)	6 months ago
 dist	Add support for partial checkout filters (#1396)	3 weeks ago
 src	Add support for partial checkout filters (#1396)	3 weeks ago
 .eslintignore	Convert checkout to a regular action (#70)	4 years ago
 .eslintrc.json	update dev dependencies and react to new linting rules (#611)	2 years ago
 .gitattributes	Add Licensed To Help Verify Prod Licenses (#326)	3 years ago
 .gitignore	Inject GitHub host to be able to clone from another GitHub instance (#...)	last year
 .licensed.yml	Add Licensed To Help Verify Prod Licenses (#326)	3 years ago
 .prettierignore	Convert checkout to a regular action (#70)	4 years ago
 .prettierrc.json	Convert checkout to a regular action (#70)	4 years ago
 CHANGELOG.md	Prepare 4.1.0 release (#1496)	3 weeks ago
 CODEOWNERS	Update CODEOWNERS to Launch team (#1510)	last week
 CONTRIBUTING.md	Replace datadog/squid with ubuntu/squid Docker image (#1002)	last year
 LICENSE	Add docs (#2)	4 years ago
 README.md	Correct link to GitHub Docs (#1511)	3 days ago
 action.yml	Add support for partial checkout filters (#1396)	3 weeks ago
 jest.config.js	Convert checkout to a regular action (#70)	4 years ago
 package-lock.json	Prepare 4.1.0 release (#1496)	3 weeks ago
 package.json	Prepare 4.1.0 release (#1496)	3 weeks ago
 tsconfig.json	update dev dependencies and react to new linting rules (#611)	2 years ago

FIGURE E.7
CI and CD.



F

A Brief Introduction to YAML

CONTENTS

F.1	Overview	369
F.2	Syntax	370
F.2.1	Key-Value Pair	370
F.2.2	Object and Nested Object	370
F.2.3	List	371
F.2.4	Multi-line String	372
F.3	Commonly Seen YAML Use Cases	373

YAML is widely used as configuration files and workflow description files. Under the scope of this notebook, YAML is used at least in *GitHub Actions* and *Kubernetes*. A brief introduction to YAML and its syntax is given here.

F.1 Overview

YAML is a human-readable data serialization language widely used for writing configuration files. The name YAML was initially interpreted as “Yet Another Markup Language” because the motivation for the authors to develop YAML was to simplify XML. However, in the later stage the authors pointed out that YAML is more of a data serialization language (like JSON) than a markup language. Hence, it is now more often known as the recursive acronym “YAML Ain’t Markup Language”.

Markup VS Serialization Languages

Markup languages focus on the marking up of various elements in a text document. In the early days, the editor of a book often needed to put marks on the manuscripts to show where each line should go, etc., before sending it to the publisher. These marks inspired markup languages, and hence the name.

Data serialization languages, on the other hand, focus on using texts

to represent data structures. Data serialization languages like JSON and YAML are able represent “objects” such as Python dictionaries, JavaScript objects, etc., using its textual syntax. Data serialization languages are useful as they maintain the data structures, allowing a machine to decode easily. Therefore, they are widely used in configuration files, data storage, and machine-to-machine data transfer.

It is possible that markup and serialization languages overlaps in some applications. For example, XML, a typical markup language, can also be used to serialize data.

F.2 Syntax

Commonly used YAML file extensions include `.yaml` and `.yml`. When learning YAML syntax, it is helpful to compare it side-by-side with JSON, as both of them are serialization languages and can be translated from one to the other.

Like Python, YAML uses line separation and indentation as part of its syntax. This makes it reader friendly.

YAML uses `#` to lead a comment. YAML is case-sensitive. Use `---` in a new line to separate a single YAML file into multiple logical sectors.

F.2.1 Key-Value Pair

Key-value pair is the most basic syntax in YAML as follows.

```
<key>: <value>
```

For example,

```
name: myApp
port: 9000
version: 1.1
tested: true
```

YAML is able to automatically interpret the data types of different variables. In the above example, for instance, port number 9000 is identified as an integer, while application name `myApp`, a string. To enforce it to interpret values as strings, use quotation marks. Notice that `true/yes/on` and their associated `false/no/off` are regarded as boolean values.

F.2.2 Object and Nested Object

Use indentation to indicate object trees. See the example below. Object can be nested.

```
<object name>:
```

```
<key1>: <value1>
<key2>: <value2>
<key3>:
  <key31>: <value31>
  <key32>: <value32>
```

where object `object name` contains 3 key-value pairs. The third key `key3` is associated with a value who is a nested object that contains another 2 key-value pairs.

F.2.3 List

It is possible that the value of a key being a list of items. See the example below. Be careful with the indentation when items in the list are nested objects.

```
<list1>:
  - <item1>
  - <item2>
  - <item3>
<list2>:
  - <item1 key>: <item1 value>
  - <item2 key>: <item2 value>
  - <item3 key>: <item3 value>
<list3>:
  - <item1 key1>: <item1 value1>
    <item1 key2>: <item1 value2>
    <item1 key3>: <item1 value3>
  - <item2 key1>: <item2 value1>
    <item2 key1>: <item2 value2>
    <item2 key1>: <item2 value3>
  - <item3 key1>: <item3 value1>
    <item3 key1>: <item3 value2>
    <item3 key1>: <item3 value3>
```

In the example above, the value of `<list1>` is simply a list of 3 items, `<item1>`, `<item2>`, `<item3>`. The value of `<list2>` is a list of 3 key-value pairs in the form `<itemN key>: <itemN value>`. The value of `<list3>` is a list of 3 objects, each object containing 3 key-value pairs, in the form of `<itemN keyM>: <itemN valueM>`.

