‹packt›



Linux for Data Engineers

Master Linux commands, file management, & Bash scripting to automate everyday data engineering tasks

BHAVESH BHADRICHA

Linux for Data Engineers

Master Linux commands, file management, and Bash scripting to automate everyday data engineering tasks

Bhavesh Bhadricha

‹packt›

Linux for Data Engineers

Copyright © 2026 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing or its dealers and distributors will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Sunith Shetty

Relationship Lead: Apeksha Shetty

Project Manager: Shashank Desai

Content Engineer: Tiksha Abhimanyu Lad

Technical Editor: Seemanjay Ameriya

Indexer: Manju Arasan

Production Designer: Vijay Kamble

Growth Lead: Ankur Mulasi

First published: November 2026

Production reference: 1301026

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-80849-333-1

www.packtpub.com

I would like to dedicate this book to my mother, father, and my little sister for their constant presence, patience, and endless support.

– Bhavesh Bhadricha

Contributors

About the author

Bhavesh Bhadricha is a senior data engineer, technical curriculum developer, and the founder of the online learning platforms SmartDataCamp and Projects Based Learning. He specializes in big data ecosystem processing using Apache Spark, Hadoop, MapReduce, and Apache Hive. Alongside his extensive experience configuring Linux environments, Bash shell scripting, and containerized deployments using Docker, Bhavesh is passionate about open-source business intelligence and generative AI frameworks. He actively builds comprehensive technical chapters, hands-on courses, and interview guides to help data professionals advance their skills.

My sincere thanks to the entire data engineering and open-source community for their shared knowledge and support. I also want to thank the team at Packt for their collaboration in bringing this book to life.

Table of Contents

Preface

Free benefits with your book

Chapter 1: Introduction to Linux

Technical requirements

Role of Linux in data engineering

Linux distributions and their characteristics

Common shell tools for data manipulation

Summary

Chapter 2: Setting Up Your Linux Environment

Installing Ubuntu: choosing your environment

Option 1: Windows Subsystem for Linux (WSL) - recommended for Windows users •

Option 2: VirtualBox Virtual Machine •

Option 3: Native installation (dedicated machine or dual-boot) •

Understanding the Linux file system hierarchy structure

Exploring the file system using the graphical interface and the terminal

Summary

Chapter 3: Mastering Basic Linux Operations and Software Installation

Technical requirements

Installing software

Practical example: installing Nano
Manual pages •
Basic operations
System navigation and directories •
File creation and manipulation •
Viewing and searching file content
Archiving and compression
System and process management
Essential terminal keyboard shortcuts
Summary

Chapter 4: File Management Permissions and Text Editors

Technical requirements
File system permissions and the chmod command
1. Who gets access? (ownership) •
2. What kind of access? (permissions) •
3. Practical examples using chmod •
Editing content with the Nano editor
Editing content with the vi/Vim editor
System log files in /var/log
Viewing system logs •
Summary

Chapter 5: Automating Tasks with Bash Shell Scripting

Technical requirements
Fundamentals of Bash shell scripting
Features of shell scripts

Writing and executing a simple Bash script •
Working with script files and permissions •
Interactive shell scripting
Defining and using functions in Bash scripts
Using script parameters and arguments
Conditional statements in Bash
The if statement •
Boolean expressions and logical operators •
Case statements for pattern matching •
Loop constructs in Bash
The for loop •
The while loop •
The until loop •
Summary
Chapter 6: Practice Test
Summary
Chapter 7: Unlock the Code Bundle and the PDF Version
Unlock this book's free benefits in three easy steps
Other Books You May Enjoy
Index

Preface

To succeed in the modern data landscape, data professionals must be able to navigate the systems where data lives and is processed. Often, this system is Linux. This book is a practical guide specifically designed to get you comfortable with the Linux operating system, equipping you with the foundational skills needed to manage files, execute command-line operations, and automate repetitive data engineering tasks using Bash shell scripting.

You will find this book useful if you want to transition from basic user interfaces to the command line, understanding how Linux serves as the backbone for data engineering workflows. You will start by setting up your own Ubuntu Linux environment and understanding its core file system structure. As you progress, you will follow step-by-step instructions to master essential terminal shortcuts, manage file system permissions, and use command-line text editors such as nano and Vim. Finally, you will dive into Bash scripting to automate tasks using functions, loops, and conditional statements, culminating in a practice test to solidify your knowledge.

By the end of this book, you will be ready to confidently operate within Linux environments and write shell scripts to streamline your everyday data engineering workflows.

Who this book is for

If you are a budding data engineer, a data analyst looking to expand your technical toolkit, or a professional transitioning into the big data ecosystem, then this book is for you. No prior experience with Linux or shell scripting is required in order to proceed.

What this book covers

Chapter 1, *Introduction to Linux*, provides an overview of the role Linux plays in data engineering, introduces various Linux distributions and their characteristics, and highlights common shell tools used for data manipulation.

Chapter 2, *Setting Up Your Linux Environment*, guides you through choosing your environment and installing Ubuntu, explaining the Linux file system hierarchy structure, and exploring the system using both the graphical interface and the terminal.

Chapter 3, *Mastering Basic Linux Operations and Software Installation,* covers the installation of new software, how to access and read manual pages, basic command-line operations, and essential terminal keyboard shortcuts to improve your efficiency.

Chapter 4, *File Management Permissions and Text Editors,* explains how to manage file system permissions using the `chmod` command, introduces command-line text editing with nano and VI/Vim, and shows you how to locate and read system log files in `/var/log`.

Chapter 5, *Automating Tasks with Bash Shell Scripting*, introduces the fundamentals and features of Bash shell scripting. It covers interactive scripting, defining and using functions, passing script parameters and arguments, and building logic with conditional statements and loop constructs.

Chapter 6, *Practice Test*, provides 20 multiple-choice questions to test your comprehension of the concepts and commands covered throughout the book.

To get the most out of this book

No prior experience with Linux is necessary. A keen interest in automation and data engineering workflows is helpful. You will need a computer capable of running an Ubuntu environment, whether installed directly or run through a virtual machine.

If you are using the digital version of this book, it is advisable to type the command-line instructions and shell scripts yourself. Doing so will help you

avoid any potential execution errors related to the copying and pasting of formatted code.

Download the color images

Your purchase includes a color, DRM-free PDF copy of this book, ideal for viewing color images, screenshots, and diagrams. Refer to *Free benefits with your book section* at the end of the *Preface* to unlock your PDF copy.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates shell commands, bash scripts, system paths, directories, filenames, terminal input, and code words in the text. Here is an example: "To change permissions, we will use the `chmod` command:"

A block of code is set as follows:

```
#!/bin/bash
if [ $1 -gt 100 ]
then
  echo "Value is greater than 100"
fi
```

Any command-line input or output is written as follows:

```
sudo apt install nano
```

> **NOTE**
>
> Warnings or important notes appear like this.

> **TIP**
>
> Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book or have any general feedback, please email us at `customercare@packt.com` and mention the book's title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit Errata**, and fill in the form.

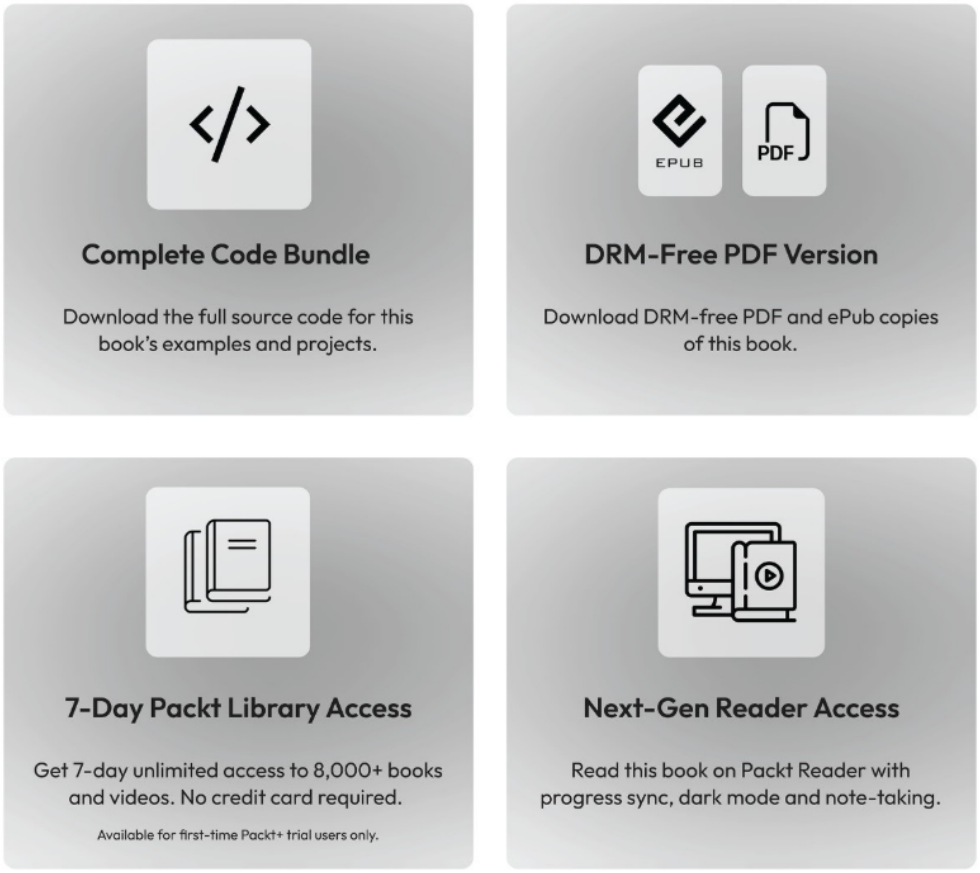
Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packt.com/`.

Free benefits with your book

This book includes free benefits designed to support your learning and help you apply what you learn effectively. Activate them now for instant access (see the *How to unlock* section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:



How to unlock

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.

UNLOCK NOW

Note: Have your invoice handy. Purchases made directly from the Packt website don't require an invoice.

Share your thoughts

Once you've read *Linux for Data Engineers*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.



https://packt.link/r/1808493338

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

1

Introduction to Linux

This chapter introduces Linux, a Unix-like, open-source operating system that has become a cornerstone for computers, servers, and embedded devices since its creation by Linus Torvalds in 1991. You will explore why Linux and the Bash shell are critical tools for data engineers, who rely on these systems to manage large, complex data sets and streamline processing tasks. You will also examine the characteristics of major Linux distributions, such as Debian, Fedora, and openSUSE, and clarify the distinctions between desktop and server environments. Mastering these foundational concepts is the first step toward effectively navigating and manipulating data on a Linux system.

This chapter covers the following topics:

- Role of Linux in data engineering
- Linux distributions and their characteristics
- Common shell tools for data manipulation

Technical requirements

This chapter is conceptual and requires no software setup.

NOTE

Download the code bundle and the PDF version of this book

Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Role of Linux in data engineering

In this section, you will be introduced to the essentials of Linux for data engineering, focusing on why Linux is a critical skill for a data engineer and what you will gain from mastering it. You will see how Linux and the Bash shell are used in real-world data engineering tasks and what practical skills you can expect to develop.

As a data engineer, you work with Linux machines that allow you to execute your code and shell scripts directly on the system. You are entering a field where the data footprint of organizations is massive, with data coming from many sources and in many formats. Dirty data can cause problems for analysts and data scientists, whose core role is to extract data and insights, so your role as a data engineer is vital in reducing this burden.

The following are the basic skills required for this data engineering role:

- **File system navigation:** The ability to seamlessly move through directories, locate raw data files, and manage storage structures without a graphical interface.
- **Shell scripting:** Writing Bash scripts to automate repetitive data **extraction, transformation, and loading** (**ETL**) workflows.
- **Command-line data manipulation:** Using built-in tools to quickly filter, parse, and clean massive datasets directly from the terminal.
- **Permissions management:** Understanding how to securely restrict or grant access to sensitive data files and system logs.

Now, let us look at some benefits of Linux and the Bash shell for data engineers.

Being comfortable with Linux and the Bash shell is a fundamental asset for data engineers. Because Linux environments are ubiquitous in enterprise data architectures, interacting with command-line tools is a daily requirement. Understanding common shell tools and how to use them to manipulate data will make you significantly more flexible and efficient when handling daily data pipelines.

Mastering these Linux essentials is a skill highly sought after by employers and technical recruiters. As organizations increasingly rely on big data processing, proficiency in the terminal provides a strong foundation for a lucrative career in data engineering. By grasping these core concepts, you will

be able to put them into practice immediately to solve real-world data challenges.

Although these skills apply broadly across Linux systems, Linux itself comes in several distributions designed for different environments and use cases. Understanding these distributions will help you recognize the types of systems you are likely to encounter as a data engineer.

