

Recommended for Freshers

OBJECTS, DATA & AI

Build Thought process to develop Enterprise Applications

Reeshabh Choudhary

Reeshabh Choudhary

[Objects, Data & AI](#) © 2023 by [Reeshabh Choudhary](#) is licensed under [CC BY-NC-SA 4.0](#)





या कुन्देन्दुतुषारहारधवला या शुभ्रवस्त्रावृता
या वीणावरदण्डमण्डितकरा या श्वेतपद्मासना।
या ब्रह्माच्युत शंकरप्रभृतिभिर्देवैः सदा वन्दिता
सा मां पातु सरस्वती भगवती निःशेषजाङ्गापहा॥

*(Salutations to Devi Saraswati) Who is pure white like jasmine, with the coolness of the moon,
that shines like a garland of snow and pearls; And who is covered with pure white robes,
Whose hands are adorned with veenas and boons; And who sits on a pure white lotus,
One who is always the adoration of Bhagwan Brahma, Bhagwan Vishnu, Bhagwan Shankar and other
deities,
O Devi Saraswati, please protect me and remove my ignorance completely.*

Acknowledgements

I bow down to my deities *Maa Saraswati, Ganesh ji and Hanuman ji* for guiding me to pursue knowledge and giving me strength to deal with challenges on the way. My regards to my late *Nanaji* (maternal grandfather) Mr. Surendra Kumar Jha, who had been my rock till the time he breathed. My childhood is incomplete without him and his stories. He made me fall in love with stories and evening walk!

My journey as a student of Computer Science and Life would not have been complete without my Guru Mr. Rupam Das, founder and CEO of Lyfas. For past 12 years, he has been guiding me time to time, devoting countless hours to resolve my queries, being available for me out of nowhere, when I needed him the most. He listens to my stupid questions again and again and does not give up until I am satisfied. He guides me with suggestions, helps me formulate road maps, calms me down when I am in a chaotic mood, and probably understands me better than anyone else on this Earth. A student must always seek a Guru (teacher) as per Hindu philosophy, and I can't thank my deities enough for crossing our paths in my college days.

Writing or compiling a book can be a tiring process. It requires a significant research, dedicated long hours, and tons of patience. Working on a book requires to be in a zone of itself. One has to be focused, dedicated enough to keep the momentum going to achieve the set targets. Everything else takes a backseat. This journey would have been incomplete without my life partner Richa Thakur. Ever since, she has come to my life, she has brought a sense of innocence, childishness, love, and kindness around me. Journey of life with you by my side has been beautiful!

I extend my regards to Mr. Saurabh Ranjan Singh (we worked together at Schlumberger India), who has been guiding me since 2016 in the course of my journey as a Software Developer. He has been patient enough to listen out my queries and help me understand the fundamentals behind many technologies. He is one of the soundest software developer one can come across.

Mr. Gurpreet Singh, Director of Engineering at PersonaTech, has been a source of strength since 2017. He is an experienced Software Architect and gives fitness training. We met at his gym, and since then, he has been generous enough to help me with my fitness program as well as technical skills in programming. He has been available for discussing even vague ideas at oddest of hours and listen out patiently everything, even though none of it would make sense. He is one of the calmest persons I have ever come across.

Family is the backbone of India society and I have been blessed to have Mr. Amar Nath Jha (maternal uncle), who have been a source of strength at the darkest hours of my life.

This journey would not have been completed if I have had not exposure to work on various projects with some esteemed organizations. I shall always be grateful enough to Mr. Ashish Agarwal for giving me the opportunity to express myself in the work I do and backing my judgement. He is one of the rare bosses with whom I can share my thoughts. I consider this my privilege to have worked with him on multiple projects.

I gently bow down to my teachers at school (Centra Public School, Samastipur, Bihar) and college (P.D.A. College of engineering, Kalburgi, Karnataka) for teaching me the value of education. I pay my regards to my college seniors, especially Mr. Saurabh Shekhar Jha and Mr. Sumit Pandey for guiding me at various phases of life.

Reeshabh Choudhary

Sports is an emotion and I have embodied this emotion right from childhood. I can't express my love enough to my friends with whom I grew up playing, at various stages of life. To learn to pass a ball at the right time is art of life! It requires you to trust the other human. Trust is an emotion and humanity thrive on trust. If you understand its value, you can successfully build and run organisations. I thank my numerous teammates for trusting me at crucial junctures of a game.

Life is to be guided by philosophy, driven by passion, and lived with joy. I pay my regards to Rishi & Muni (Saints) of Hindu civilization, who have been compiling wisdom and knowledge from very ancient times and passing it on generation to generation in the form of *shruti* and *smriti* of Vedas, Puranas, and literary epics. I am grateful to the publishers of Hindi story books like Chanda Mama, Nandan, Nanhe Samrat, Champak, Raj Comics, etc. for igniting the passion for stories in my life and shaping my imagination during childhood. I bow down gently to writers Ram Dhari Singh 'Dinkar', Maithli Sharan Gupt, Harivansh Rai Bachchan, Charles Dickens, Dan Brown, J.K. Rowling, Stieg Larsson, George Orwell, Ruskin Bond, Sir Arthur Canon Doyle, Amor Towles, Ernest Hemmingway, Vikram Sampath, Paulo Coelho, Walter Isaacson and Robert Greene, whom I grew reading in my 20s and 30s and their works have had profound effect on shaping my thought process and making me fall in love with life and kept me going at the low points of life. Cinema is another powerful medium of storytelling which I have thoroughly enjoyed experiencing. I have grown up reading story books, playing in the fields and watching movies. Movie directors Christopher Nolan, Ridley Scott, Alfred Hitchcock, Steven Spielberg, David Fincher, and Denis Villeneuve have always fascinated me with their imagination, vision, and philosophy towards life. Their work needs to be celebrated and studied with great care.

And I am always grateful to my parents for bringing me to life!

For Maithili!

Preface

Today we are living the Age of Data. Data is the king. If you have data, you can extract information, behavioral patterns, manipulate behaviors, create successful businesses and what not. This book is not about specialization in a particular sect of Computer Science. This book is about setting the philosophical context of how digital age has evolved and as a newcomer in this industry, how you can connect dots with few of the technologies around you. With so many technologies evolving around every single day, with new breaches in innovation in the field of AI/ML or Data Science, which gets the job done in a whisker, as programmers we tend to think, where do we stand. The journey or even the thought of making sense of everything around us can be quite overwhelming. From the days of C/C++ programming to Java/C#/JavaScript and Python/MATLAB/R, programming has exponentially evolved. And so, does the computational ability of computers, which also helped in faster execution of these programs, but also to extraction of Information from the data generated via the applications developed by these programs. In this digital age, everything seems to be connected and yet we sweat making sense of all these connections.

In the interconnected digital age, understanding the connections between various technologies can be challenging. The book aims to bridge some of these gaps by providing readers with a foundational understanding of how programming, data, and machine learning are interconnected. By grasping these fundamentals, software developers can connect the dots according to their specific requirements.

The book also acknowledges the concerns and thoughts about the future of programmers and programming. It recognizes the potential impact of generative AI capabilities that may threaten to displace certain aspects of human workforce in the near future. By addressing these concerns and providing insights into the interconnections of programming, data, and machine learning, the book aims to equip readers with the knowledge and understanding necessary to navigate the evolving landscape of software development.

Overall, the book's goal is to empower readers with a basic understanding of how programming, data, and machine learning intersect and how they can leverage this knowledge as software developers to meet their specific requirements in the digital age.

I must admit that I am a mere compiler of the knowledge about various topics present in this book. There are already many fine resources present on the topics covered in this book, I have just tried to connect the dots and presented them to the reader from my perspective.

References

Throughout human history, books have been the biggest source of knowledge. A good book inspires countless other books. A student reads different books on same topic to understand different perspectives. I am no different. Compilation of this book would have been incomplete without numerous books and online source materials present. Although, it would not be possible to mention each and every source, from where I have gathered information, I would definitely like to mention some key sources which have served as the axis of this book.