Items in a list in YAML do not need to have the same data type or object structure.

For a list whose items are primitive data types, such as integer, float, boolean, string, etc., or a mix of them, it is also possible to use `[item1, item2, ...]`. For example, the following two expressions are equivalent.

```
port:
  - 9000
```

```
- 9001
- 9002
```

versus

```
port: [9000, 9001, 9002]
```

where the former and later expressions are known as the block style and flow style respectively.

It is possible to have a list of only one item. Examples are given below.

```
<list1>:
  - <item1>
<list2>: [<item2>]
<list3>:
  - <item3 key>: <item3 value>
<list4>:
  - <item4 key1>: <item4 value1>
    <item4 key2>: <item4 value2>
    <item4 key3>: <item4 value2>
```

In the above example, all 4 lists, `<list1>` to `<list4>`, have 1 items. The first two lists `<list1>` and `<list2>` have primitive items `<item1>` and `<item2>` respectively. List `<list3>` has one item which is a key-value pair. List `<list4>` has one item which is an object that contains 3 key-value pairs.

F.2.4 Multi-line String

Sometimes the value to be stored in YAML can be a long or multi-line string, for example, a command. Use `|` to indicate a multi-line string. An example is given below.

```
<key>: |
  this is the first line of a string
  this is the second line of a string
  this is the third line of a string
```

A practical example is given below.

```
run: |
  ls \
  -la \
  > files.txt
```

where a bash command `ls -la > files.txt` is stored as the value of key `run`.

Do not confuse a multi-line string with a wrapped single-line string. For example,

```
<key>: |
  abc
  def
  ghi
```

is different from

```
<key>: abcdefghi
```

This is because when | is used, YAML automatically adds line break characters between different rows.

To express the wrap of a single string, use > instead of | as follows.

```
<key>: >
  abc
  def
  ghi
```

which is equivalent with

```
<key>: abc def ghi
```

because > adds a space instead of a line break character to each row.

Notice that both | and >, if used alone, will generate a line breaker in the end of the entire text. If no such line breaker is designed, use |- and >- instead.

To express a very long continuous string without even a space, consider using

```
<key>: "abc\
  def\
  ghi"
```

which is equivalent to

```
<key>: abcdefghi
```

Sometimes for better readability, one may consider storing long strings into lists, then concatenate them later using the program that takes in the YAML file.

F.3 Commonly Seen YAML Use Cases

YAML has gained its popularity among configuration files, especially when containers, cloud service and CI/CD are involved. Some commonly seen services that support YAML are given below. This is only a small portion of all the services and programs that use YAML.

- *GitHub Actions* configuration files. *GitHub Actions* uses YAML as the workflow configuration files.
- *Kubernetes* configuration files. *Kubernetes* pods, images, services, deployments, etc., have to be configured by the developer using YAML.

- AWS CloudFormation. AWS CloudFormation is the manuscript using which AWS can start and configure a service automatically so that the user does not need to do everything using the dashboard. It is useful when the user needs to run similar and repetitive services. The AWS CloudFormation instruction file is written in YAML.
- Azure pipeline, and many other CI/CD services. Many CI/CD services uses YAML for the configuration of pipelines.

iiiiii HEAD ===== ɿɿɿɿɿɿ 9590561686dbab978e33ec2e9fc9dcc148803fe4
ɿɿɿɿɿɿ 9eeb46352eb2580dff8745d93d78e28721fb793c

Bibliography

- [1] M. Gasser, *Building a secure computer system*. Citeseer, 1988.
- [2] S. Kenlon, “10 ways to use the linux find command.” <https://www.redhat.com/sysadmin/linux-find-command>, 2022. Accessed: 2024-07-02.
- [3] R. S. Management, “Dnf, the next-generation replacement for yum.” https://dnf.readthedocs.io/en/latest/command_ref.html#dnf-command-reference, 2021. Accessed: 2024-07-12.
- [4] Debian, “Basics of the debian package management system.” <https://www.debian.org/doc/manuals/debian-faq/pkg-basics.en.html>, 2024. Accessed: 2024-07-12.
- [5] R. Hat, “Red hat enterprise linux 9: Managing, monitoring, and updating the kernel.” https://docs.redhat.com/en-us/documentation/red_hat_enterprise_linux/9/pdf/managing_monitoring_and_updating_the_kernel/Red_Hat_Enterprise_Linux-9-Managing_monitoring_and_updating_the_kernel-en-US.pdf, 2024. Accessed: 2024-07-08.
- [6] MongoDB, “Aggregation stages.” <https://www.mongodb.com/docs/manual/reference/operator/aggregation-pipeline/#aggregation-stages>, 2024. Accessed: 2024-06-26.
- [7] MongoDB, “Aggregation operators.” <https://www.mongodb.com/docs/manual/reference/operator/aggregation/#aggregation-operators>, 2024. Accessed: 2024-06-26.
- [8] Redis, “Connect with redis clients.” <https://redis.io/docs/latest/develop/connect/clients/>, 2024. Accessed: 2024-07-10.
- [9] S. Grinder, “Docker and kubernetes: The complete guide.” <https://www.udemy.com/course/docker-and-kubernetes-the-complete-guide/>, 2023. Accessed: 2023-06-15.
- [10] A. S. Foundation, “Projects directory.” <https://projects.apache.org/projects.html>, 2024. Accessed: 2024-07-17.
- [11] A. S. Foundation, “Docker image for httpd.” https://hub.docker.com/_/httpd, 2024. Accessed: 2024-07-17.

- [12] J. Honai, “CI/CD for beginners.” <https://nlbsg.udemy.com/course/ci-cd-devops/>, 2023. Accessed: 2023-10-10.

Index