Linux distributions and their characteristics

While all Linux systems share the same core kernel, they are packaged into different "distributions" (often called "distros") tailored for specific use cases. This is depicted here:

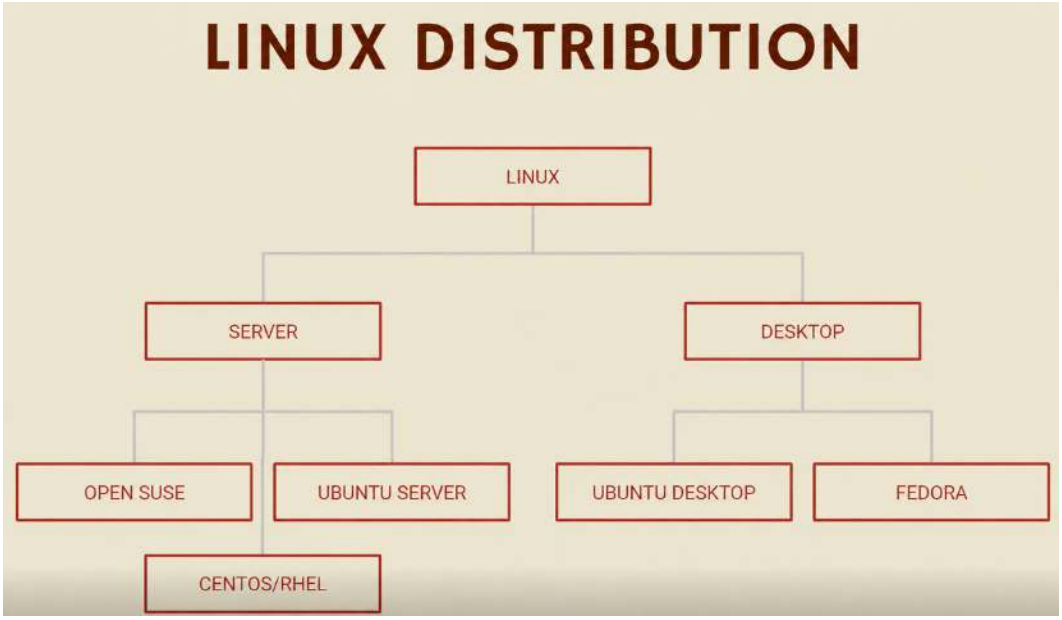


Figure 1.1 – The Linux distribution tree showing major server and desktop families

As shown in *Figure 1.1*, these distributions generally fall into two main categories:

- **Server distributions:** These are optimized for stability, security, and performance, typically without the overhead of a **graphical user interface** (**GUI**). They are the backbone of enterprise data engineering, running databases, web servers, and big data clusters. Examples include Ubuntu Server, CentOS, and openSUSE.
- **Desktop distributions:** These include a GUI and are designed for everyday computing, development, and personal use. They are user-

friendly and include pre-installed desktop applications. Examples include Ubuntu Desktop and Fedora.

Server and desktop describe how a Linux system is commonly used, while distribution families describe its broader ecosystem, packaging, and software-management lineage. From that perspective, data engineers commonly encounter the Debian, Red Hat, and SUSE families as mentioned here:

- **The Debian family:** Known for its stability and user-friendliness, with Ubuntu being the most popular derivative for both desktop and server environments.
- **The Red Hat family:** Highly prevalent in corporate environments. It includes commercial options such as **Red Hat Enterprise Linux** (**RHEL**) and community-driven alternatives such as CentOS and Fedora.
- **The SUSE family:** Strong in enterprise sectors, featuring SUSE Linux Enterprise Server and its community version, openSUSE.

Whichever distribution or environment you work with, many everyday data-engineering tasks are performed from the command line. Bash provides a common set of tools for navigating files and quickly inspecting or organizing data.

Common shell tools for data manipulation

As a data engineer, you will frequently need to inspect, filter, and move data files directly from the command line without opening heavy graphical applications. The Bash shell provides a powerful suite of built-in tools designed exactly for this purpose.

While you will dive deeper into how to execute these commands later in the book, here is a quick preview of the essential tools you will use to manage data:

- `cat`: Used to read the contents of a file and output it directly to your terminal screen. It is excellent for quickly inspecting small datasets, scripts, or configuration files.
- `head` and `tail`: When dealing with massive data files (such as a CSV with millions of rows), you do not want to load the entire file just to check its format. The `head` command allows you to preview the first few lines

(usually the column headers), while `tail` lets you check the final lines to verify the newest data entries.

- File management (`pwd`, `cd`, `ls`, `mkdir`, `rmdir`): Organizing your environment is critical. Verifying your current location (`pwd`), navigating through directories (`cd`), listing folder contents to ensure a dataset arrived (`ls`), and creating or removing folders for data pipelines (`mkdir` and `rmdir`) are everyday tasks.

By familiarizing yourself with these basic commands early on, you will be able to quickly validate data extracts, check system logs for errors, and organize files before loading them into databases or data warehouses.

These commands are only a preview; in the next chapter, you will begin working with the Linux environment directly and see how these concepts fit into the file-system structure.

Summary

In this chapter, you learned why Linux and the Bash shell are important and which foundational skills are necessary for handling large-scale data tasks in a professional environment. Starting from the role of Linux in data engineering, you explored common shell tools used for data manipulation and examined the characteristics of major Linux distributions, culminating in a comparison of desktop and server editions. A key distinction was drawn between data engineers, who focus on managing and cleaning data, and analysts or data scientists, who rely on this foundation to perform advanced analysis.

In the next chapter, you will put these concepts into practice by setting up your Linux environment and learning to navigate the file system hierarchy.

2

Setting Up Your Linux Environment

Setting up a Linux environment begins with getting the operating system running on your machine and understanding how it organizes its files and directories, a foundation that shapes every interaction you have with Ubuntu. By the end of this chapter, you will have successfully installed Ubuntu, whether natively or via Windows, and will be able to confidently navigate the Linux file system hierarchy. You will recognize the purpose of major directories such as `/home`, `/bin`, `/boot`, and `/etc`, and explore the file system through the graphical interface and through terminal commands such as `cd` and `ls`.

Along the way, key distinctions will be clarified, including how the root directory compares to the C drive in Windows, why `/bin` and `/sbin` serve different roles, and how `/media` and `/mnt` are used for removable and temporary storage. Mastering these basics not only helps you avoid confusion and errors but also lays the groundwork for more advanced tasks, such as deploying data engineering tools and databases, making your future work on Ubuntu smoother and more efficient.

In this chapter, the following topics are covered:

- Installing Ubuntu: choosing your environment
- Understanding the Linux file system hierarchy structure
- Exploring the file system using the graphical interface and the terminal

Installing Ubuntu: choosing your environment

To follow along with the exercises in this book, you need an Ubuntu environment. You can choose from three main installation paths depending on your hardware and preferences.

NOTE

Operating system installers receive minor graphical updates over time. If the screens you see differ slightly from the steps ahead, defer to the official Ubuntu documentation linked at the end of each section.

Option 1: Windows Subsystem for Linux (WSL) - recommended for Windows users

WSL allows you to run a full Linux environment directly inside Windows without the heavy overhead of a traditional virtual machine:

1. **Open PowerShell as administrator:** Click the Windows Start menu, type `PowerShell`, right-click the application, and select **Run as administrator**.
2. **Run the install command:** Type the `wsl --install` command and press Enter. This command automatically enables the required Windows features and downloads the default Ubuntu distribution.
3. **Restart your computer:** Once the installation finishes, reboot your machine to apply the changes.
4. **Initialize Ubuntu:** After restarting, a Linux terminal will open automatically. It will prompt you to create a UNIX username and a password. Keep these credentials safe, as you will need them to execute administrative commands.

NOTE

Official guide:

`https://ubuntu.com/wsl/docs/stable/howto/install-ubuntu-wsl2/`

Option 2: VirtualBox Virtual Machine

A **virtual machine** (VM) provides an isolated sandbox environment. This is an excellent choice if you want to keep your Linux setup completely separate from your host operating system and files.

To set up Ubuntu in a virtual machine using VirtualBox, follow these steps:

1. **Download prerequisites:** Download and install Oracle VM VirtualBox. Then, download the latest Ubuntu Desktop ISO file from the official

Ubuntu website.

2. **Create the VM**: Open VirtualBox and click **New**. Name your machine `Ubuntu`, and point the wizard to the ISO file you downloaded.
3. **Allocate resources:** Assign memory and CPU cores. For standard data engineering tasks on a computer with 16 GB of total RAM, allocating 4 GB to 8 GB of RAM to the VM ensures smooth performance while leaving enough memory for your host operating system to run comfortably.
4. **Create a virtual hard disk:** Allocate at least 25 GB of disk space for the operating system and future datasets.
5. **Install Ubuntu**: Start the VM. The Ubuntu installer will launch. Select **Install Ubuntu**, choose your keyboard layout, select **Normal Installation**, and choose **Erase disk and install Ubuntu** (this only erases the virtual disk, not your actual computer). Follow the prompts to create your user profile.

> NOTE
> Official guide:
> https://ubuntu.com/tutorials/how-to-run-ubuntu-desktop-on-a-virtual-machine-using-virtualbox

Option 3: Native installation (dedicated machine or dual-boot)

Installing Ubuntu natively provides maximum performance by giving the operating system direct access to your hardware. To install Ubuntu natively on a dedicated machine or alongside your existing operating system, follow these steps:

1. **Create a bootable USB**: Download the Ubuntu Desktop ISO. Use a tool such as Rufus (on Windows) or `BalenaEtcher` to flash the ISO onto a USB flash drive (at least 4 GB in size).
2. **Boot from the USB**: Insert the USB drive, restart your computer, and access your BIOS/UEFI settings (usually by pressing F2, F12, or DEL during startup). Change the boot order to boot from the USB drive first.

3. **Launch the installer:** Once Ubuntu loads from the USB, select **Install Ubuntu**.
4. **Choose the installation type:**
    - To dedicate the entire machine to Linux, choose **Erase disk and install Ubuntu**.
    - To keep your existing OS, choose **Install Ubuntu alongside Windows** (ensure you have backed up your files before resizing partitions).
5. **Complete setup**: Select your time zone, create your user account, and wait for the installation to finish. Remove the USB drive when prompted and restart.

> NOTE
> Official guide:
> `https://ubuntu.com/tutorials/install-ubuntu-desktop`

Understanding the Linux file system hierarchy structure

The Linux file system hierarchy organizes how your system, directories, and files are stored and accessed by the operating system. You will learn about the main directories, their purposes, and how they relate to each other, followed by hands-on exploration using both the graphical interface and the terminal.

The following diagram illustrates the structure of the Linux file system hierarchy, showing the root directory and its major subdirectories:

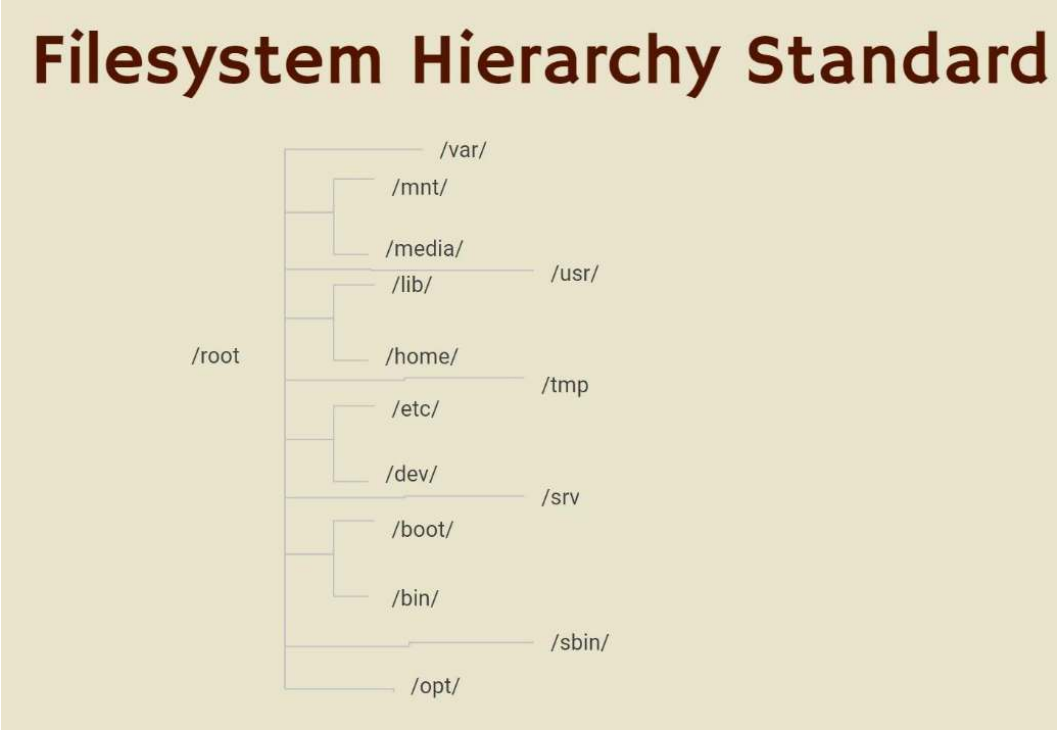


Figure 2.1 – Filesystem Hierarchy Sxtandard showing root and common Linux directories

The following is a detailed description of the directories depicted in *Figure 2.1*:

- `Root`: The root of the entire file system (`/`) is comparable to the Windows C drive, whereas `/root` is specifically the home directory for the root user.
- `Home`: The `home` directory is where user files are stored.
- `bin`: The `bin` directory contains essential command libraries.
- `boot`: The `boot` directory holds static files of the boot loader, which is where the operating system gets loaded.
- `dev`: The `dev` directory contains device files.
- `etc`: The `etc` directory stores host-specific system configuration.
- `lib`: The `lib` directory contains essential shared libraries and kernel modules.
- `media`: The `media` directory is the mount point for removable media.
- `mnt`: The `mnt` directory is the mount point for mounting a file system temporarily.
- `opt`: The `opt` directory is for additional application software packages.
- `sbin`: The `sbin` directory contains essential system binaries.