1. Designing Data-Intensive Applications – Martin Kleppmann
2. Software Architecture Pattern – Mark Richards
3. Patterns of Enterprise Application Architecture – Martin Fowler
4. Head First Design Patterns – Eric Freeman & Elizabeth Freeman
5. Java, the complete reference – Herbert Schildt
6. Fundamentals of Database Systems – Ramez Elmasri, Shamkant B. Navathe
7. Database system concepts – Henry F. Korth, Abraham Silberschatz, S. Sudarshan
8. Dive into Design Patterns – Alexander Shvets
9. Agile Software Development, Principles, Patterns, and Practices – Robert Martin
10. Gang Of Four - Design Patterns, Elements Of Reusable Object Oriented Software – Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides
11. Python Object-Oriented Programming – Steven F. Lott, Dusty Phillips
12. History of computer science – Dodig-Crnkovic G.
13. Machine Learning – Thomas Mitchell
14. Microservices Patterns – Chris Richardson
15. Practical Microservices Architectural Patterns – Binildas Christudas
16. Feature Engineering for Machine Learning – Alice Zheng, Amanda Casari
17. The Art of Feature Engineering – Pablo Duboue
18. Modern Deep Learning for Tabular Data – Andre Ye, Zian Wang
19. Hands-On Machine Learning – Aurélien Géron
20. Machine Learning for Absolute Beginners – Oliver Theobald
21. Transformers for Machine Learning – Uday Kamath, Kenneth L. Graham, Wael Emara
22. Kaggle resources
23. Analytics Vidhya
24. Google Garage Series
25. Open AI Documentation
26. LangChain Documentation

What to expect from the book?

The ultimate goal of the book is to help readers develop thought process for designing **modern-day enterprise applications**. The **first section** will cover introduction to **object-oriented programming** and architecture and design patterns, allowing readers to grasp the foundations of building enterprise applications.

Moving on to the **second section**, the book will delve into **database** systems that cater to the current needs of data storage and retrieval. It will explore different types of databases, their structures, and techniques for efficient data management. This section will equip readers with the knowledge necessary to design and implement robust data storage solutions for enterprise applications.

Finally, in the **third section**, the book will introduce readers to the captivating world of **artificial intelligence (AI)**. It will focus on machine learning and deep learning, providing introductory concepts and techniques. This section aims to familiarize readers with AI principles and empower them to apply these concepts to enhance their enterprise applications.

Overall, the end goal of the book is to equip readers with the skills and knowledge needed to develop sophisticated enterprise applications that leverage object-oriented programming, utilize efficient database systems, and incorporate artificial intelligence techniques. By covering these essential topics, the book aims to bridge the gap between theoretical understanding and practical implementation, enabling readers to create innovative and cutting-edge software solutions.

What not to expect from this book

This book is not a tutorial to learn programming. Reader is expected to be familiar with basic programming concepts. This book focusses on building thought process for designing enterprise level software applications. Hence, the code examples and its details are going to be abstract and at times, readers are left to figure it out themselves.

DISCLAIMER 1

I shall not be going into the details of programming languages and syntax. Idea is to present you with the instinct to write your own code in any language of your choice. For my convenience, I will be using Python or Java programs as per requirement.

DISCLAIMER 2

I am not an expert in any of the programming language. As a developer, I have never stuck to one programming language. I started my career with C# .Net, later worked with JavaScript frameworks and then switched over to Java Spring Boot, and in recent years I have worked on Scala and Python.

In my understanding, programming languages are just the tools to tell your computer at your convenience what actions to perform. Programming languages come with their own advantages and limitations. As a software developer, we should be able to understand their use case as required. In practical scenarios, this is also one of the reasons for having microservices designed in multiple programming languages, with languages chosen as per their merit to perform specific tasks. For example, Node JS might be preferred by some architects to handle concurrency over Java Spring Boot.

Hence, this book focuses on developing the thought process of a curious software developer, who can figure out the missing pieces and connect the dots in the real-world programming.

Contents

References.....	9
What to expect from the book?.....	10
What not to expect from this book	10
DISCLAIMER 1	10
DISCLAIMER 2	10
Contents	12
Chapter 1	20
Setting the context ... By turning a page of History	20
1.1. Introduction.....	20
1.2. What is Programming?	21
1.3. History of Programming	21
1.3.1. World's first Programmer: Ada Lovelace.....	21
1.3.2. How Humans pass command to Computers?.....	22
1.4. Story of Databases.....	23
1.4.1. History of DATABASES	24
1.4.2. Continuous advancements in the field of Data Storage.....	27
1.5. Machine Learning and AI	27
1.5.1. History of ML.....	28
1.6. Summary.....	31
Chapter 2	34
Object Oriented Programming.....	34
2.1. What is a Program?	34
2.2. Pillars of Programming	34
2.3. Objects	36
2.4. Classes	37
2.5. Programmatic representation of Class	37
2.6. Important Concepts	39
2.6.1. Abstraction	39
2.6.2. Encapsulation	39
2.6.3. Inheritance.....	41
2.6.4. Polymorphism	43
2.7. Summary: Encapsulation, Inheritance and Polymorphism.....	44
Chapter 3	46
Design Patterns in Object Oriented Programming.....	46

3.1. Design Patterns: Introduction.....	46
3.2. Design Patterns: Mandate or Necessity?	46
3.2.1. Limitation of Inheritance	50
3.2.2. Composition.....	51
3.3. SOLID Principles.....	51
3.3.1. The Single Responsibility Principle.....	51
3.3.2. Open Close Principle	52
3.3.3. Liskov Substitution Principle.....	54
3.3.4. Interface Segregation Principle	55
3.3.5. The Dependency-Inversion Principle.....	56
3.4. Cataloguing Design Patterns.....	60
3.4.1. Creational Patterns	60
3.4.2. Structural Patterns	61
3.4.3. Behavioral Patterns.....	61
3.5. Summary.....	61
Chapter 4	63
 Creational Design Patterns	63
4.1. Factory Pattern	63
4.2. Abstract Factory Pattern.....	66
4.3. Builder Pattern.....	68
4.4. Prototype Pattern.....	72
4.5. Singleton Pattern.....	74
4.6. Summary.....	75
 Chapter 5	76
 Structural Design Patterns	76
5.1. Adapter Pattern	76
5.2. Bridge Pattern	80
5.3. Composite Pattern	84
5.4. Decorator Pattern	87
5.5. Facade Pattern	92
5.6. Flyweight Pattern.....	95
5.7. Proxy Pattern	99
5.8. Summary.....	103
 Chapter 6	105
 Behavioral Design Patterns.....	105

6.1. Chain of Responsibility	105
6.2. Command Pattern	110
6.3. Iterator Pattern	113
6.4. Mediator Pattern	118
6.5. Memento Pattern	122
6.6. Observer Pattern	127
6.7. State Pattern	132
6.8. Strategy Pattern	137
6.9. Template Method Pattern	141
6.10. Visitor Pattern	146
6.11. Summary	151
Chapter 7	156
Database Systems	156
7.1. Database Management System	156
7.1.1. Queries and Transaction	157
7.1.2. Data Abstraction	157
7.1.3. Instances and Schema	159
7.2. Database Languages	159
7.2.1. Data Definition Language (DDL)	159
7.2.2. Data Manipulation Language (DML)	160
7.3. Database Internals	160
7.3.1. Storage Manager	161
7.3.2. Query Processor	162
7.3.3. Transaction Manager	163
7.4. Distributed Databases	166
7.4.1. Distributed Database Management System (DDBMS)	167
7.4.2. Database Replication	168
7.4.3. Database Sharding	172
7.4.4. Concurrency Control	175
7.4.5. Transaction Management	176
7.4.6. Distributed Recovery	178
7.4.7. Distributed Query Management (DQP)	180
7.5. In-Memory Databases	182
7.5.1. Benefits	182
7.5.2. Challenges	182

7.5.3. Use Cases.....	183
7.5.4. In-Memory Databases in Distributed Architecture	184
7.6. SUMMARY	184
Chapter 8	188
Data Storage	188
8.1. Physical Storage Mediums	188
 8.1.1. Primary Storage	188
 8.1.2. Secondary Storage.....	189
 8.1.3. Tertiary Storage	189
 8.1.4. RAID	190
 8.1.5. Storage Area Networks.....	192
 8.1.6. Network-Attached Storage.....	193
 8.1.7. Object-Based Storage	193
8.2. Storage Structures	194
 8.2.1. File Storage	194
8.3. Indexing	206
8.4. Summary.....	212
Chapter 9	214
Data Models.....	214
9.1. Designing Data Models.....	214
9.2. Relational Data Models	215
 9.2.1. Inception of Relational Data Models.....	215
 9.2.2. Structuring of Relational Data Models	216
 9.2.3. Query Language.....	218
9.3. NoSQL Data Models	220
 9.3.1. Inception of NoSQL	220
 9.3.2. Different NoSQL Systems	221
 9.3.3. NoSQL & CAP Theorem.....	222
9.4. SQL (Relational) versus NoSQL (Non-Relational)	223
9.5. Convergence	225
9.6. Summary.....	225
Chapter 10	229
Enterprise Architecture.....	229
10.1. Layered Architecture.....	230
 10.1.1. Pattern Description	230