- `srv`: The `srv` directory holds data for services provided by the system.
- `tmp`: The `tmp` directory is for temporary files.
- `usr`: The `usr` directory is for user data and is a secondary hierarchy.
- `var`: The `var` directory is where variable data is stored.
- `/bin vs. /sbin`: The `/bin` directory contains essential commands (such as `ls` or `cd`) available to all users. In contrast, `/sbin` contains administrative commands (such as reboot) that require root or administrator privileges.
- `/media vs. /mnt`: The `/media` directory is an automatic mount point for removable media such as USB drives. The `/mnt` directory serves as a temporary, manual mount point for system administrators to mount external file systems.

Now that we have explored the Linux file system hierarchy and the purpose of its main directories, let's see how to navigate and access these directories using both the graphical interface and the terminal.

Exploring the file system using the graphical interface and the terminal

Let's explore the files.

There are two ways to do this:

1. To navigate via the graphical interface, open **Files**, select **Other Locations**, and click **Computer**:

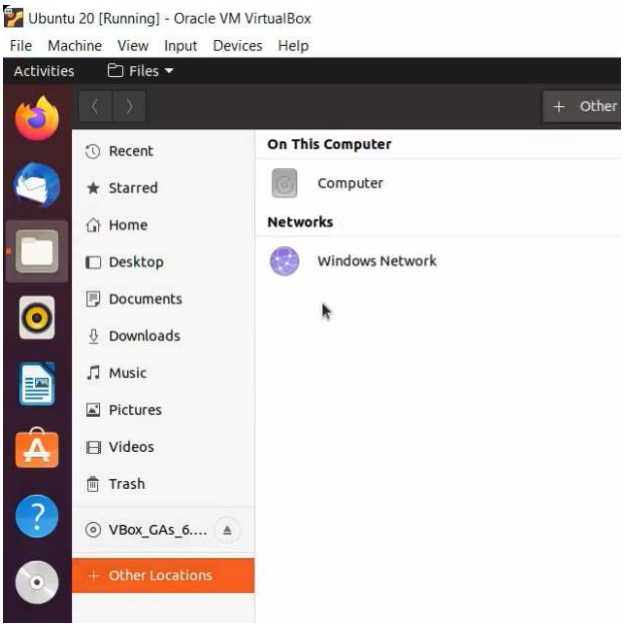


Figure 2.2 – Accessing the computer directory via the Other Locations menu

2. This opens the root directory, displaying the folders discussed earlier:



Figure 2.3 – Viewing the root directory folders in the graphical interface

Depending on your specific version of Ubuntu, you may see additional directories such as lib32 and lib64. These directories contain shared libraries specifically compiled for 32-bit and 64-bit applications, respectively.

The second way is through the terminal, depicted as follows:

1. To explore the file system using the command line, click **Show Applications** at the bottom left of the screen:



Figure 2.4 – Opening the terminal via Show Applications

2. Then type `terminal` in the search bar, and a command prompt window appears:

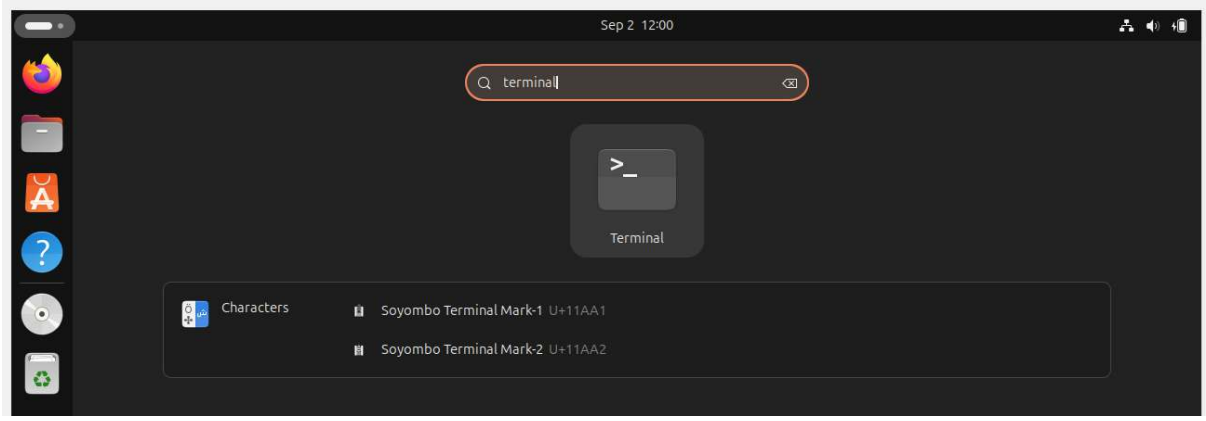


Figure 2.5 – Searching for the terminal application

3. Once the terminal is open, you can interact directly with the file system using commands. Have a look at the following screenshot:

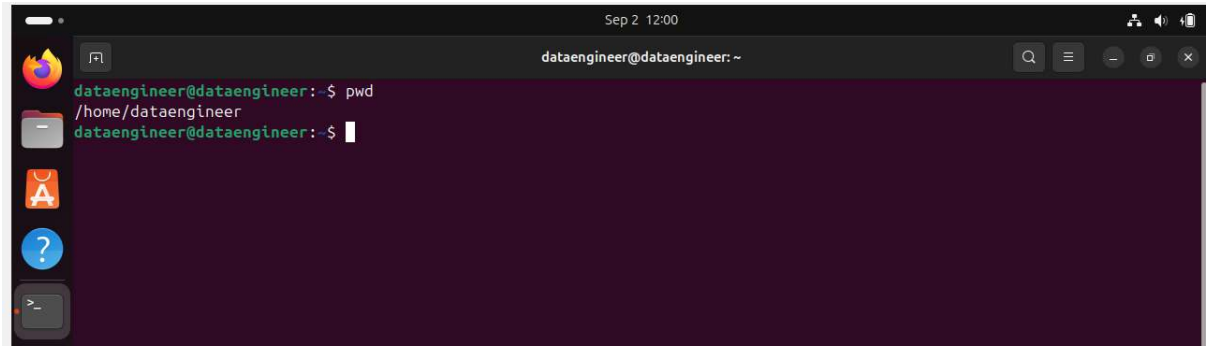


Figure 2.6 – The Ubuntu terminal interface ready for commands

Here are the essential commands:

- `pwd` (Print Working Directory): This command tells you exactly where you currently are in the file system hierarchy.
- `ls` (List): Typing `ls` will display all the files and folders within your current directory.
- `cd` (Change Directory): This allows you to navigate. For example, typing `cd /etc` will move you into the system configuration directory. Typing `cd ..` will move you up one level.
- `ls -ltr`: This is a more powerful version of the list command. The `-ltr` flags tell the system to display the files in a "long" format (showing permissions and file sizes), sorted by "time" modified, in "reverse" order (so the newest files appear at the very bottom).

You have now explored the Ubuntu file system using both the graphical interface and the terminal. While the graphical view helps you recognize the main directories visually, terminal commands such as `pwd`, `ls`, and `cd` give you a faster and more precise way to navigate and inspect them.

Summary

In this chapter, you explored three ways to set up Ubuntu: Windows Subsystem for Linux (WSL), a VirtualBox virtual machine, and a native installation. You then examined the Linux file system hierarchy, distinguishing the root directory (`/`) from the root user's home directory

(/root) and identifying the roles of key directories such as `/home`, `/etc`, `/usr`, and `/var`. You also explored graphical navigation in Ubuntu Desktop and terminal commands such as `pwd`, `ls`, and `cd` for locating and exploring directories.

Understanding this structure helps you find files, recognize where applications and system settings belong, and navigate your Ubuntu environment with greater confidence. This foundation prepares you to work with the tools and datasets used throughout the book.

In the next chapter, you will build on these basics by exploring essential Linux commands for managing files and processes, viewing and searching file contents, and monitoring system resources.

3

Mastering Basic Linux Operations and Software Installation

Navigating the Linux command line efficiently is a fundamental skill for any data engineer or developer working with big data infrastructure. By the end of this chapter, you will be able to confidently navigate directories, manipulate files, search for specific data patterns, and manage software packages entirely from the terminal.

Along the way, you will use essential utilities such as `cd`, `ls`, and `pwd` for system navigation, commands like `cat`, `cp`, `mv`, and `rm` for file management, and powerful search and archive tools including `grep`, `tar`, and `gzip`. You will also learn how to access built-in system documentation using `man` pages and manage software dependencies using the Advanced Package Tool (`apt`). Mastering these foundational operations not only speeds up your daily workflow but also allows you to combine commands and quickly prototype solutions, making the Linux command line an indispensable part of your toolkit.

In this chapter, we will cover the following topics:

- Installing software
- Practical example: installing Nano
- Basic operations
- Viewing and searching file content
- Archiving and compression
- System and process management
- Essential terminal keyboard shortcuts

Technical requirements

To follow along with the practical examples in this chapter, you will need access to a Linux terminal. This can be a native Linux system, a virtual machine (such as Ubuntu running on VirtualBox), or **Windows Subsystem for Linux** (**WSL**). No additional software is required.

> NOTE
>
> **Download the code bundle and the PDF version of this book**
>
> Your purchase includes a DRM-free PDF copy of this book, the code bundle, and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Installing software

In the dynamic environment of Linux, you will frequently need to add new tools, update existing utilities, and remove outdated dependencies to keep your system secure and functional. In Ubuntu and other Debian-based distributions, software management is primarily handled by **APT** (Advanced Package Tool). APT is not just a simple package installer; it is a fully fledged package manager. You can use APT to seamlessly install, update, remove, and even search for software packages across your system's repositories.

The most common and simplest way to install a package via APT is to run the `install` command followed by the package name:

```
sudo apt install package_name
```

> NOTE
>
> You use `sudo` before the command to execute it with "superuser" or administrative privileges, which is strictly required when modifying system-wide software.

Now that you understand how to install software using APT, let's look at a practical example of installing the nano text editor.

Practical example: installing Nano

Throughout this book, you rely on the `nano` text editor to write your Bash scripts. If `nano` is not already installed on your system, you can easily pull it from the repositories using APT.

To install the `nano` editor, simply run the following command in your terminal:

```
sudo apt install nano
```

Manual pages

A man page (short for manual page) is a form of built-in software documentation usually found on Unix or Unix-like operating systems. The topics covered in these manuals are extensive, including computer programs, library and system calls, formal standards and conventions, and even abstract concepts.

You do not need to memorize every flag or option for a command; instead, you can invoke a manual page at any time by issuing the `man` command followed by the topic you want to research.

> **NOTE**
>
> When you open a full manual page, it will take over your terminal screen. You can use your up and down arrow keys to scroll through the document. When you are finished reading, press *q* to quit and return to your normal command prompt.

The `man` command supports several options for finding the information you need. The following examples demonstrate how to display a short command description, access a specific manual section, and browse multiple manual pages for the same topic:

1. To print a short description of what a command does (in this case, the `mkdir` command), use the `-f` flag:

   ```
   man -f mkdir
   ```

   Output:

   ```
   mkdir (1)            - make directories
   ```

2. Manual pages are divided into distinct sections (for example, Section 1 for general commands and Section 3 for library calls). To force `man` to look in a specific section, provide the section number. For example, to view the C library function for `printf` instead of the standard shell command:

```
man 3 printf
```

Output: (This opens the interactive manual interface, starting with the following header)

```
PRINTF(3)               Linux Programmer's Manual              PRINTF(3)
NAME
       printf, fprintf, dprintf, sprintf, snprintf, vprintf, vfprintf,
       vdprintf, vsprintf, vsnprintf - formatted output conversion
```

3. To display all available manual pages matching a specific name across all sections in succession, use the `-a` flag:

```
man -a printf
```

This opens the first available manual page. When you press *q* to quit, the terminal will prompt you before opening the next related page.

Output:

```
--Man-- next: printf(3) [ view (return) | skip (Ctrl-D) | quit (Ctrl-C) ]
```

Now that you know how to look up command syntax using manual pages, let's apply that knowledge to everyday tasks such as navigating directories, managing files, and monitoring your system.