10.1.2. Pattern Analysis	234
10.2. Microservices Architecture	234
10.2.1. Pattern Description	236
10.2.2. Pattern Analysis	238
10.3. Event Driven Architecture	240
10.3.1. Pattern Description	242
10.3.2. Pattern Analysis	245
10.4. Summary.....	245
Chapter 11	249
Rise of Big Data and Machine Learning.....	249
11.1. From static web pages to Social Media	249
11.2. Empowering the Data-Driven Enterprise.....	249
11.3. Characteristics of Big Data	250
11.4. Storage Systems for Big Data.....	252
11.5. Unravelling the Synergy: Big Data Analytics and Machine Learning.....	252
11.6. Anatomy of Machine Learning: Unveiling the Core of Artificial Intelligence	254
11.6.1. Machine Learning and Data Mining	255
11.7. Summary.....	255
Chapter 12	258
Machine Learning: A Brief Introduction	258
12.1. Stepping away from Explicit Programming	258
12.2. Data & Models.....	260
12.3. Classification of Machine Learning Systems.....	261
12.3.1. Learning Strategies	261
12.3.2. Knowledge Representation.....	271
12.3.3. Application Domain	271
12.4. Challenges with Machine Learning	272
12.4.1. Challenges Based on Data	272
12.4.2. Challenges Based on Algorithms	272
12.5. Summary.....	273
Chapter 13	275
Feature Engineering	275
13.1. Scalers and Vectors	276
13.1.1. Scalers	276
13.1.2. Vectors	276

13.2. Data Visualization and Analysis	277
13.3. Dealing with Numerical Data.....	279
13.3.1. Binarization	281
13.3.2. Binning.....	282
13.3.3. Feature Scaling.....	284
13.4. Handle Missing Data.....	289
13.5. Dealing with Text Data.....	291
13.5.1. Bag-Of-Words	291
13.5.2. Bag-Of-N-Grams.....	293
13.5.3. Filtering Techniques	297
13.5.4. Parsing and Tokenization.....	297
13.5.5. TF-IDF Approach	298
13.6. Dealing with Categorical Variables.....	300
13.7. Dimensionality Reduction using PCA	304
13.7.1. Mathematical explanation of PCA	305
13.7.2. PCA Implementation	306
13.7.3. PCA Visualization	310
13.7.4. PCA Limitations.....	314
13.8. Working with Images	314
13.8.1. SIFT.....	317
13.8.2. CNN.....	319
13.9. Summary.....	320
Chapter 14	323
Deep Learning & Neural Network	323
14.1. Human Brain as Inspiration	323
14.2. Perceptron	324
14.3. Feed Forward Neural Network.....	326
14.4. Neural Network Use Case.....	329
14.5. Deep Learning	335
14.6. Computer Vision model using CNN	336
14.7. Summary.....	344
Chapter 15	348
Rise of Gen-AI.....	348
15.1. Evolution of Deep Learning: from predicting to generating data	348
15.2. Encoder-Decoder.....	350

15.3. Transformers: A step towards Language Models	352
15.4. Generative AI	356
15.4.1. Large Language Models (LLMs)	358
15.4.2. Challenges of training LLMs	359
15.4.3. When to go for LLM Training?	361
15.4.4. Prompt Engineering	362
15.4.5. LLM Configuration	364
15.4.6. Gen AI Project Lifecycle	365
15.4.7. Fine Tuning LLMs	366
15.4.8. LLM use cases with Enterprises	375
15.4.9. LangChain: Bridge between LLMs and Software Development	376
15.4.10. A Sample Use Case: Document Retrieval and Question Answering	376
15.5. Road Ahead	381
15.6. End Note	382

Chapter 1

Setting the context ... By turning a page of History

“ “ *If you don't know history, then you don't know anything. You are a leaf that doesn't know it is part of a tree.* ” - Michael Crichton

1.1. Introduction

History of Humans and their evolution goes back to 70,000 years ago. In this course of time, our ancestors have witnessed major changes in the Earth eco system. Some organism got extinct; some new breeds were discovered. People migrated to new places, settled new colonies. Humans achieved major mathematical and scientific breakthroughs as their ever-evolving requirements grew, they kept on innovating and researching. Science has always been the biproduct of requirements of Human society. That is how, most of the major inventions of modern Human era came into existence, as scientists tried to solve various problems of their specific domains, driven with the goal of making a salient contribution to the lives of human society. This evolution of human cognitive function has been in pace with the course of time and progress of society, until the last century.

Leonardo Da Vinci (1452-1519) was convinced that humans would be able to fly like birds. He had this insight some 200 years before the actual aero-plane model was conceived. Benjamin Franklin would not very surprised if he would discover large locomotives and machines operating on electrical circuits.

However, I can bet they would be more than surprised, even fascinated, and overwhelmed, if they got to experience what Computers can do today. The advent of Digital age has grown into many folds exponentially. Today, we can order groceries at our home, book flight tickets around the globe, oh ... wait, we can even experience a place without even visiting it, thanks to Augmented Reality and Meta verse. To cut it short, if you are a 90s kid, you are the bridge that has seen the crossover of Sci-fi films into the reality of Daily Lives of average human.

So how did computers evolve so quickly and manage to do all these stuffs? It was in only 1944 when Howard H. Aiken (1900-1973) built the Mark I electromechanical computer, with the assistance of IBM at Harvard. Military code breaking capabilities in world war II led to various computational projects. Some scientists started building computers to solve mathematical equations by 1950s. However, it was in 1962, when Computer Science as a discipline came into existence at Purdue University. This led to assembling of various theoretical branches of Computer applicability under one roof. Decade of 1960s saw an unprecedented development in the field of computer science. Operating systems made major advancements. New programming languages such as BASIC were invented. Intel designed the first microprocessor (computer on a chip) in 1969-1971. 1970s saw major advances in the field of data management. Operating System like Linux and supercomputers like CRAY-1 were designed. Computational ability of computers grew significantly. A major breakthrough came in 1981 when first portable computer Osborne I was marketed by IBM, however, personal computers became a revolution thanks to Steve Wozniak and Steve Jobs, founders of Apple Computer. And since then, there is no turning back, and by 1990s with boom of internet, the landscape of computer applicability changed for good. Computers transitioned from

computational machines to daily usage machines, adapted by businesses, writers, salesperson, students, etc. Today computers are not just medium of storing online documents or automating a business flow, but they have become a mode of communication, thanks to internet.

Computer Science as a discipline grew many folds. New programming languages came into picture, which helped develop break through applications for enterprise business. Their applicability relied on programming tasks, storing information and their retrieval (database) and making sense of stored information (machine learning). Let us briefly look at the history of these three aspects, how they evolved in parallel and yet kept converging.

1.2. What is Programming?



Programming involves the development of software and algorithms to instruct computers. Programming is the process of creating computer programs or software by writing sets of instructions that a computer can execute. It involves designing, coding, testing, and maintaining computer programs to solve specific problems or perform desired tasks.

At its core, programming is about giving precise instructions to a computer to perform a series of operations. These instructions are written using programming languages, which serve as a medium of communication between humans and computers. Programming languages vary in syntax and capabilities, but they all provide a set of rules and structures to express computations and manipulate data. Programmers use various tools and integrated development environments (IDEs) to write, edit, test, and debug their code. They use programming concepts such as variables, loops, conditionals, functions, and data structures to create efficient and maintainable programs.

The applications of programming are vast and diverse. It is used to develop software applications, websites, mobile apps, video games, artificial intelligence systems, databases, and much more. In today's interconnected world, programming has become an essential skill, empowering individuals, and businesses to leverage the power of computers to automate tasks, process data, and create innovative solutions.

1.3. History of Programming

1.3.1. World's first Programmer: Ada Lovelace

Growth of Computer Science in last century has been rapid fast with mathematics adding wings to its flight. In the early days of computing, the focus was primarily on developing coding languages and algorithms to perform calculations and solve mathematical problems. Ada Lovelace's collaboration with Charles Babbage on the Analytical Engine in the 19th century laid the foundation for modern programming. In 1843, Lovelace wrote an algorithm for the Analytical Engine, which is considered the first computer program. She recognized the potential of numbers and their ability to represent more than just numerical values, envisioning that machines could be programmed to perform complex calculations and manipulate symbols. Lovelace's algorithm for computing Bernoulli numbers showcased the concept of step-by-step instructions that could be executed by a machine, serving as a blueprint for modern programming. Her visionary ideas about the capabilities of computing machines and the significance of programming were ahead of their time.

Although the Analytical Engine was never built during Lovelace's lifetime, her work and insights paved the way for future advancements in programming. Her contributions to the field earned her the title of the world's first programmer.

Since Ada Lovelace's era, programming languages have evolved significantly. They have become more powerful, expressive, and specialized, allowing humans to interact with computers more efficiently and effectively. Today, there is a wide range of programming languages available, each designed to address specific needs and cater to different paradigms of programming.

The development of programming languages has not only focused on mathematical computations but has expanded to encompass various domains such as web development, data analysis, artificial intelligence, and more. Modern programming languages offer a rich set of features, libraries, and frameworks that empower developers to create sophisticated software applications, from simple scripts to complex systems.

1.3.2. How Humans pass command to Computers?

 **Assembly language** is a low-level programming language that bridges the gap between machine code (binary instructions directly understood by a computer's hardware) and higher-level programming languages.

It uses mnemonic instructions that represent specific machine operations and memory locations. It came to prominence in 1949. In assembly language, each mnemonic instruction corresponds to a specific machine instruction. These instructions are then converted into machine code using an assembler, a specialized program that translates assembly language into machine code. Assembly language provides direct access to the underlying hardware of a computer, allowing programmers to write code that can take full advantage of the computer's capabilities. It is often used in situations where performance is critical or when direct hardware control is required, such as in embedded systems, device drivers, and operating system kernels.

Alick Glennie's Autocode is considered by some to be the first compiled computer programming language. Autocode was developed in the early 1950s at the University of Manchester in England. Autocode was designed to simplify programming for the Manchester Mark 1 computer, an early computer system. It allowed programmers to write high-level instructions that could be translated directly into machine code. The key innovation of Autocode was its ability to be compiled. **Compilation** is the process of translating a program written in a high-level language into machine code that can be executed directly by the computer's hardware. Autocode was one of the earliest languages to introduce this concept. Autocode eliminated the need for manual translation of instructions into machine code, making programming more efficient and less error prone. It allowed programmers to express their instructions in a higher-level language, which was then transformed into machine code by the compiler.

The development of Autocode marked a significant advancement in programming languages, as it demonstrated the potential for translating high-level instructions into machine code automatically. This laid the foundation for future compiled languages and paved the way for the development of more sophisticated programming languages.

The programming languages, such as FORTRAN (1957, John Backus) and COBOL (1959, Dr. Grace Murray Hopper), were developed to facilitate scientific and business computations respectively. These early languages introduced the concepts of control flow, variables, and subroutines, laying the foundation for subsequent programming languages. FORTRAN, in particular, gained widespread adoption and was instrumental in advancing the idea of high-level languages that could be compiled.

In the 1960s and 1970s, procedural programming languages, such as ALGOL (1960) and C (1972), gained prominence. Procedural programming focused on breaking down complex problems into smaller, manageable procedures or functions. It introduced concepts like loops, conditionals, and modularization,

improving code organization and reusability. Procedural programming languages remained popular for several decades and are still widely used today.

The 1980s witnessed a significant paradigm shift with the emergence of object-oriented programming. Languages like Simula (1967) and Smalltalk (1972) laid the groundwork for OOP, but it was the introduction of C++ (1983) and later Java (1995, originally intended to be used with hand-held devices) that popularized the paradigm. OOP emphasized the organization of code around objects, encapsulating data, and behavior together. This approach provided modularity, reusability, and the ability to model real-world entities effectively.

Functional programming, rooted in mathematical concepts, gained traction in the 1970s and 1980s. Languages like Lisp (1958) and ML (1973) introduced functional programming concepts such as higher-order functions, immutability, and recursion. Functional programming focused on composing functions and minimizing mutable state, leading to more concise and maintainable code. Modern functional programming languages like Haskell (1990) and Elixir (2011) continue to be influential.

Domain-specific languages (DSLs) were designed to address specific problem domains, enabling developers to express solutions in a language tailored to the problem at hand. DSLs like HTML (1993) for web development and MATLAB (1984) for mathematical computations have gained widespread adoption.

As computers became more powerful and multi-core architectures became prevalent, concurrent, and parallel programming gained importance. Concurrent programming focused on handling multiple tasks that execute independently but potentially interact with each other, while parallel programming aimed to leverage multiple processing units for increased performance. Languages like Erlang (1986) and Go (2009) were designed to provide built-in support for concurrency and parallelism. Martin Odersky created Scala as a programming language that combines aspects of functional programming.

In recent years, programming has witnessed a convergence of paradigms and the rise of new trends. Languages like Python (1991), developed by Guido Van Rossum as a simplified computer language that is easy to read, have embraced multiple programming paradigms, offering flexibility and expressiveness. Functional programming concepts have influenced mainstream languages like JavaScript, while OOP principles are employed.

Modern computer programming languages have evolved from earlier languages, with many older languages still in use or serving as the basis for new languages. The newer languages strive to simplify the programming process for developers. Programming skills helped developers to use their creativity and develop useful software products for enterprises. And these products interacted with data of enterprises, which are stored using Databases.

1.4. Story of Databases

Humans have been storing information (gathering data) since ages. However, as computers became more accessible and businesses started using them for practical purposes, the need for managing and storing large amounts of data became apparent. Initially, data was considered secondary to the primary computational tasks. It was often stored in files or even on punched cards, and the emphasis was on processing and performing calculations. But as businesses started relying on computers to handle their operations and store valuable information, the importance of managing data efficiently grew. This led to the emergence of databases as a structured and organized way to store, retrieve, and manage data.



A database is a self-describing collection of integrated records. A record is a representation of some physical or conceptual object. A database is self-describing in the context that it contains a description of its own structure. This description is called **metadata** - *data about the data*. The database is integrated in that it includes the relationships among data items, as well as including the data items themselves (Kroenke & Auer, 2007).

The integration aspect of a database refers to the inclusion of relationships among data items in addition to the data items themselves. This means ***that a database not only holds the individual data items but also captures the connections and associations between different data elements.***

By incorporating metadata and maintaining relationships among data items, databases provide a structured and organized approach to storing and managing information, making it easier to search, update, and retrieve information as needed. They allowed for the creation of relationships between different sets of data, enabling more complex and sophisticated data management.

Over time, databases evolved to include more advanced features, such as support for multiple users accessing the data simultaneously, data integrity constraints, and data security mechanisms. Relational databases, which organize data into tables and use structured query language (SQL) for managing data, became particularly popular and widely adopted.

With the advent of the internet and the explosive growth of digital information, databases became even more crucial. They underpin numerous applications and services we rely on today, from e-commerce platforms and social media sites to banking systems and healthcare records.

In recent years, we've also seen the rise of non-relational databases, such as NoSQL databases, which provide flexible data models and scalability options to handle the demands of modern applications.

1.4.1. History of DATABASES

During the 1960s and mid-1960s, the popularity and cost-effectiveness of computers led to a new direction in the field of databases. The term “Database” emerged during this time, coinciding with the availability of direct-access storage devices like disks and drums. These storage technologies allowed for shared interactive use, contrasting with the previous tape-based systems that relied on batch processing.

1.4.1.1. Primitive Digital Databases

During the 1960s, hierarchical databases emerged as one of the earliest database management systems (DBMS). IBM's Information Management System (IMS) was a notable example of a hierarchical database. It organized data in a tree-like structure, with parent-child relationships. Soon after, network databases, such as CODASYL (Conference on Data Systems Languages), were introduced, enabling more complex relationships between data elements.

The introduction of the relational model revolutionized the database landscape. In 1970, Edgar Codd proposed the concept of a relational database, which represented data as sets of tables with rows and columns. The Structured Query Language (SQL) was developed as a standardized language to interact with relational databases. Oracle, IBM's DB2, and Microsoft SQL Server were some of the early relational database management systems (RDBMS) that gained popularity.

Object-oriented databases (OODBMS) emerged in the 1980s, driven by the need to store and manipulate complex data structures used in object-oriented programming. OODBMS represented data as objects, allowing for the storage of both data and associated behaviors. However, despite their potential, OODBMS adoption remained limited due to challenges in standardization and compatibility.

To bridge the gap between relational databases and object-oriented programming, relational-object mapping frameworks and tools were developed. These frameworks, such as Java Database Connectivity (JDBC) and Object-Relational Mapping (ORM) tools like Hibernate, allowed developers to map objects to relational database tables and perform operations on them.

1.4.1.2. XML

In 1997, a significant development occurred with the introduction of the Extensible Markup Language (XML). XML is a markup language that establishes a set of rules for encoding documents in a format that is readable by both humans and machines. The XML 1.0 Specification, produced by the World Wide Web Consortium (W3C), along with other related specifications, defined XML as a gratis open standard. The design principles of XML prioritize simplicity, generality, and usability, particularly over the Internet. It is a text-based data format that provides extensive support, through Unicode, for languages worldwide.