Basic operations

The Linux command line is incredibly powerful thanks to its robust suite of built-in utilities. To master Bash shell scripting, you must first be comfortable interacting with the system directly. The following are 26 essential commands, categorized by their primary functions, that will allow you to navigate your environment, manipulate data, and manage system resources.

System navigation and directories

Before working with files or running scripts, you need to understand how to move around the Linux file system and identify your current location. This section introduces the commands used to navigate directories, inspect their contents, create new folders, and remove empty ones. Together, these commands provide the foundation for organizing files and working confidently from the terminal:

1. `cd`: The `cd` command is used to change directory, allowing you to navigate seamlessly through the Linux file system.
   Syntax: `cd [directory_path]`

   Example:

   ```
   cd /var/log
   ```

   Explanation: This moves your current session into the system directory, making it your new working directory.

2. `ls`: The `ls` command is used to list the contents of the present working directory. It helps you see the files and folders around you.
   Syntax: `ls [options] [directory]`

   Common options:

   - `-l`: Long format. Shows detailed file permissions, ownership, sizes, and timestamps.
   - `-a`: All. Shows hidden files (files starting with a dot).

   Example:

   ```
   ls -l
   ```

   Explanation: This prints a detailed, vertically formatted list of all visible files and directories in your current location, including their read/write/execute permissions and file sizes.

3. `pwd`: The pwd command displays the present working directory. This is essential for confirming your exact location within the file system hierarchy.
   Syntax: `pwd`

   Example:

```
pwd
```

Explanation: This outputs the absolute path of your current directory (such as `/home/user/documents`), verifying exactly where you are before you execute destructive commands.

4. `mkdir`: The `mkdir` command creates a directory.
   Syntax: `mkdir [options] [directory_name]`

   Common options:

   - `-p`: Parent. Creates intermediate parent directories if they do not already exist.

   Example:

```
mkdir bash_scripts
```

   Explanation: This creates a brand-new, empty directory named `bash_scripts` inside your current working directory.

5. `rmdir`: The `rmdir` command removes an empty directory.
   Syntax: `rmdir [directory_name]`

   Example:

```
rmdir old_scripts
```

   Explanation: This deletes the directory named `old_scripts`, but only if it contains no files or subdirectories. If the folder contains data, the command will safely fail to prevent accidental data loss.

6. `whereis`: The `whereis` command is used to find the location of the source or binary file and manual sections for a given command.
   Syntax: `whereis [command_name]`

   Example:

```
whereis python3
```

   Explanation: This searches the system and outputs the absolute file paths for the `python3` executable binary, its source code, and its manual pages.

With an understanding of how to navigate directories, let's move on to creating and managing files within them.

File creation and manipulation

Once you know how to navigate directories, the next step is to work with the files they contain. Linux provides commands for creating files, copying them, moving or renaming them, and removing files that are no longer needed. In this section, you will learn how to perform these everyday file operations directly from the terminal while understanding the effect each command has on your files.

1. `touch`: The `touch` command is often used to set or update the access, change, and modify times of files. However, its most common use case for beginners is simply creating a new, empty file.
   Syntax: `touch [filename]`

   Example:

   ```
   touch new_script.sh
   ```

   Explanation: This creates a blank file named `new_script.sh` in the current directory. If the file already exists, it updates its modification timestamp without altering its contents.

2. `cp`: The `cp` command is used to copy files from one location to another while leaving the original intact.
   Syntax: `cp [options] [source_file] [destination]`

   Common options:

   `-r`: Recursive. Required when copying entire directories and their contents.

   Example:

   ```
   cp config.txt config_backup.txt
   ```

   Explanation: This creates an exact duplicate of `config.txt` and saves the new copy as `config_backup.txt` in the same directory.

3. `mv`: The `mv` command renames a file or directory or physically moves it to a completely new location.

Syntax: `mv [source] [destination]`

Example:

```
mv data.csv /home/user/archive/
```

Explanation: This moves the file `data.csv` out of your current directory and places it into the `/home/user/archive/` directory. If the destination were just a new filename, it would rename the file instead.

4. `rm`: The `rm` command removes a file. Use this carefully, as deleted files in the terminal are not sent to a recycling bin.
   Syntax: `rm [options] [filename]`

   Common options:

   - `-r`: Recursive. Deletes a directory and all files inside it.
   - `-f`: Force. Bypasses confirmation prompts.

   Example:

   ```
   rm outdated_report.txt
   ```

   Explanation: This permanently deletes the file `outdated_report.txt` from the system.

Now that you know how to create and manage files, let's explore the commands used to view their contents and search for specific information.

Viewing and searching file content

Working with files often involves examining their contents rather than modifying them. Linux provides several commands to display text, inspect portions of large files, search for matching patterns, and count lines or words. This section introduces these utilities and explains when to use each one, whether you need to preview a dataset, examine a log file, or locate specific information without opening a text editor:

1. `cat`: The `cat` (concatenate) command is used for viewing files. It prints the entire content of a file directly to your terminal screen.
   Syntax: `cat [filename]`

   Example:

```
cat /etc/os-release
```

Explanation: This reads the system file `/etc/os-release` and prints its entire contents to the terminal, which is useful for checking which Linux distribution you are using.

2. `less`: The `less` command is very efficient for viewing huge log files, as it does not need to load the full file when opening it. It allows you to scroll up and down interactively.
   Syntax: `less [filename]`

   Example:

```
less /var/log/syslog
```

   Explanation: This opens the system log file in an interactive reading environment. You can use your arrow keys to scroll through the data page by page without overwhelming your terminal window.

3. `head`: The `head` command prints the top 10 lines of the specified files. This is perfect for quickly previewing the headers of a large dataset.
   Syntax: `head [options] [filename]`

   Common options:

   - `-n [number]`: Specifies exactly how many lines to print instead of the default 10.

   Example:

```
head -n 5 sales_data.csv
```

   Explanation: This reads the `sales_data.csv` file and prints only the first five lines to the terminal, allowing you to quickly inspect the file's structure.

4. `tail`: The tail command prints the last 10 lines of the specified files. It is highly useful for checking the most recent entries in error logs.
   Syntax: `tail [options] [filename]`

   Common options:

   - `-n [number]`: Specifies exactly how many lines to print.

- `-f`: Follow. Keeps the file open and outputs appended data live as the file grows.

Example:

```
tail error.log
```

Explanation: This outputs the final 10 lines of `error.log`, allowing you to see the most recently recorded system or application errors.

5. `grep`: The `grep` filter searches a file for a particular pattern of characters and displays all lines that contain that pattern.
Syntax: `grep [options] "pattern" [filename]`

   Common options:

   - `-i`: Ignore case (matches both upper- and lowercase).
   - `-v`: Invert match (shows only lines that do not contain the pattern).

   Example:

```
grep "ERROR" server.log
```

   Explanation: This scans the entirety of `server.log` and filters the output, printing only the specific lines that contain the exact uppercase word `ERROR`.

6. `wc`: The `wc` (word count) command counts the number of lines, words, and characters in a file or list of files.
Syntax: `wc [options] [filename]`

   Common options:

   - `-l`: Prints only the line count.
   - `-w`: Prints only the word count.
   - `-c`: Prints only the byte/character count.

   Example:

```
wc -l document.txt
```

   Explanation: Because of the `-l` flag, this reads `document.txt`, counts the total number of lines, and prints that single numerical value to the screen.

Now that you can inspect and search file contents, let's explore how to package files into archives and compress them for easier storage and transfer.

Archiving and compression

When working with multiple files or large datasets, it is often useful to package files together or reduce the amount of storage space they occupy. Linux provides utilities for creating archives and compressing files, making them easier to store, transfer, and share. In this section, you will explore `gzip`, `zip`, and `tar` and understand how these commands support different archiving and compression tasks:

1. `gzip`: The `gzip` command is the most frequently used Linux compression utility. It replaces the original file with a compressed `.gz` version to save space.
   Syntax: `gzip [options] [filename]`

   Common options:

   - `-d`: Decompress. Extracts the `.gz` file back to its original state.

   Example:

   ```
   gzip large_dataset.csv
   ```

   Explanation: This compresses `large_dataset.csv`, replacing the original file on your disk with a smaller compressed file named `large_dataset.csv.gz`.

2. `zip`: The `zip` command is often required to examine and decompress archives from other operating systems, making cross-platform file sharing easier.
   Syntax: `zip [options] [archive_name.zip] [files_to_zip]`

   Common options:

   - `-r`: Recursive. Packages a directory and all of its contents.

   Example:

   ```
   zip project_files.zip script1.sh script2.sh
   ```

   Explanation: This takes two files, `script1.sh` and `script2.sh`, and packages them together into a single, compressed archive named

`project_files.zip`.

3. `tar`: The `tar` utility bundles multiple files into a single archive file. To extract all the files in `mydir.tar` into the `mydir` directory, use the `xvf` flags.
   Syntax: `tar [options] [archive_name.tar]`

   Common options:

   - `-c`: Create a new archive.
   - `-x`: Extract an existing archive.
   - `-v`: Verbose (shows the files being processed).
   - `-f`: File (specifies the name of the archive file).

   Example:

   ```
   tar xvf mydir.tar
   ```

   Explanation: This extracts (`x`) the contents of the archive `mydir.tar (f)` while verbosely (`v`) printing the name of each file to the screen as it is being unpacked.

With archiving and compression covered, let's turn to monitoring running processes, checking resource usage, and managing your Linux system.

System and process management

In addition to managing files, you need to understand what is happening on your Linux system. Monitoring running processes, checking resource usage, and reviewing available disk space can help you identify performance issues and manage everyday tasks more effectively. This section introduces commands for inspecting system activity, checking storage usage, displaying the current date, reviewing previously executed commands, and ending terminal sessions:

1. `ps`: The `ps` command produces a list of processes along with status information for the system.
   Syntax: `ps [options]`

   Common options:

   - `-e`: Selects all active processes on the system.

- `-f`: Full-format listing (provides more details such as user ID and start time).

Example:

```
ps -ef
```

Explanation: This displays a comprehensive snapshot of every active process currently running on the system (`-e`), formatted with full technical details (`-f`).

2. `top`: The `top` command is a utility that shows processes running, CPU usage, and memory usage in a live, interactive dashboard.
Syntax: `top`

   Example:

```
top
```

   Explanation: This launches a real-time, dynamically updating interface displaying system resource usage and a ranked list of the most resource-intensive processes. Press *q* to exit.

3. `df`: The df ("disk free") command is used to show information about free disk space on a file system.
Syntax: `df [options]`

   Common options:

   - `-h`: Human-readable format (displays sizes in MB and GB instead of raw bytes).

   Example:

```
df -h
```

   Explanation: This prints a table of the total, used, and available disk space for all mounted file systems, making the output easy to read by formatting the numbers into gigabytes and megabytes (`-h`).

4. `du`: The `du` ("disk usage") command is most often used to determine the size of the disk or a specific directory.
Syntax: `du [options] [directory]`

   Common options:

- `-s`: Summarize. Shows only a grand total for the directory instead of listing every file inside it.
- `-h`: Human-readable sizes.

Example:

```
du -sh /var/log
```

Explanation: This calculates the total disk space consumed by the `/var/log` directory and all its contents, summarizing the final result (`-s`) in a human-readable format (`-h`).

5. `date`: The date command is helpful for displaying the date in several formats. It is frequently used in scripting to timestamp log files.
   Syntax: `date [options] [+format]`

   Common format specifiers:

   - `%Y`: Four-digit year (for example, 2026).
   - `%m`: Two-digit month (01-12).
   - `%d`: Two-digit day (01-31).

   Example:

   ```
   date "+%Y-%m-%d"
   ```

   Explanation: This outputs the current system date cleanly formatted as a four-digit year, two-digit month, and two-digit day (for example, "2026-10-01").

6. `history`: The `history` command displays the history of commands executed during your terminal sessions.
   Syntax: `history`

   Example:

   ```
   history
   ```

   Explanation: This prints a numbered, chronological list of the most recent commands you have typed and executed in your terminal, which is helpful if you need to remember a complex command you ran earlier.

7. `exit`: The exit command is used to exit a session, safely closing your terminal or logging you out of a remote server.

Syntax: `exit`

Example:

```
exit
```

Explanation: This terminates your current terminal session and logs you out of the shell environment.

Now that you understand how to monitor system activity and manage processes, let's explore some essential keyboard shortcuts that make working in the Linux terminal faster and easier.

Essential terminal keyboard shortcuts

Beyond typing individual commands, the Linux terminal supports several crucial keyboard shortcuts. These shortcuts send specific control signals to the shell, allowing you to manage running processes, edit long commands on the fly, and navigate your session with speed and efficiency.

1. `Tab` (Command and file autocompletion).
   Task: Autocompletes commands, file names, or directory paths.