While XML's design initially focused on documents, it has become widely utilized for representing diverse data structures, including web services. Numerous application programming interfaces (APIs) have been developed to enable software developers to process XML data effectively. Additionally, various XML schema systems exist to facilitate the definition of XML-based languages, enhancing the interoperability and structure of XML data.

1.4.1.3. Relational DATABASES

While databases have traditionally stored and retrieved data, correlations and decisions based on the data have often required human interaction or external programs. However, advancements in database technology have aimed to address this limitation.

In the past, databases primarily focused on enforcing data integrity by controlling the types of data that could be inserted into specific fields. For instance, if a field was designated for storing date and time values, any other type of data would trigger an error message. Over time, databases have become more advanced and introduced additional features such as triggers, cascading updates, and deletes. These features prevent inconsistencies between tables and help maintain data integrity during updates.

Databases have also incorporated simplified procedural languages that include embedded SQL, along with looping and control structures. Examples of these languages include Sybase/SQL Servers' Transact-SQL, Oracle's PL/SQL, and PostgreSQL. These languages provide a means for implementing more complex business logic within the database itself, reducing the reliance on external programs for data manipulation and decision-making.

1.4.1.4. NoSQL and Graph data

Carlo Strozzi's use of the term "NoSQL" in 1998 referred to his lightweight, open-source relational database that deviated from the standard SQL interface. However, it was Eric Evans, a Rackspace employee, who reintroduced the term "NoSQL" later on.

The term "NoSQL" aimed to label the emergence of a growing number of non-relational, distributed data stores that often did not prioritize providing ACID guarantees. ACID stands for Atomicity, Consistency, Isolation, and Durability, which are key attributes of traditional relational database systems like Sybase, IBM DB2, MySQL, Microsoft SQL Server, PostgreSQL, Oracle RDBMS, and Informix.

With the explosion of internet and social media, new data storage and processing needs emerged. NoSQL (Not Only SQL) databases emerged as alternatives to traditional relational databases, offering scalability, flexibility, and faster performance for handling large volumes of unstructured or semi-structured data.

Types of NoSQL databases include key-value stores, document stores, column stores, and graph

databases. Graph databases gained prominence for handling highly connected data, such as social networks and recommendation systems. Graph databases use graph structures to represent and query relationships between data elements efficiently. These databases gained prominence alongside major internet companies like Google, Amazon, Twitter, and Facebook, which faced unique challenges in handling vast amounts of data that traditional relational databases struggled to cope with.

1.4.1.5. Big DATA

As the volume of data continues to grow exponentially, databases are facing new challenges in handling and analyzing large datasets. Systems with terabytes of data, such as those used in scientific projects like the genome project, geological research, national security, and space exploration, require novel approaches for data management and analysis. Techniques like data mining, data warehousing, and data marts have emerged as common practices for extracting valuable insights from vast datasets. Additionally, the rise of big data necessitated the development of technologies like Apache Hadoop and Apache Spark, which provided distributed processing capabilities for handling massive datasets.

1.4.1.6. Database as Service

The advent of cloud computing brought about cloud databases, which offered scalable and on-demand data storage and processing capabilities. Database-as-a-Service (DBaaS) providers, such as Amazon RDS and Google Cloud Spanner, offered managed database solutions, eliminating the need for organizations to handle database administration and infrastructure management.

In recent times, databases are undergoing further advancements and incorporating more complex logic. One notable feature that is expanding is the concept of separating the physical location from the abstract concept of the database itself, as originally defined by Edgar F. Codd. This feature allows a database to be distributed across multiple locations and queried as a unified unit. Such instances are referred to as distributed or federated databases.

1.4.1.7. Distributed Database

With the introduction of distributed databases, different portions of a database can be physically located in separate geographical locations. For example, one part of a database can reside in New York while another part is in Mumbai. Queries that involve data from multiple locations can be executed concurrently at both locations, providing a seamless and unified view of the data.

The concept of distributed databases has become feasible due to advancements in network speed and reliability. The increased speed of networks enables efficient communication and data transfer between distributed locations, allowing for real-time or near-real-time access to data regardless of its physical location.

By leveraging distributed databases, organizations can achieve improved scalability, fault tolerance, and load balancing across multiple locations. This approach facilitates efficient data processing and analysis, particularly in scenarios where data is geographically dispersed or where high availability and redundancy are essential.

As data storage becomes increasingly complex, various types of specialized data pose unique challenges. Spatial data, for example, requires specific functions and structures to effectively store and query geographical information. This type of data is commonly used in geographic information systems (GIS) and requires spatial indexing techniques to optimize retrieval and analysis.

1.4.2. Continuous advancements in the field of Data Storage

Image data and scanned-in data present another set of challenges. Storing and managing large volumes of image files or scanned documents require efficient storage formats and indexing methods to support quick retrieval. Additionally, the processing of image data often involves techniques such as image recognition and computer vision, which require advanced algorithms and computational resources.

In the field of medicine, complex data types like gene sequences are crucial for research and analysis. Storing and processing genetic information demands specialized database structures and algorithms tailored to handle DNA sequences. Furthermore, medical devices that record physical data, such as patient monitoring devices, generate vast amounts of data that need to be efficiently processed and stored for further analysis.

Advancements in data storage and processing capabilities have been essential in meeting these challenges. The development of high-performance storage systems, including solid-state drives (SSDs) and distributed storage solutions, has improved data access speeds and overall performance. Additionally, advancements in computational power and cloud computing enable the processing of large-scale data sets, such as DNA sequences, to derive meaningful patterns and insights.

1.5. Machine Learning and AI

Ever since, Humans developed mechanism to store data, we have always tried to extract information from the data. The classification of similar objects into groups is an important human activity. In everyday life, this is part of the learning process: A child learns to distinguish between cats and dogs, between tables and chairs, between men and women, by means of continuously improving subconscious classification schemes.

Beyond classification, another important aspect of understanding data is identifying patterns within it. Humans have an innate tendency to recognize patterns and derive meaning from them. This ability allows us to identify trends, make predictions, and gain insights into complex systems. Recognizing patterns in data has significant implications across various domains, from scientific research to business analysis.

The concept of computers learning and improving automatically has been a long-standing fascination since their invention. Artificial Intelligence is the theory and development of computer systems able to perform tasks normally requiring human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages. **ML is a subfield of Artificial Intelligence (AI) that focuses on the development of algorithms and models that enable computers to learn and make intelligent decisions based on data.**

The process of pattern discovery in ML involves training models on labeled data, where the patterns and relationships between input features and output labels are explicitly provided. Through iterative learning processes, ML models analyze the data, identify relevant patterns, and adjust their internal parameters to improve performance. This learning process allows the models to generalize their knowledge and make accurate predictions on new, unseen data.

ML algorithms employ various techniques to identify patterns, such as statistical analysis, clustering, regression, and neural networks. These techniques enable the models to identify complex relationships and dependencies within the data. ML models can uncover patterns that may not be easily discernible to humans due to the high dimensionality or non-linear nature of the data.

Pattern recognition and understanding in ML have found numerous applications across various domains. For instance, in image and speech recognition, ML models can learn to identify objects, faces, or speech

patterns by extracting and analyzing relevant features. In natural language processing (NLP), ML models can discover patterns in text data to perform tasks such as sentiment analysis, language translation, or question-answering. By recognizing patterns in language, AI systems can comprehend and generate human-like responses.

In business and finance, ML algorithms can analyze large datasets to identify market trends, make stock predictions, or detect fraudulent transactions. In healthcare, ML models can analyze patient data to predict disease outcomes, identify risk factors, or assist in medical diagnosis. ML is also extensively used in recommendation systems, where patterns in user preferences and behavior are analyzed to provide personalized recommendations.

The field of machine learning continues to evolve, with advancements in algorithms, models, and computational power. As more data becomes available and computational capabilities increase, the potential applications of machine learning expand across diverse domains, ranging from image and speech recognition to natural language processing and autonomous driving. The continuous development and deployment of machine learning techniques have the potential to drive significant progress in the field of artificial intelligence.

The potential impact of programming computers to learn from experience is immense. Imagine the possibilities of computers learning from medical records to identify effective treatments for new diseases, houses learning to optimize energy costs based on occupants' usage patterns, or personal software assistants learning user interests to deliver highly relevant information. If we could successfully unlock the ability for computers to learn, it would revolutionize computer usage, enabling new levels of customization and competence.

1.5.1. History of ML

1.5.1.1. First Neural Network

In the year 1943, Logician Walter Pitts and neuroscientist Warren McCulloch published world's first mathematical model, McCulloch-Pitts neuron, of a neural network to create algorithms that mimic human thought process. The McCulloch-Pitts neuron was inspired by the biological neurons found in the human brain. It introduced the concept of binary threshold units that received inputs and produced outputs based on predefined activation thresholds. The neuron's output would only activate if the summed inputs crossed a specific threshold value. This simple computational model paved the way for the development of more complex neural networks.