   Explanation: This is arguably the most time-saving key in Linux. Type the first few letters of a file or command and press Tab. If there is only one match, the terminal types the rest for you. If there are multiple matches, pressing *Tab* twice lists all possible options.

2. `Up/Down` arrow keys (Navigate history).
   Task: Cycles through your previously executed commands.

   Explanation: Instead of typing out a long command a second time, simply press the Up arrow to recall it. You can seamlessly edit the recalled command before pressing **Enter** to run it again.

3. `CTRL-L` (Clear screen).
   Task: Clears the terminal screen.

   Explanation: This visually sweeps away all previous command output, bringing your fresh prompt back to the top of the terminal window. It performs the exact same function as typing the `clear` command, but is much faster to execute when you need a clean workspace.

4. `CTRL-C` (Interrupt process).
   Task: Kills the current process.

   Explanation: This sends an interrupt signal (SIGINT) to the application currently running in the foreground. Think of this as your emergency brake—use it to immediately force-quit runaway scripts, infinite loops, or commands that are taking too long to execute.

5. `CTRL-Z` (Suspend process).
   Task: Puts the current process into a suspended background state.

   Explanation: Unlike *CTRL-C* which kills a task entirely, *CTRL-Z* simply pauses it (sending a SIGTSTP signal) and immediately gives you your terminal prompt back. This is incredibly useful if you need to run a quick command without completely terminating the job you were just running. *(Note: You can later resume the paused process by typing* `fg` *for foreground or* `bg` *for background.)*

6. `CTRL-D` (End of file / exit).
   Task: Exits the current shell.

   Explanation: This shortcut sends an **end-of-file** (**EOF**) marker to the terminal. If you press this at an empty command prompt, it signals to the shell that no more input is coming, which will safely log you out of your current session or close the terminal window. It functions identically to the `exit` command.

7. `CTRL-A` (Move to beginning).
   Task: Moves the cursor to the very beginning of the current line.

   Explanation: If you type a long command and realize you forgot to add `sudo` at the front, pressing *CTRL-A* instantly jumps your cursor to the start of the line so you do not have to hold down the left arrow key.

8. `CTRL-E` (Move to end).
   Task: Moves the cursor to the very end of the current line.

   Explanation: The exact counterpart to *CTRL-A*. After fixing a typo at the beginning of a line, use *CTRL-E* to jump back to the end so you can finish typing your command.

9. `CTRL-U` (Cut to beginning).

Task: Deletes everything from your cursor's current position to the beginning of the line.

Explanation: If you completely mess up a command you are typing and want to start over, pressing *CTRL-U* acts as a rapid "clear line" button, saving you from holding down the Backspace key.

10. `CTRL-R` (Reverse search).
    Task: Searches your command history for a specific keyword.

    Explanation: Press *CTRL-R* and start typing a fragment of a command you used yesterday. The terminal will instantly pull up the most recent match. Press *CTRL-R* again to cycle backward through older matches until you find exactly what you need.

With these commands and keyboard shortcuts, you can navigate the Linux terminal more efficiently, monitor system activity, manage processes, and perform everyday tasks with greater confidence.

Summary

In this chapter, you developed the foundational skills necessary to confidently navigate and manage a Linux system directly from the command line by utilizing a core set of built-in utilities. Starting with software management, you explored how to extend your system's capabilities and resolve dependencies by installing tools such as `nano` using the Advanced Package Tool (`apt`). You then learned how to independently research command syntax and options by accessing the system's built-in documentation through `man` pages. Moving into basic operations, you systematically covered the essential commands required to traverse directories (`cd, pwd, ls`), create and manipulate files (`touch, mkdir, cp, mv, rm`), and inspect or filter data (`cat, less, grep, tail`). You also expanded your toolkit to include archiving files with `tar` and `gzip` and monitoring system resources using `ps`, `top`, `df`, and `du`. Finally, you were introduced to essential terminal keyboard shortcuts – such as *Tab* for command autocompletion, *CTRL-C* to interrupt running processes, and *CTRL-R* for reverse searching execution history – highlighting their role in drastically reducing manual typing and accelerating daily workflows. With these techniques, you built a highly practical foundation of command-line

fluency, preparing you to seamlessly manage file permissions and write robust Bash scripts in the upcoming chapters.

In the next chapter, you will learn to manage file permissions with `chmod`, edit text using nano and vi/Vim, and inspect system logs in `/var/log` to troubleshoot issues and monitor system activity.

4

File Management Permissions and Text Editors

File management on Unix systems demands a clear understanding of permissions and the ability to edit files efficiently, and this chapter is dedicated to building those skills from the ground up.

By the end, you will be able to confidently interpret and modify file system permissions using numeric values, navigate and edit text files with both the nano editor and the vi (Vim) editor, and view essential system log files such as `auth.log`, `lastlog`, `faillog`, and dpkg.log within the `/var/log` directory.

Along the way, you will use the `ls -later` command to distinguish between user, group, and other permission categories and view their respective read, write, and execute permissions. You will also use `chmod` to modify these permissions using numeric values and `cat` to read file contents directly from the command line.

Mastering file permissions is crucial for maintaining system security, troubleshooting issues, and ensuring that logs are accessible only to authorized users. Additionally, proficiency with command-line text editors will significantly streamline your everyday scripting and configuration tasks.

In this chapter, we will cover the following topics:

- File system permissions and the `chmod` command
- Editing content with the Nano editor
- Editing content with the vi/Vim editor
- System log files in `/var/log`

Technical requirements

To follow along with the practical examples in this chapter, you will need access to a Linux terminal. This can be a native Linux system, a virtual machine (such as Ubuntu running on VirtualBox), or Windows Subsystem for Linux (WSL). No additional software is required.

File system permissions and the chmod command

Understanding file system permissions is crucial for managing who can read, write, or execute files and directories in Linux. Let's break this down into two simple concepts: **who** gets access, and **what** kind of access they get.

1. Who gets access? (ownership)

Whenever a file is created in Linux, access is divided into three specific categories:

- **User**: The person who created and owns the file.
- **Group**: Other users who are part of the file's assigned group. For example, assigning a file to a `database` group gives all database administrators access to it.
- **Others**: Everyone else on the system who is neither the owner nor part of the group.

2. What kind of access? (permissions)

Permissions are displayed as a 9-character string broken into three sets of triplets, one for the **User**, one for the **Group**, and one for **Others**. Within each triplet, `r` stands for read, `w` for write, and `x` for execute (a hyphen `-` means the permission is denied). For example, a full triplet of `rwx` means all three permissions are granted for that specific category. To make changing these permissions easy, Linux assigns a numeric value to each action:

- **Read (r) = 4**: The ability to view the file's contents.
- **Write (w) = 2**: The ability to modify or delete the file.
- **Execute (x) = 1**: The ability to run the file as a script or program.

3. Practical examples using chmod

Before changing permissions, it helps to see how Linux displays them. If you run the (`ls -ltr sample.txt`) list command, you will see output that looks like this:

```
-rwxr-xr-- 1 bigdata database 4096 Sep 28 10:00 sample.txt
```

Let's break down that permission string (`-rwxr-xr--`) from left to right:

- **- (File Type)**: The very first dash indicates this is a regular file.
- **rwx (User)**: The next three characters show the owner (`bigdata`) has read, write, and execute access.
- **r-x (Group)**: The middle three characters show the assigned group (`database`) has read and execute access, but write is denied.
- **r-- (Others)**: The final three characters show everyone else only has read access.

To change a file's permissions, you use the `chmod` command. You simply calculate the new permissions by adding the numeric values together for the user, group, and others.

Let's look at how to apply this math to a file named `sample.txt`:

- `chmod 766 sample.txt`
  - **User**: 4+2+1 = 7 (Read, write, and execute).
  - **Group**: 4+2 = 6 (Read and write).
  - **Others**: 4+2 = 6 (Read and write).
- `chmod 744 sample.txt`
  - **User**: 7 (Read, write, execute).
  - **Group and others:** 4 (read-only access).
- `chmod 444 sample.txt`
  - **All users**: 4 (Strictly read-only access for everyone).
- `chmod 777 sample.txt`
  - **All users**: 7 (Read, write, and execute for everyone).

NOTE

(Expert note: Granting `777` permissions is considered a severe security risk and should generally be avoided in production environments!)

Now that you understand how to control access to your files, let's look at how to create and modify their contents using the Nano editor.

Editing content with the Nano editor

The Nano editor is a simple and easy-to-use text editor, similar to Notepad in Windows. It requires very little effort to learn, making it accessible for beginners.

To open an existing file or create a new one, use the command in this format:

```
nano filename.extension
```

This command opens an existing file or creates a new one. For example, running `nano file1.txt` will open the file for editing if it already exists. If it does not exist, Nano opens a new workspace (buffer) for you to begin typing.

Let's look at a practical example. First, check which files are currently in your directory by listing them:

```
ls -ltr
```

Now, type `nano file1.txt`. Because the file does not yet exist, Nano opens a blank editor screen where you can write your content. Type the following:

```
This is file1
hello world
```

To save the file to your disk, press *Ctrl+O* (Write Out) and then press Enter to confirm the file name. To exit Nano, press *Ctrl+X*.

If you want to view the contents of your newly created file without opening the editor, use the `cat` command:

```
cat file1.txt
```

This will print the text you just inserted directly to your screen.

If you run `nano file1.txt` again, the editor opens your saved file so you can add more text. For example, add a new line:

```
This is the third line
```

If you press *Ctrl+X* to exit without saving first, Nano will ask if you want to save the modified buffer. Press *N* to discard your changes, or press *Y* to save them. After pressing *Y*, Nano will prompt you to confirm the file name; press Enter to overwrite the file with your new changes.

You can verify the file was updated by running `cat file1.txt` again. Nano makes text editing simple and efficient, allowing you to quickly create and modify files using only a few basic keystrokes.

Next, you will move to the vi/Vim editor.

Editing content with the vi/Vim editor

In this section, you will explore `vi` and `Vim` (Vi IMproved). Originally created for the Unix operating system, `vi` is a powerful, screen-oriented text editor.

Before opening the editor, you can check the contents of your current directory by typing `ls -ltr` in your terminal. For this exercise, you will work on a file named `myfile.txt`. If it is missing from your directory, you can create and open it simultaneously by executing the following command:

```
vi myfile.txt
```

When `vi` opens, it starts in **Normal mode**. To begin writing content in the file, you must switch to **Insert mode**. Press the *I* key to enter Insert mode, which allows you to start typing. Type the following text:

```
This is the first line of my file
```

When you have finished typing, press the *Esc* (Escape) key to return to Normal mode. From Normal mode, you can save your changes or exit the editor by typing the following commands and pressing *Enter*:

- `:w` – Saves the file.
- `:wq` (or `:x`) – Saves the file and exits the editor.
- `:q` – Exits the editor (this only works if no changes have been made).
- `:q!` – Exits the editor *without* saving any changes.

Now have a look at the cursor on the screen:

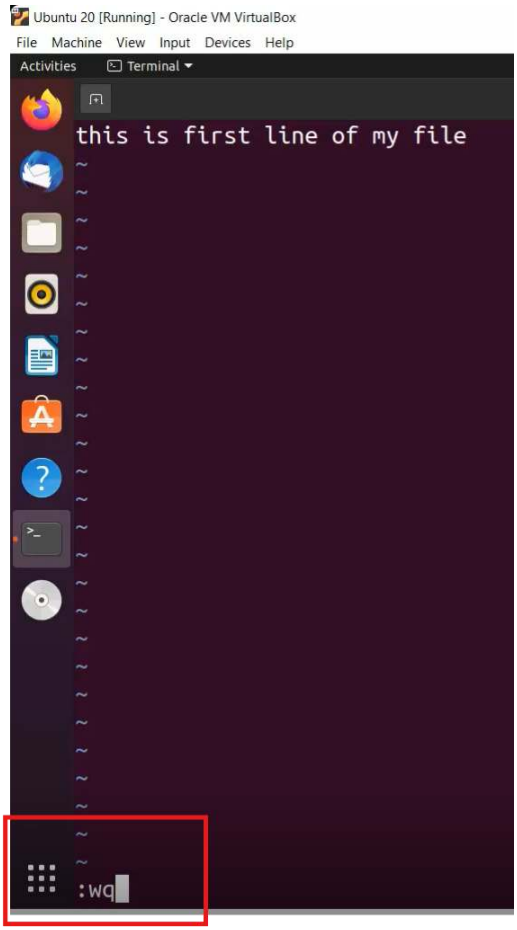


Figure 4.1 – Saving and exiting a file in the vi editor

As highlighted in the preceding screenshot, entering `:wq` will write your changes to the disk and quit the editor. Conversely, if you type `:q!`, `vi` will forcefully quit without saving any of your recent edits. If you were to verify the file's contents from the terminal using the command `cat filename.extension`, you would see that none of your unsaved changes were applied.

To properly save your work, always ensure you are in Normal mode by pressing the *Esc* key, then type `:wq` and press *Enter*. Once you return to the terminal command prompt, you can use the `cat` command to verify that your new text, including any blank lines, has been successfully saved to the file.

This covers the basic workflow for creating and editing files using the `vi` editor. While `vi` is a powerful tool universally available on nearly every Unix and Linux operating system, many users prefer the `nano` editor for its straightforward, user-friendly interface.

Now that you can create and edit text files, let's explore how to read system logs in /var/log to investigate issues and monitor system activity.

System log files in /var/log

Monitoring system log files is an essential skill in Linux, as these logs provide vital clues for troubleshooting issues, tracking user activity, and monitoring security events. In this section, you will learn how to navigate the system's log directory and interpret the contents of essential log files.

System log files are stored in the `/var/log` directory. The four log files you will examine are `auth.log`, `lastlog`, `faillog`, and `dpkg.log`.

Let's understand each of these:

- `auth.log`: Contains system authentication information, including user logins and the authentication mechanisms used.
- `lastlog`: Stores recent login information for all users.
- `faillog`: Contains records of failed user login attempts.
- `dpkg.log`: Logs information regarding software package installations and removals.

Now that you understand what each log records, let's examine how to view them from the terminal.

Viewing system logs

Let's navigate to the `/var/log` directory to explore the four specific logs mentioned above. Follow these steps:

1. First, navigate to the directory and list the files:

```
cd /var/log
ls -ltr
```

2. View the authorization logs to observe how users are logging into the system and which ports are being utilized (port 22 is used for SSH by default):

```
cat auth.log
```

3. Since the `lastlog` file is not a standard ASCII text file, you cannot use the `cat` command. Instead, execute the command directly to view a

formatted list of recent user logins:

```
lastlog
```

4. Display the contents of the failed login records. (If there have been no unauthorized access attempts, the output will simply be empty):

```
faillog
```

5. Finally, view package-related activity to see the current status of all package installations, upgrades, and removals:

```
cat dpkg.log
```

You have now explored how to view authentication activity, recent logins, failed login attempts, and package changes using logs in `/var/log`. These records help you investigate system issues, identify suspicious activity, and understand changes made to your Linux system.

Summary

In this chapter, you established a practical understanding of file management and text editing. You started with the fundamentals of file system permissions, learning how to assign specific access levels to the user, the group, and others. You then applied this knowledge by using numeric values and the `chmod` command to securely manage file access. Next, you moved into hands-on text editing, contrasting the beginner-friendly **nano** editor with the powerful, screen-oriented **vi** editor. Finally, you navigated to the `/var/log` directory to explore key system logs such as `auth.log, lastlog, faillog, and dpkg.log`.

With these skills, you are now equipped to control file permissions, modify configuration scripts, and monitor essential system activity in any Unix-based environment.

In the next chapter, you will build on these skills by writing and running Bash scripts that automate tasks, accept user input, and control execution through functions, arguments, conditional statements, and loops.

5

Automating Tasks with Bash Shell Scripting

Automating repetitive or complex tasks is a core skill for any engineer working with big data, and Bash shell scripting offers a direct, efficient way to achieve this on Unix-like systems. By the end of this chapter, you will be able to write, execute, and manage Bash scripts that automate file operations, process user input, handle conditional logic, and control program flow using loops and functions.

Along the way, we will use tools such as `nano` for editing scripts, and commands like `chmod 755` and `ls -ltr` for managing file permissions. You will also explore key Bash features, including the shebang (`#!/bin/bash`), script parameters (`$0`, `$1`, `$2`), and control structures like `if`, `else`, `case`, `for`, `while`, and `until` loops. Mastering these techniques not only reduces manual errors and saves time but also allows you to combine commands, share repeatable procedures, and quickly prototype solutions, making Bash scripting an indispensable part of the data engineer's toolkit.

In this chapter, we will cover the following topics:

- Fundamentals of Bash shell scripting
- Features of shell scripts
- Interactive shell scripting
- Defining and using functions in Bash scripts
- Using script parameters and arguments
- Conditional statements in Bash
- Loop constructs in Bash

Technical requirements

To follow along with the practical examples in this chapter, you will need access to a Linux terminal. This can be a native Linux system, a virtual machine (such as Ubuntu running on VirtualBox), or the Windows Subsystem for Linux (WSL). No additional software is required.

Fundamentals of Bash shell scripting

Bash is a command processor that runs in a text window, allowing you to type commands that trigger actions. You can also have Bash read and execute commands from a file, which is called a shell script. You learn how to write a simple Bash script that displays a two-line message on the screen and you see how to execute it.

Features of shell scripts

Shell scripts offer several features that make them useful for automating tasks. Some are listed here:

- Shell scripts can automatically perform tasks and reduce the risk of error.
- You can combine long and repetitive sequences of commands into one simple command, making complex operations easier to manage.
- Shell scripts allow you to share procedures among users.
- Shell scripts provide a control interface to users.
- You can create new commands by combining existing utilities.
- You can quickly prototype solutions, and there is no need to compile the scripts before running them.

These features make shell scripts a practical tool for many scenarios.

With these advantages in mind, let's begin with the simplest Bash script and see how to create and execute it.

Writing and executing a simple Bash script

To write a simple Bash script that prints two lines, perform the following steps:

1. Open the nano editor with the command:

```
nano example-shell-script.sh
```

2. Type the following three lines:

```
#!/bin/bash
echo big data
echo engineer
```

The first line is the **shebang** (`#!`), which specifies which interpreter should run the script.

There are various kinds of shells available in Linux, such as `bash`, `sh`, and `zsh`. Here, you are using `bash`. The shebang (`#!`) followed by the absolute path to the bash interpreter (`/bin/bash`) ensures that the system uses the correct shell to execute your commands.

3. Save and exit the editor.
4. Provide execute permissions so the file can run as a script. Type the following command:

```
chmod 755 example-shell-script.sh
```

5. You can run the script by typing either:

```
./example-shell-script.sh
```

Or you can run:

```
bash example-shell-script.sh
```

The result will be displayed on two different lines:

```
big data
engineer
```

Now that you have written and executed a basic Bash script, the next step is to understand how the script file itself is managed, including its location and execute permissions.

Working with script files and permissions

Here is how we begin:

1. Type the following command to change directories:

```
cd /home/big_data/files
```

2. To confirm your current location, type:

```
pwd
```

This indicates that you are in the `/home/big_data/files` directory.

3. Then, you list the files in this directory by typing the following:

```
ls -ltr
```

In this list, you will see that `example-script.sh` is not present.

4. So next, create a new shell script file that opens the nano editor by typing:

```
nano example-script.sh
```

You are now in the nano editor.

5. Copy and paste the following three lines into the editor:

```
#!/bin/bash
echo "big data"
echo "engineer"
```

Press *Ctrl+X* to exit. When prompted to save the file, type *y* for yes and press *Enter*.

6. To verify the contents of your new script, type the following command:

```
cat example-script.sh
```

This will display the three lines you entered.

So now you need to make your script executable before running it. To do this, first check the permissions on your script file:

7. Type the following command to list the permissions:

```
ls -ltr example-script.sh
```

You will see that you do not have execute permission for the script.

8. Grant execute permission by running:

```
chmod 755 example-script.sh
```

Now, if you check the permissions again, you will see that you have read, write, and execute permissions.

```
ls -ltr example-script.sh
```

9. Clear the terminal screen for clarity:

```
clear
```

10. Execute your script using:

```
./example-script.sh
```

This will display the output on different lines as shown here:

```
big data
engineer
```

11. Alternatively, you can run the script with:

```
bash example-script.shp
```

This will execute the same script and display the same result.

This demonstrates how to write a small Bash shell script and execute it.

So far, your scripts have used fixed values; next, you will make them interactive by accepting user input.

Interactive shell scripting

In this section, you will explore interactive shell scripting.

You will begin by writing an interactive shell script that accepts a value from a user and then displays it to them.

Perform the following steps:

1. You first list all the files using `ls -lrt`. Because `interactive-script.sh` is missing, add it as follows:

```
nano interactive-script.sh
```

2. You input the following code:

```
#!/bin/bash
# Reading of variables
echo "Enter your name"
read name
# Display of variable values
echo "Your name is"
echo $name
```

Here, the code asks the user to enter their name, which is then displayed using the `read name` statement. And then that is displayed with `echo $name`.

After writing your script, press *Ctrl*+X to exit the editor. You will be prompted to save the file; press *y* to confirm, then press *Enter* to complete the save.

3. Next, change the permissions of your script by running:

```
chmod 755 interactive-script.sh
```

This command gives the file execute, read, and write permissions. You can verify that the permissions are set correctly.

4. To clear the terminal screen, use the `clear` command. Now, execute your script by typing:

```
./interactive-script.sh
```

When you execute this script, it prompts you for your name. You enter it, and the script displays it on the next line:

```
Enter your name
BIGDATA
Your name is BIGDATA
```

This demonstrates how your script accepts user input and displays it using variables.

Accepting user input makes a script more dynamic. As scripts grow, you can organize the actions they perform into reusable blocks by defining functions.

Defining and using functions in Bash scripts

Functions in Bash scripts group statements into reusable blocks that run when called. In the following example, you will create a new script that

defines a function named `display1` and calls it between two other commands:

1. Open a new file in the nano editor by typing:

```
nano function-script.sh
```

2. Type the following code into the editor. This script defines the `display1` function first and then calls it later in the execution sequence:

```
#!/bin/bash
display1() {
    echo "This is an example display1 function"
 }
 echo "This statement is before display1 function"
display1
echo "This statement is after display1 function"
```

3. Save the file and exit the editor. Since this is a new file, grant it execute permissions by running:

```
chmod 755 function-script.sh
```

4. Execute the script using:

```
./function-script.sh
```

   You will see the following three lines printed to your terminal, demonstrating how the script jumps to the function when it is called:

```
This statement is before display1 function
This is an example display1 function
This statement is after display1 function
```

This demonstrates how you can define a function, call it within your script, and control the flow of execution around function calls.

Functions help you reuse logic within a script. You can make the script itself more flexible by passing values to it when it is executed, which is where parameters and arguments come in.

Using script parameters and arguments

You can pass parameters or arguments to a Bash script, such as filenames or dates, and access them within the script using special variables. Within the script, these arguments or parameters are represented by the dollar sign ($)

and a number. So, `$0` indicates the script name, the first parameter is represented by `$1`, the second by `$2`, and so on.

For example, you can write the following statements in your script to display the script name and the first two parameters:

```
echo "Script name $0"
echo "First parameter is $1"
echo "Second parameter is $2"
```

So now let's move to the practical demonstration of passing parameters to a script:

1. Open your terminal and type the following command to create a new script file:

   ```
   nano parascript.sh
   ```

2. Type the following code into the editor to display the script names and the first two parameters:

   ```
   #!/bin/bash
   echo "Script name $0"
   echo "First parameter is $1"
   echo "Second parameter is $2"
   ```

   Save the file by pressing *Ctrl+X*, then *y*, and then *Enter*.

3. Change the permissions of your script to make it executable:

   ```
   chmod 755 parascript.sh
   ```

4. Run the script with two parameters, for example `abc` as the first parameter and `xyz` as the second parameter:

   ```
   ./parascript.sh abc xyz
   ```

   The output will be as follows:

   ```
   Script name ./parascript.sh
   First parameter is abc
   Second parameter is xyz
   ```

   From the output, you can see that the script name is `./parascript.sh`, the first parameter is `abc`, and the second parameter is `xyz`.

5. You can also run the script using `bash` and provide different parameters, such as `def` and `pqr`:

```
bash parascript.sh def pqr
```

The output will be as follows:

```
Script name parascript.sh
First parameter is def
Second parameter is pqr
```

So, here the script name is `parascript.sh`, the first parameter is `def`, and the second parameter is `pqr`.

This demonstrates how you pass parameters to a script and access them using special variables.

Passing parameters gives a script values to work with. The next step is to let the script make decisions based on those values, which you can do using Bash conditional statements.

Conditional statements in Bash

In this section, you will explore conditional logic using the `if-then-else` structure. Conditional statements allow a script to choose what to do based on the values or conditions it encounters. You will begin with `if` and `else`, combine conditions using Boolean operators, and then use `case` when a value may match several possible patterns.

The if statement

In this section, you will learn about the if statement.

Here is the syntax for it:

```
if condition
then
    statements
else
    statements
fi
```

Let's try a practical example where we check whether variable `a` is equal to variable `b`.

1. Create a new script file named `ifelsescript.sh` by typing:

```
$ nano ifelsescript.sh
```

2. Type the following code into the editor:

```
#!/bin/bash
a=10
b=20
if [ $a == $b ]
then
    echo " a is equal to b"
else
    echo " a is not equal to b"
fi
```

> **NOTE**
>
> In Bash, it is critical to include spaces inside the brackets around your variables and operators!

   The first statement is the shebang (`#!/bin/bash`), which indicates that this is a Bash script. You assign the values `10` and `20` to the variables `a` and `b`, respectively. The script then checks if `$a` is equal to `$b` using the `==` operator. If the condition is true, it prints "`a is equal to b`"; otherwise, it prints "`a is not equal to b`".