Neural networks, inspired by the McCulloch-Pitts model, became an essential component of many machine learning algorithms. Neural networks consist of interconnected nodes or artificial neurons that can learn from data, recognize patterns, and make predictions.

1.5.1.2. World War and Alan Turing

Alan Turing played a significant role in World War II through his work at the Government Code and Cypher School (GC&CS) at Bletchley Park, the British code-breaking center. Turing and his colleagues were tasked with breaking the German Enigma machine's encrypted messages, which were considered to be unbreakable at the time. Turing made significant contributions to the development of machines and techniques for code breaking.

The **Turing Test**, proposed by Alan Turing in 1950, is a significant milestone in the field of artificial intelligence. It serves as a benchmark for evaluating the ability of a machine to exhibit intelligent behavior indistinguishable from that of a human. The test involves a human evaluator interacting with a machine and a human through a computer interface. The evaluator's task is to converse with both the machine and the

human, asking them questions and receiving responses. If the evaluator cannot consistently distinguish between the machine and the human based on their answers, the machine is considered to have passed the Turing Test.

While the Turing Test has been influential in stimulating research and discussions around artificial intelligence, it has also faced criticism. Some argue that the test is subjective and merely measures the machine's ability to imitate human behavior rather than demonstrating true intelligence.

1.5.1.3. A Game of Checkers

In 1952, Arthur Samuel developed a computer program that played checkers at a championship level, marking an important milestone in the application of artificial intelligence techniques to gaming. Samuel's program employed various techniques, including the use of pattern recognition and a self-learning algorithm. It was one of the first instances of a computer program that could improve its performance through experience. The program would play numerous games against itself, gradually refining its strategy based on the outcomes and learning from its mistakes.

One of the key techniques used in Samuel's program was **alpha-beta pruning**, which is an algorithmic approach for minimizing the number of game positions that need to be evaluated. This technique allows for more efficient and effective decision-making by exploring only the most promising branches of the game tree.

Additionally, Samuel developed the **minimax algorithm**, a fundamental concept in game theory and artificial intelligence. The minimax algorithm is used to determine the best possible move for a player in a game by considering the potential outcomes and minimizing the maximum possible loss.

1.5.1.4. Perceptron: A Major Breakthrough

In 1957, Frank Rosenblatt, a psychologist and computer scientist, introduced the perceptron, which was a groundbreaking development in the realm of artificial neural networks. The perceptron is a type of algorithm that simulates the functioning of a biological neuron. It was inspired by the structure and functionality of the human brain. The perceptron consists of input nodes, weights assigned to each input, an activation function, and an output node. It takes in input data, processes it through the activation function, and produces a binary output.

The key innovation of the perceptron was its ability to learn from training data through a process called supervised learning. It adjusts the weights assigned to each input based on the observed outcomes, with the aim of improving the accuracy of its predictions. This process allows the perceptron to iteratively update its parameters until it achieves a desired level of accuracy.

Rosenblatt's perceptron algorithm gained attention for its ability to classify patterns and make predictions, and it played a significant role in the development of artificial neural networks and machine learning. However, it should be noted that perceptron has limitations in handling complex problems that are not linearly separable. This limitation led to the perception that perceptron was not capable of addressing more complex real-world problems.

1.5.1.5. Nearest Neighbor

In 1967, the article "Nearest neighbor pattern classification" by Cover and Hart introduced the concept of the nearest neighbor algorithm for pattern classification. This algorithm is widely used in machine learning and serves as a foundation for various classification tasks.

The nearest neighbor algorithm is an instance-based learning method that classifies an input object by comparing it to its nearest neighbors in the training dataset. The basic idea is to assign the same class label

to the input object as its closest neighbors in the feature space. In other words, the algorithm determines the class membership of an object based on the classes of its neighboring instances. To classify a new object using the nearest neighbor algorithm, the distances between the new object and all the instances in the training set are computed. The object is then assigned the class label that is most common among its k nearest neighbors, where k is a user-defined parameter representing the number of neighbors to consider. The distance metric used can vary, with popular choices being Euclidean distance or Manhattan distance.

1.5.1.6. Backpropagation: The Core of Deep Learning

In 1974, Paul Werbos introduced the concept of backpropagation in his doctoral dissertation titled "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences." Backpropagation is a fundamental algorithm for training artificial neural networks. Backpropagation is a gradient-based optimization algorithm that enables neural networks to learn from labeled training data. It involves two main steps: forward propagation and backward propagation. During forward propagation, the input data is fed through the network, and the outputs are computed layer by layer. Then, during backward propagation, the error between the predicted outputs and the true labels is calculated and propagated backward through the network. The weights of the network are adjusted based on the computed error gradients, aiming to minimize the prediction errors.

The key idea behind backpropagation is to iteratively update the weights of the neural network using gradient descent. By calculating the gradients of the error with respect to the weights, the algorithm determines how much each weight contributes to the overall error and adjusts them accordingly. This iterative process continues until the network converges to a point where the error is minimized, and the network can make accurate predictions on unseen data.

1.5.1.7. Power of Computation in 21st century

With time, computing power became more affordable and rise of Machine Learning was exponential.

1.5.1.8. 1997: A Machine Defeats a Man in Chess

In 1997, IBM's Deep Blue defeated world chess champion Garry Kasparov in a six-game match. This event showcased the potential of computers to outperform humans in complex tasks requiring strategic thinking and decision-making.

1.5.1.9. 2002: Software Library Torch

The development of Torch, a scientific computing framework with wide ML capabilities, in 2002, marked a significant milestone. Torch provided a powerful and flexible environment for researchers and practitioners to develop and deploy ML models.

1.5.1.10. 2006: Geoffrey Hinton, the father of Deep Learning

Geoffrey Hinton, a prominent figure in the field of ML, made significant contributions to the advancement of deep learning. In 2006, Hinton's research on deep neural networks and the introduction of the "deep belief network" demonstrated the potential of deep learning models to learn hierarchical representations of data.

1.5.1.11. 2011: Google Brain

The creation of Google Brain, a deep learning research project, in 2011, was a pivotal moment for ML. Google Brain employed large-scale neural networks to train models on massive datasets using distributed computing infrastructure.

1.5.1.12. 2014: DeepFace

DeepFace, developed by researchers at Facebook in 2014, demonstrated remarkable progress in facial recognition technology. The deep learning-based model achieved near-human-level accuracy in identifying faces in photographs.

1.5.1.13. 2017: ImageNet Challenge

The ImageNet Challenge is an annual competition that focuses on object detection and image classification tasks using large-scale visual recognition datasets. The challenge involves training models on a massive dataset called ImageNet, which contains millions of labeled images belonging to thousands of different categories. The goal is to develop models that can accurately classify and identify objects within images. In 2017, the ImageNet Challenge saw a remarkable performance from participating teams, with 29 out of 38 teams achieving an impressive 95% accuracy with their computer vision models.

Traditionally, computer vision tasks such as image recognition were approached using handcrafted features and traditional machine learning algorithms. However, the introduction of deep learning and convolutional neural networks (CNNs) revolutionized the field. These deep learning models are capable of automatically learning hierarchical representations from raw image data, leading to improved accuracy and performance.

1.5.1.14. 2020 & beyond

At the time of writing of this book, Generative AI is taking giant leaps in the field of computer science. **Generative AI**, also known as **Generative Adversarial Networks (GANs)**, is a subset of artificial intelligence (AI) that focuses on generating new content, such as images, text, music, or even videos, that is not directly sourced from existing data. It involves training models to understand and mimic the patterns and characteristics of the input data and then generate new samples that resemble the original data.

In Generative AI, two key components work together: the generator and the discriminator. The generator's role is to create new content based on random noise or a given input. The discriminator, on the other hand, tries to distinguish between the generated content and real examples from the original dataset. Through an iterative process, both components continuously improve their performance. Many players like Azure OpenAI and Google Bard have come to the picture with their Large Language Models, providing API to connect and train data, even finetune custom models.

1.6. Summary

In this chapter, we discussed about three major applications of Computer science: **Programming, Data and Machine Learning**. We observed their interconnectivity. Apart from computational abilities, Programs are used to develop software applications which are used by the customer like us. Software Applications retain data of customers to give them a custom user experience. This analysis of behavioral aspect of a customer is extracted from data using Machine Learning, a subset of Artificial intelligence. With time, amount of data being captured and generated has grown significantly, but so has grown the ability of computers to extract information from these dataset, thanks to powerful computational powers and advancements in the field of Data Science and ML. Information extracted is further used by Software Companies to enhance experience of their customers on their platforms. However, the use case of Machine Learning is not just limited to enhancing customer experience, but goes way beyond towards solving complex equations, pattern recognition, speech, and text generation, etc.