3. Save and exit the editor. Grant execute permissions to the script:

```
chmod 755 ifelsescript.sh
```

4. Now execute the script:

```
./ifelsescript.sh
```

Here, you will see the output `a is not equal to b`. Because the value of `a` is 10 and `b` is 20, the condition in the `if` statement is not satisfied, meaning the script executes the `else` block and displays this message.

This provides a practical demonstration of how the `if` and `else` statements work in a shell script when the equal condition is not met.

A single condition is enough for simple decisions, but scripts often need to evaluate several conditions together. Boolean operators let us combine or reverse those conditions.

Boolean expressions and logical operators

You now explore how Boolean expressions work in shell scripts, allowing you to evaluate whether a condition is true or false. The operators you use are as follows:

- `&&` for logical AND, which means the action is performed only if both conditions are true.
- `||` for logical OR, where the action is performed if at least one condition is true.
- `!` (the exclamation mark) for logical NOT, which performs the action only if the condition is false.

To see these operators in action, follow this demonstration using a script named `booleanscript.sh`:

1. Open a new script named `booleanscript.sh` using nano:

```
nano booleanscript.sh
```

2. Type the following complete code into the editor. This script demonstrates both the `&&` (AND) and `||` (OR) operators:

```
#!/bin/bash
a=10
b=20
if [ $a == $b ]
then
    echo "a is equal to b"
else
    echo "a is not equal to b"
fi
john=21
jesse=9
if [ $john -eq 21 ] && [ $jesse -eq 9 ]
then
    echo "john is 21 and jesse is 9"
elif [ $john -eq 21 ] && [ $jesse -eq 10 ]
then
    echo "john is 21 and jesse is 10"
fi
```

```
x=10
y=21
if [ $x -eq 10 ] || [ $y -eq 20 ]
then
    echo "yes x is 10 or y is 20"
else
    echo "x is not 10 and y is not 20"
fi
```

3. Save the file and exit the editor. Grant execute permissions to the script:

```
chmod 755 booleanscript.sh
```

4. Run the script using the following command:

```
./booleanscript.sh
```

You will see the following output:

```
a is not equal to b
john is 21 and jesse is 9
yes x is 10 or y is 20
```

This output confirms that the Boolean expressions and logical operators work as expected in your shell script.

Boolean operators make `if` statements more flexible when several conditions must be evaluated. When one value needs to be checked against several possible choices, however, a case statement often provides a cleaner structure.

Case statements for pattern matching

The `case` statement in Bash allows you to check a pattern or condition and execute a command if that condition is true. The syntax involves writing `case` followed by the expression, then listing patterns and their corresponding commands, and ending with `esac` (which is 'case' spelled backward).

Here is how it looks:

```
case expression in
    pattern1) execute commands;;
    pattern2) execute commands;;
    pattern3) execute commands;;
pattern4) execute commands;;
*)    execute some default commands or nothing;;
esac
```

Let's try this in action.

1. Create a new file named `casescript.sh` using nano:

```
nano casescript.sh
```

2. Type the following code into the editor. In this script, you declare a variable named `fruit` and use a `case` statement to match it against different fruit names:

```
#!/bin/bash
fruit=orange
case $fruit in
  apple)
    echo "apple"
    ;;
  mango)
    echo "mango"
    ;;
  orange)
    echo "orange"
    ;;
  banana)
    echo "banana"
    ;;
esac
```

3. Save the file and { XE "conditional statements, in `Bash:case` statements, for pattern matching" }exit the editor. Make the script executable by running:

```
chmod 755 casescript.sh
```

4. Execute the script:

```
./casescript.sh
```

   The output will be:

```
orange
```

   Since the value of `fruit` is `orange`, the corresponding command is executed and `orange` is displayed. This demonstrates how the case statement works for pattern matching in shell scripts.

Conditional statements control which commands a script executes. Another important part of program flow is controlling how often commands execute, which we do using loops.

Loop constructs in Bash

Loops in Bash allow you to automate repetitive tasks by executing a set of commands multiple times. You explore different types of loop constructs, including `for` loops, `while` loops, and `until` loops, each of which provides a way to repeat statements based on specific conditions or lists.

The for loop

The `for` loop operates on a list of items and repeats a set of commands for every item in that list. The general syntax for a `for` loop is as follows:

```
for variable in word1 word2 ... wordn
do
  statements to be executed
done
```

To see how a `for` loop works in practice, follow these steps:

1. Create a new file named `forscript.sh` using nano:

   ```
   nano forscript.sh
   ```

2. In this script, write the following code to display numbers from 1 to 10:

   ```
   #!/bin/bash
   for ((i=1; i<=10; i++))
   do
     echo $i
   done
   ```

3. Save the file and exit the editor. Make the script executable by running:

   ```
   chmod 755 forscript.sh
   ```

4. Execute the script:

   ```
   ./forscript.sh
   ```

   The output displays the numbers from 1 to 10 sequentially:



This demonstrates how a `for` loop iterates over a sequence and executes the specified statement for each value.

A `for` loop works well when the values or sequence to process are known in advance. When repetition instead depends on whether a condition remains true, a `while` loop is more suitable.

The while loop

A `while` loop in Bash evaluates a command or condition, and as long as the result is true, the loop continues to execute. When the command evaluates to false, the loop terminates.

The syntax for a `while` loop is as follows:

```
while command
do
  statements to be executed
done
```

For example, to display numbers in reverse order from 10 to 0, you can use the following script:

1. Open a new file named `whilescript.sh` using nano:



2. Write the following code into the editor:

```
#!/bin/bash
i=10
while [ $i -ge 0 ]
do
  echo "reverse order number $i"
  let i--
done
```

3. Save the file and exit the editor. Make the script executable by running:

```
chmod 755 whilescript.sh
```

4. Execute the script:

```
./whilescript.sh
```

This is the output:

```
reverse order number 10
reverse order number 9
reverse order number 8
reverse order number 7
reverse order number 6
reverse order number 5
reverse order number 4
reverse order number 3
reverse order number 2
reverse order number 1
reverse order number 0
```

This shows how a `while` loop works by executing the statements as long as the condition remains true.

Next, let's look at a loop that reverses this condition logic.

The until loop

The `until` loop in Bash provides a way to execute a set of statements repeatedly until a specified condition becomes true. Unlike the while loop, which continues as long as its condition is true, the `until` loop continues as long as its condition is false and stops once the condition evaluates to true.

The syntax for the `until` loop is as follows:

```
until [ condition ]
do
  # statements to be executed
done
```

Let's try a script to understand the working of this loop better:

1. Open a new file named `untilscript.sh` using `nano`:

```
nano untilscript.sh
```

2. Type the following code to demonstrate an `until` loop that prints numbers from 1 to 10:

```
#!/bin/bash
a=1
until [ $a -gt 10 ]
do
  echo $a
  ((a++))
done
```

3. Save the file and exit the editor. Make the script executable by running:

```
chmod 755 untilscript.sh
```

4. Execute the script:

```
./untilscript.sh
```

The output displays:

```
1
2
3
4
5
6
7
8
9
10
```

In this example, the loop continues to execute as long as the condition `[ $a -gt 10 ]` is false. Each time through the loop, the value of `a` is incremented by 1. When `a` becomes greater than 10, the condition evaluates to true and the loop terminates. So when the value of `a` is equal to `11`, the condition turns true and hence the loop stops. This demonstrates how the until loop works in Bash scripting.

Summary

In this chapter, you learned how to automate tasks using Bash shell scripting by building scripts with tools such as `nano`, `#!/bin/bash`, and commands such as `echo`, `chmod 777`, and `ls -ltr`.

Starting from the fundamentals of Bash as a shell, you moved through writing and executing simple scripts, explored features such as automation and error reduction, worked with interactive scripts that accept user input, defined and called functions, utilized script parameters (`$0`, `$1`, `$2`), and implemented conditional logic with `if else`, Boolean operators (`&&`, `||`, `!`), and pattern matching using `case`, before mastering loop constructs with `for`, `while`, and `until`. A key distinction we established is that while the `while` loop continues as long as its condition is true, the `until` loop runs until its condition becomes true, highlighting their complementary roles in script control flow. With these techniques, you built a practical foundation for automating repetitive tasks, processing input, and controlling execution logic directly from Bash scripts.

In the next chapter, you will consolidate your learning with a 20-question practice test covering Linux commands, file permissions, text editors, and Bash scripting, with answers and explanations to help you review your understanding

6

Practice Test

Congratulations on making it to the final chapter of this book! Over the previous five chapters, you have built a robust foundation in Linux—from setting up your Ubuntu environment and mastering core command-line operations to managing file permissions and automating repetitive tasks with Bash shell scripting.

To help you consolidate your learning and evaluate your readiness to apply these skills in real-world data engineering scenarios, this final chapter presents a comprehensive practice test.

This assessment consists of 20 multiple-choice questions covering the essential utilities, syntax, and scripting logic discussed throughout the book. Treat this as a self-guided review: take your time, read each question carefully, and try to recall the hands-on exercises we practiced.

For immediate feedback, the correct answer and a brief explanation are provided directly beneath each question. You may want to cover the answers as you read to truly test your knowledge. Good luck!

1. Which of the following is not a Linux distribution?

A. Ubuntu

B. CentOS

C. Windows 11

D. Debian

Correct Answer: **C**

Explanation: Windows 11 is not a Linux distribution. Ubuntu, CentOS, and Debian are Linux-based operating systems.

2. Which directory contains system configuration files?

A. `/home`

B. `/etc`

C. `/var`

D. `/usr`

Correct Answer: **B**

Explanation: `/etc` contains system-wide configuration files.

3. Which command is used to list files in Linux?

A. `show`

B. `list`

C. `ls`

D. `dirlist`

Correct Answer: **C**

Explanation: The `ls` command is the standard command used to list files and directories.

4. Which command is used to install software in Ubuntu?

A. `yum install`

B. `apt install`

C. `dnf install`

D. `installpkg`

Correct Answer: **B**

Explanation: Ubuntu uses the `apt` package manager for installing software.

5. What does the command `pwd` do?

A. Shows the user's password

B. Prints working directory

C. Changes directory

D. Prints system logs

Correct Answer: **B**

Explanation: The *pwd* command stands for **print working directory.**

6. Which directory contains user home directories?

A. `/root`

B. `/etc`

C. `/home`

D. `/bin`

Correct Answer: **C**

Explanation: All regular user home directories are located inside `/home`.

7. Which command is used to change file permissions?

A. `chmod`

B. `chperm`

C. `setperm`

D. `perm`

Correct Answer: **A**

Explanation: The `chmod` command changes file and directory permissions.

8. What does permission 755 mean?

A. Owner: read, write, execute | Group: read | Others: read

B. Owner: read, write, execute | Group: read, execute | Others: read, execute

C. Owner: read only | Group: write | Others: execute

D. All users: full permissions

Correct Answer: **B**

Explanation: 7 = rwx, 5 = r-x, 5 = r-x.

9. Which directory stores most system log files?

A. `/etc/log`

B. `/var/log`

C. `/home/log`

D. `/sys/log`

Correct Answer: **B**

Explanation: System log files are stored in `/var/log`.

10. Which editor is beginner-friendly and simple to use?

A. `vi`

B. `nano`

C. `emacs`

D. `vimdiff`

Correct Answer: **B**

Explanation: `nano` is simple and easier for beginners compared to `vi`.

11. In the `vi` editor, which key is used to enter insert mode?

A. `ESC`

B. `:w`

C. `i`

D. `x`

Correct Answer: **C**

Explanation: Press `i` to enter insert mode in `vi`.

12. Which symbol is used to represent the root directory?

A. `\`

B. `.`

C. `~`

D. `/`

Correct Answer: **D**

Explanation: `/` represents the root directory in Linux.

13. What is the default shell in Ubuntu?

A. `zsh`

B. `bash`

C. `sh`

D. `csh`

Correct Answer: **B**

Explanation: Ubuntu uses Bash (Bourne Again Shell) as the default.

14. Which line must appear at the top of a Bash script?

A. `#start bash`

B. `#!/bin/bash`

C. `#bash`

D. `#script`

Correct Answer: **B**

Explanation: `#!/bin/bash` is called the shebang and defines the interpreter.

15. How do you make a script executable?

A. `run script.sh`

B. `chmod +x script.sh`

C. `exec script.sh`

D. `enable script.sh`

Correct Answer: **B**

Explanation: `chmod +x` gives execute permission.

16. How do you access the first argument passed to a shell script?

A. `$0`

B. `$1`

C. `$arg1`

D. `$first`

Correct Answer: **B**

Explanation: `$1` represents the first command-line argument.

17. Which loop executes at least once even if the condition is false initially?

A. The `for` loop

B. The `while` loop

C. The `until` loop

D. None of the above

Correct Answer: **C**

Explanation: The `until` loop runs until the condition becomes true (opposite of `while`).

18. Which statement is used for multiple condition branching?

A. `if`

B. `switch`

C. `case`

D. `branch`

Correct Answer: **C**

Explanation: `case` is used for multi-condition branching in Bash.

19. Which command shows manual documentation of a command?

A. `help`

B. `man`

C. `doc`

D. `info`

Correct Answer: **B**

Explanation: The `man` command displays manual pages.

20. Which command displays the contents of a file?

A. `show`

B. `read`

C. `cat`

D. `open`

Correct Answer: **C**

Explanation: The `cat` command displays file content in the terminal.

Summary

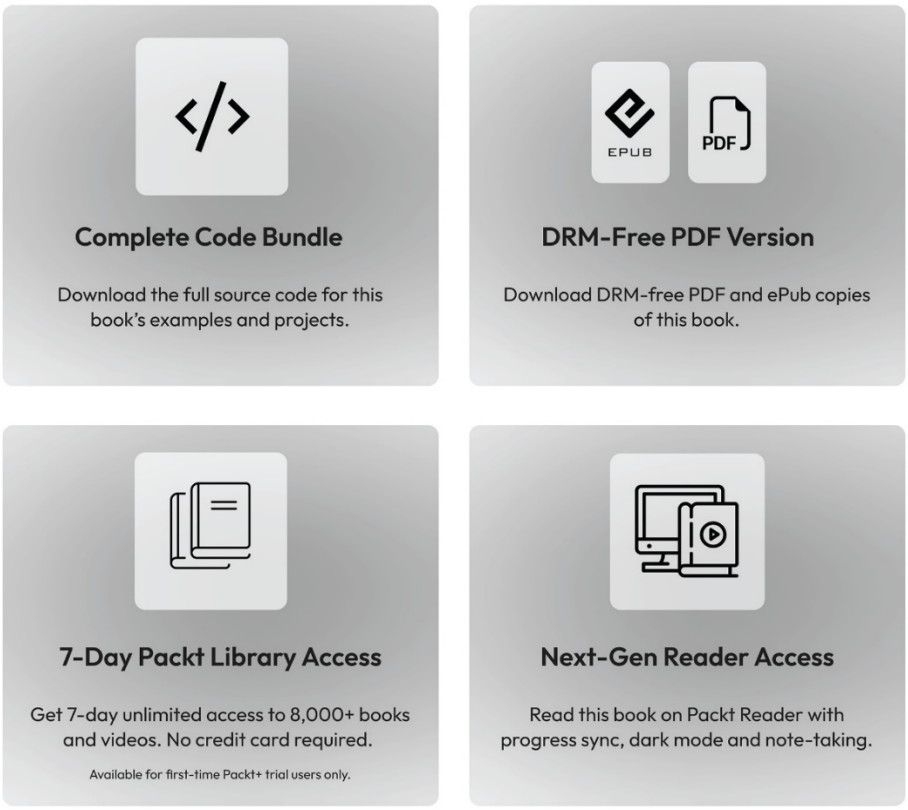
By completing this practice test, you have successfully reviewed and validated the core Linux concepts necessary for modern data engineering. Throughout this book, you have progressed from understanding basic file system navigation and software installation to mastering complex file permissions, text editors, and Bash shell scripting. The command-line utilities and automation techniques you have practiced here are not just academic exercises; they are the practical, everyday tools used to manage servers, process large datasets, and orchestrate data pipelines.

With this hands-on foundation now firmly established, you are fully equipped to leverage the power of Linux in your ongoing data engineering journey. Keep experimenting, keep scripting, and continue exploring the vast capabilities of the Linux environment.

7

Unlock the Code Bundle and the PDF Version

Your copy of this book includes the following exclusive benefits:



Follow this guide to unlock them. The process takes only a few minutes and only needs to be completed once.

Unlock this book's free benefits in three easy steps

Step 1

Have your purchase invoice ready for *step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit `https://www.packtpub.com/unlock-benefits/help`.

NOTE

Note: If you bought this book directly from the Packt website, no invoice is required. After *step 2*, you can access your exclusive content right away.

Step 2

Scan the QR code or go to `packtpub.com/unlock`.

On the page that opens (similar to *Figure 7.1* on desktop), search for this book by name and select the correct edition.

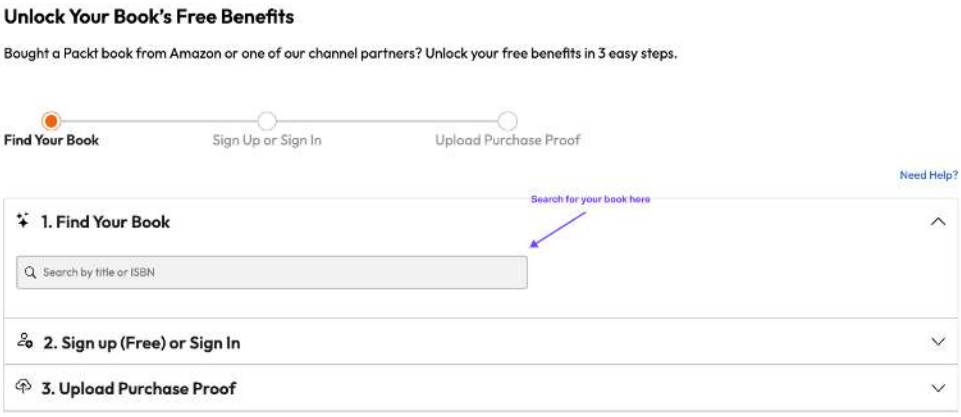


Figure 7.1: Packt unlock landing page on desktop

Step 3

After selecting the book, sign in to your Packt account or create one for free. Then, upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need help?

If you get stuck and need help, visit `https://www.packtpub.com/unlock-benefits/help` for a detailed FAQ on how to find your invoice and more. This QR code will take you to the help page.



NOTE

Note: If you are still facing issues, reach out to `customercare@packt.com`.



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

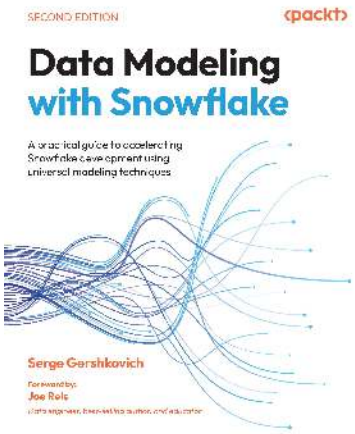
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

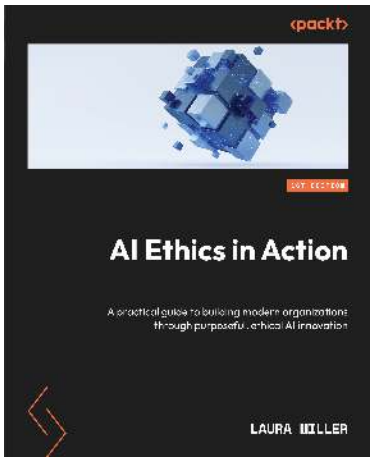


Data Modeling with Snowflake, Second Edition

Serge Gershkovich

ISBN: 9781837028030

- Master data modeling fundamentals to get your designs right on the first attempt with time-saving benefits
- Discover Snowflake's cloud-native architecture and unique features
- Apply modeling concepts to achieve higher efficiency by leveraging Snowflake architecture
- Get to grips with modeling concepts like normalization and slowly changing dimensions (SCDs)
- Read and transform semi-structured data, including hierarchies, using pre-built recipes and examples
- Explore organizational frameworks like Data Vault and Data Mesh
- Understand the business metrics that drive scalable, cost-effective data projects



AI Ethics in Action

Laura Miller

ISBN: 9781835469507

- Discover how to solve AI business dilemmas by applying the ethics of artificial intelligence.
- Develop and utilize ethical frameworks using measurable goals for responsible AI innovation.
- Future-proof your AI strategy with actionable risk mitigation techniques and guidance for navigating the evolving regulatory landscape.
- Understand the local and global implications of your AI decisions.
- Work with real-world use cases, examples, and best practices to navigate complex ethical scenarios.
- Recognize ethical improvements that meet the "Good Enough" standard.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packt.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Linux for Data Engineers*, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.



https://packt.link/r/1808493338

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

Symbols

/var/log
system log files 41

A

Advanced Package Tool (APT) **18**

B

Bash shell scripting
features 44
functions, using 50
fundamentals 44
interactivity 48, 49
parameters and arguments, using 50, 51
working, with Script files and permissions 46
writing and executing 44

Boolean expressions **53 – 55**

C

Case statements
for pattern matching 55 – 57

cat command **24**

cd command **21**

chmod command **36**
used, for practical examples 36, 37

conditional statements, in Bash **52**
Boolean expressions and logical operators 53 – 55
case statements, for pattern matching 55
if statement 52, 53

content
editing, with Nano editor 38, 39
editing, with vi/Vim editor 39, 40

cp command **23**

D

data engineering
Linux, role in 2

data manipulation
shell tools, using for 4

date command **29**

df command **29**

directories
commands, used for navigating 20 – 22

distros **3**

du command **29**

E

End of File (EOF) marker **32**

exit command **30**

extraction, transformation, and loading (ETL) workflows **2**

F

file management system
chmod command 36
permissions 36
practical examples, using chmod command 36, 37

files
commands 22 – 27

for loop **57, 58**

G

graphical user interface (GUI) **3**

grep filter **25**

gzip command **26**

H

head command **25**

history command **30**

I

if statement **52, 53**

L

Linux
distributions and their characteristics 3, 4
role, in data engineering 2

Linux file system
exploring, with graphical interface and terminal 12 – 16
hierarchy structure 10 – 12

less command **24**

logical operators **53 – 55**

loop constructs, in Bash **57**
for loop 57, 58
until loop 60, 61
while loop 58

ls command **21**

M

man command **19**

manual page (man page) **19**

mkdir command **22**

mv command **23**

N

Nano editor
installing 19
used, for editing content 38, 39

P

pattern matching
Case statements for 55 – 57

process management **28**

ps command **28**

pwd command **21**

R

Red Hat Enterprise Linux (RHEL) **4**

rm command **24**

rmdir command **22**

S

shell tools
using, for data manipulation 4

software
installing 18

system log files
in /var/log 41
viewing 41, 42

system management **28**

system navigation
commands 20 – 22

T

tail command **25**

tar utility **27**

terminal keyboard shortcuts **31, 32**

top command **28**

touch command **23**

U

Ubuntu environment **8**
Native installation 9
VirtualBox Virtual Machine 9
WSL, recommended for Windows Users 8

until loop **60, 61**

V

vi/Vim editor
used, for editing content 39, 40

virtual machine (VM) **9**

W

Windows Subsystem for Linux (WSL)
for Windows Users 8

wc command 26
while loop 58

Z

zip command 27

Untitled

1. Preface
    1. Free benefits with your book
2. Chapter 1: Introduction to Linux
    1. Technical requirements
    2. Role of Linux in data engineering
    3. Linux distributions and their characteristics
    4. Common shell tools for data manipulation
    5. Summary
3. Chapter 2: Setting Up Your Linux Environment
    1. Installing Ubuntu: choosing your environment
        1. Option 1: Windows Subsystem for Linux (WSL) - recommended for Windows users
        2. Option 2: VirtualBox Virtual Machine
        3. Option 3: Native installation (dedicated machine or dual-boot)
    2. Understanding the Linux file system hierarchy structure
    3. Exploring the file system using the graphical interface and the terminal
    4. Summary
4. Chapter 3: Mastering Basic Linux Operations and Software Installation
    1. Technical requirements
    2. Installing software
    3. Practical example: installing Nano
        1. Manual pages
    4. Basic operations
        1. System navigation and directories
        2. File creation and manipulation

    5. Viewing and searching file content
    6. Archiving and compression
    7. System and process management
    8. Essential terminal keyboard shortcuts
    9. Summary
5. Chapter 4: File Management Permissions and Text Editors
    1. Technical requirements
    2. File system permissions and the chmod command
        1. 1.Who gets access? (ownership)
        2. 2.What kind of access? (permissions)
        3. 3.Practical examples using chmod
    3. Editing content with the Nano editor
    4. Editing content with the vi/Vim editor
    5. System log files in /var/log
        1. Viewing system logs
    6. Summary
6. Chapter 5: Automating Tasks with Bash Shell Scripting
    1. Technical requirements
    2. Fundamentals of Bash shell scripting
    3. Features of shell scripts
        1. Writing and executing a simple Bash script
        2. Working with script files and permissions
    4. Interactive shell scripting
    5. Defining and using functions in Bash scripts
    6. Using script parameters and arguments
    7. Conditional statements in Bash
        1. The if statement
        2. Boolean expressions and logical operators
        3. Case statements for pattern matching
    8. Loop constructs in Bash
        1. The for loop
        2. The while loop

        3. The until loop
    9. Summary
7. Chapter 6: Practice Test
    1. Summary
8. Chapter 7: Unlock the Code Bundle and the PDF Version
    1. Unlock this book's free benefits in three easy steps
9. Other Books You May Enjoy
10. Index