Now that we have the context, we shall be discussing Programming, Databases and Machine Learning in separate sections and at the right juncture we shall be discussing their potential convergence in the upcoming chapters.

Objects, Data & AI: Build thought process to develop Enterprise Applications

SECTION 1

OBJECTS

Chapter 2 Chapter 2

Object Oriented Programming

“The assumption that the problem of love is the problem of an object, not the problem of a faculty. People think that to love is simple, but that to find the right object to love - or to be loved by- is difficult.” - Erich Fromm

2.1. What is a Program?

 A set of instructions or a sequence of commands given to a computer to accomplish a specific task is called **Program**. Tasks can be anything from a simple calculation to complex operations. It may involve interaction with end users or other software systems.

Programming languages help us to write instructions to the computer in a human readable language. Instructions once written, are compiled, and translated into machine code, which consists of low-level instructions, that are understood and performed directly by computer's hardware.

2.2. Pillars of Programming

Enterprise Software programs are centered on two elements: code and data, based upon which the organization and construction of a program can be approached: **process-oriented** (code) and **object-oriented programming** (data).

The **process-oriented** model focuses on the sequence of steps or instructions (code) that a program follows. In this model, the program is seen as a series of linear steps, with code acting on data. A program is structured as a collection of independent processes that communicate and coordinate with each other to achieve the desired functionality. Each process is responsible for performing a specific task, and they interact by passing messages or sharing resources.

Imagine we are developing a simple video game where the characters move, shoot, and collect coins. In a process-oriented approach, program will be divided into separate processes, each responsible for one aspect of the game:

Movement Process: This process handles the character's movement when the player presses the arrow keys.

Shooting Process: This process deals with shooting bullets when the player presses the spacebar.

Coin Collection Process: This process manages the character's interaction with coins and keeps track of the collected coins.

These processes run independently and communicate with each other as needed. For instance, when the character collects a coin, the Coin Collection Process informs the game's score display process to update the player's score. When the character shoots a bullet, it informs the Shooting Process to create a new bullet object on the screen.

This approach helps to organize and manage the game's different functionalities separately, making it easier to understand and modify each part of the game without affecting the others. If we want to improve the shooting mechanism, we can focus on the Shooting Process without altering the Movement or Coin Collection processes.

Procedural languages like C are commonly used in this approach. It is used in systems where concurrency and asynchronous operations are crucial, especially while designing Operating Systems or networking systems. However, as programs grow larger and more complex, problems may arise due to the increasing difficulty of managing and understanding the codebase.

The applicability of Process oriented programming is limited, especially to systems heavily dependent on concurrency or parallelism. They are not best suited for enterprise application development, as the goal is to best represent the enterprise domain in a virtual world and process-oriented approach does not provide expressiveness in data modeling. Procedural languages don't capture active aspects of the domain, even though they do support complex data types. Programs written in procedural languages like C, Fortran, etc. are just a series of technical manipulation of data. They fail to capture any higher level meaning specific to the domain.

Enterprise applications involve wide range of functionalities and use of independent processes can make the life of a developer hell, as managing inter process communication can turn out to be a very challenging task. Memory usage is another consideration, as process-oriented approach tends to consume more resources due to the need for separate memory spaces and process management. Enterprise applications often need to efficiently manage their resources to cater a large number of users or transactions.

To address the challenges of increasing complexity, **object-oriented programming (OOP)** was developed. OOP organizes a program around its data, represented by objects which are real world entities, and the interactions between those objects (real world entities). In this paradigm, an object is an instance of a class, which defines the properties (data) and behaviors or functions (code) that objects of that class possess. The program's structure is defined by the relationships and interactions between these objects (real world entities), which exist in memory.

Imagine we need to develop a comprehensive system to manage various types of vehicles, including cars, bikes, and trucks, as well as their individual components, such as engines and brakes. Each vehicle type has unique attributes and behaviors, and each component plays a specific role in the operation of the vehicle.

Class Definitions:

1. *Vehicle Class: Our base class for all vehicles, encompassing common attributes like brand, model, and color. It includes methods for starting and stopping the vehicle.*
2. *Car, Bike, and Truck Classes: These are subclasses of the Vehicle class, each customized to suit the unique characteristics of its respective vehicle type. They include methods specific to their behaviors.*
3. *Vehicle Part Classes: We can define separate classes for key vehicle components (attributes of vehicle sub classes), such as engines and brakes. These classes encapsulate the attributes and behaviors of individual vehicle parts.*

As Eric Evans mentions in his book, Domain Driven Design, “*Object-oriented programming is powerful because it is based on a modeling paradigm, and it provides implementations of the model constructs*”.



OOP provides several advantages over the process-oriented model. It promotes modularity, encapsulation, and reusability by bundling data and code together in self-contained objects. Objects have well-defined **interfaces**, specifying how other parts of the program can interact with them, while keeping their internal details hidden. This **abstraction** allows for better code organization, easier maintenance, and the ability to model complex real-world systems more accurately. It is a natural fit for expressing domain level concepts across the programs and is often the first choice of architects while implementing model driven design across the application development.

In Object oriented languages, the paradigm (a fundamental approach or way of thinking about and solving problems) is logic, and the model, a representation or abstraction of a system, concept, or domain, is created by defining a set of logical rules and facts that describe the relationships, constraints, and behavior of the entities within the system or domain being modeled.

While process-oriented approach can be scalable by adding more processes, in contrast OOP compensates scalability through the use of design patterns and architecture principles like microservices or event driven design. OOP's modular nature allows different components of an enterprise application to be developed, deployed, and scaled independently.

By organizing a program around its data and defining clear interfaces to access and manipulate that data, object-oriented programming provides a more structured and manageable approach to software development. It offers benefits such as code reusability, extensibility, and easier maintenance, making it particularly useful for larger and more complex projects. These tools allow developers to model real-world entities more naturally, making it easier to represent and manipulate complex data structures. *So, what are Objects in Object Oriented Programming?*

2.3. Objects

In an enterprise world of software programs, an **Object is a collection of data and associated behaviors**. When dealing with real world problems, we can encounter many objects while designing a system. Let us take an example:

Suppose we are developing a social media application. In this application, we have two main types of objects: User and Post.

User: The User object represents an individual user of the social media platform. It would have associated data such as the user's name, email, date of birth, and profile picture. Additionally, it would have behaviors such as logging in, updating profile information, and posting content.

Post: The Post object represents a post or message shared by a user on the social media platform. It would have associated data such as the content of the post, the user who created it, the timestamp, and the number of likes or comments. The behaviors associated with a post could include editing the content, deleting the post, or interacting with other users' posts through comments or likes.

In this example, we differentiate between the User and Post objects based on their distinct characteristics and functionalities. The User object focuses on representing and managing user-related data and behaviors, while the Post object deals with the content and interactions between users' posts. By modeling these objects separately, we can maintain a clear separation of concerns and establish different sets of properties and methods for each object type. This separation allows us to perform operations specific to users or posts without mixing their functionalities.

2.4. Classes

To define what kind of object we are dealing with, we use Classes. In a car showroom, we have different models of car, but they all have attributes such as wheels, steering, headlights, engine, etc. and behaviors such as driving, honking, etc. associated with one class: CAR, a general class of cars.

 A **Class** serves as a blueprint or template that defines the structure, attributes, and behaviors of objects. An object, on the other hand, is an instance of a class, representing a specific occurrence or realization of that class.

Each object created from a class has its own unique set of data values for the attributes defined in the class. These data values represent the state of the object. Additionally, objects possess the behaviors or methods defined in the class, which determine how they can interact and manipulate their data.

Objects can also be associated with each other, forming relationships and interactions within a program. For example, in a social networking application (Figure2.1), User objects can be associated with other User objects through friendship connections, and Post objects can be associated with User objects as the creators of the posts. By creating multiple instances of a class (i.e., multiple objects), each object can have its own specific data values and can behave independently. This allows for the modeling of complex systems where different objects interact and collaborate with each other.

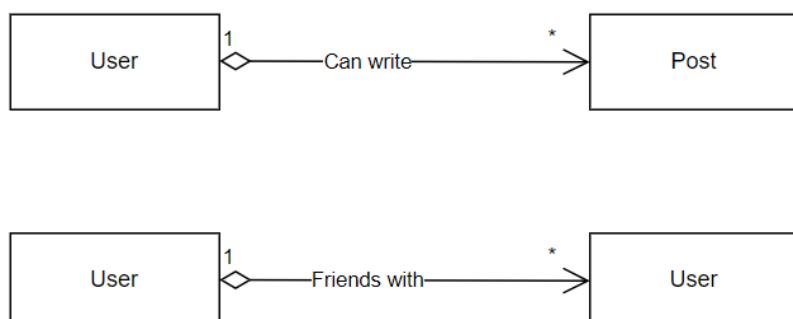


Figure 2-0-1

2.5. Programmatic representation of Class

Following is the program code in Python for User Class for the Social Media example discussed above:

```
class User:
    """
    Represents a user in the social networking application.

    Attributes:
        username (str): The username of the user.
        email (str): The email address of the user.
        password (str): The password of the user.
    """

    def __init__(self, username, email, password) -> None:
        """
        Represents a user in the social networking application.
        """

```

```
Attributes:  
username (str): The username of the user.  
email (str): The email address of the user.  
password (str): The password of the user.  
....  
self.username = username  
self.email = email  
self.password = password  
  
def add_friend(self) -> None:  
    ....  
    Adds another User object as a friend.  
    ....  
    # Write your own code  
    pass  
  
def create_post(self) -> None:  
    ....  
    Adds another User object as a friend.  
    ....  
    # Write your own code  
    pass
```

In the example code above, we have created a User class with **attributes** such as username, email, and password. We have also identified two behaviors of this class: create post and add friend. Behaviors are actions that can occur on an object. In programming sense, behaviors are also termed as **methods**. Methods accept parameters and attributes of the same class or actual object instances of a class can be passed to a method as an **argument**. Methods can also **return values**.

In the fascinating world of **Object-Oriented Programming (OOP)**, we create virtual realms, intricate systems of entities with their own unique attributes, capabilities, and interconnectedness. This incredible concept finds its inspiration in the tapestry of human behavior that we encounter in our daily lives.

Let us imagine a bustling school, a microcosm teeming with life and purpose. Within this educational system, we encounter a vibrant tapestry of characters (entities): the wise and nurturing teachers, the authoritative headmaster, the eager students, the diligent clerks, the helpful associates, and the knowledgeable librarians. Each of these individuals assumes a distinct role and carries specific responsibilities within the grand scheme of the school.

*Picture, for instance, a section of the library reserved exclusively for students above the 10th grade(authorization). School security mandates authentication through identity cards before granting access to the school premises. Delving deeper, we uncover a clever **abstraction** that shields junior teachers in the hierarchy from the intricate process of question paper creation for exams. This intricate task is reserved for a select few, carefully chosen for their expertise. By segregating teachers based on their academic qualifications, we distinguish junior teachers from their esteemed counterparts who hold a distinguished Ph.D., thus granting them the coveted title of senior teachers or Professors(**hierarchy**).*

*As we explore further, we discover the meticulous orchestration of the school's financial affairs. Here, the entities seeking financial assistance must engage with a diligent clerk who acts as the conduit between them and the enigmatic realm of the account's office (**encapsulation**).*

*Resourceful helper staff members lend their skills and versatility to fulfill multiple utility tasks across the daily functioning of the school (**polymorphism**).*

2.6. Important Concepts

2.6.1. Abstraction

Abstraction is used to manage complexities. A clerk while typing on a computer screen is not bothered by the internal workings of a computer. He will use a mouse, keyboard, and computer screen to perform his tasks on a Software. On the other hand, a hardware engineering would be dealing with different physical parts of computer such as CPU, RAM, Hard Drives, etc. and connecting them all together to make it work. Here, clerk and hardware engineer both work at different level of abstraction.



Abstraction means dealing with the level of detail that is most appropriate to a given task.

Hierarchical Classification is an effective way to manage abstraction. It helps us to break complex systems in more manageable pieces. We consider a computer as a single system, but internally it is divided into many sub systems. For dealing with physical parts, we have a hardware subsystem, which in turn deals with objects like CPU, Disk, CD, Monitor, Keyboard, Mouse, etc. These objects have their own workings and are made of many such sub systems.

This concept applies to Object Oriented Programming as well. Abstraction is the process of encapsulating information with a public interface. In programming, abstraction is typically achieved through the use of abstractions such as classes, interfaces, and functions. These abstractions define a set of behaviors or functionalities while hiding the internal workings or implementation details.

A class can be considered a layer of abstraction, which provides an interface to create objects and implement certain behaviors, yet information hiding can be done via **private members** of the class. The class provides a clear interface (methods and properties) for interacting with the object, while the internal implementation details are hidden from the outside world. This allows other parts of the program to use the class without needing to know how it is implemented internally, promoting modularity and reusability.



Abstraction allows programmers to work with high-level concepts and constructs that are closer to human understanding, rather than dealing with low-level implementation details. It provides a way to manage complexity, increase reusability, and improve the overall design and maintainability of software systems.

2.6.2. Encapsulation

Encapsulation, as the name suggests, refers to process of information hiding, in our case, it would be internal implementation of a class. However, it has broader aspect than just information hiding.



Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse, by controlling access to them to ensure data integrity and provide a well-defined interface to interact with the object.

Objects, Data & AI: Build thought process to develop Enterprise Applications

In encapsulation, the internal state of an object is hidden from the outside world, and only selected methods, known as accessors and mutators (getters and setters), are provided to manipulate the object's data. This helps in preventing direct access to the object's data and ensures that the object's state is accessed and modified in a controlled manner.

Let us look at a sample Java Program which tries to create a Bank account system in programming world.

```
import java.util.List;

/**
 * A Bank account class
 */
public class BankAccount {
    private int accountNumber;
    private String accountHolderName;
    private double balance;
    private List<Transaction> transactionHistory;

    public int getAccountNumber() {
        return accountNumber;
    }

    public void setAccountNumber(int accountNumber) {
        this.accountNumber = accountNumber;
    }

    public String getAccountHolderName() {
        return accountHolderName;
    }

    public void setAccountHolderName(String accountHolderName) {
        this.accountHolderName = accountHolderName;
    }

    public void setBalance(double balance) {
        this.balance = balance;
    }

    // Constructor and other methods...

    /**
     * Method for depositing amount to account
     * @param amount
     */
    public void deposit(double amount) {
        // Logic for depositing amount to the account
        // Update balance and transaction history
    }

    /**
     * Method to withdraw money from account
     * @param amount
     */
}
```

```
/*
public void withdraw(double amount) {
    // Logic for withdrawing amount from the account
    // Update balance and transaction history
}

/**
 * Method to fetch balance of the account.
 * @return
 */
public double getBalance() {
    return balance;
}
}
```

In this example, the *BankAccount* class encapsulates the account details and provides public methods (*deposit*, *withdraw*, *getBalance*) to interact with the account. The internal data fields are **private**, preventing direct access from outside the class. Users or other parts of the banking system can only access and modify the account's data through the defined **public** methods, ensuring data integrity and security. Different programming languages may have different syntax, but underlying concept remains the same.



Encapsulation enhances code maintainability by localizing changes. If the internal representation of an object changes, only the class itself needs to be modified while the external code remains unaffected. This reduces the ripple effect of changes throughout the codebase, making it easier to manage and maintain. Also, these objects can be easily used in different parts of the codebase or in other projects, as they provide a well-defined interface for interaction, thus promoting code reusability.

2.6.3. Inheritance

To inherit or acquire is Inheritance. In the object-oriented world, classes (**subclass or child class**) inherit or acquire certain properties or behaviors from other class (**superclass or parent class**). Hierarchical relationships between classes is formed where a subclass can inherit and extend or override the characteristics of a superclass.



Inheritance establishes an "IS-A" relationship between the superclass and subclass. This means that a subclass is a specialized type of the superclass.

Let us look at it from the Banking context via a simple Java Program:

```
/*
 * Represents a bank account.
 */
public class SavingsAccount extends BankAccount {
    private double interestRate;
    /**
     * Constructs a SavingsAccount object with the specified account number, account holder name,
     * initial balance, and interest rate.
     */
}
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
* @param accountNumber      the account number
* @param accountHolderName  the account holder's name
* @param balance            the initial balance
* @param interestRate       the interest rate for the savings account
*/
public SavingsAccount(int accountNumber, String accountHolderName, double balance, double interestRate) {
    super(accountNumber, accountHolderName, balance);
    this.interestRate = interestRate;
}

/**
 * Applies interest to the savings account by calculating the interest based on the current balance
 * and adding it to the account.
 */
public void applyInterest() {
    double interest = getBalance() * interestRate;
    deposit(interest);
}
}
```

Here, Savings Account "IS-A" Bank Account. We create a *SavingsAccount* object, deposit some funds using the inherited *deposit()* method from the *BankAccount* class, and then apply interest using the *applyInterest()* method specific to the *SavingsAccount* class. Thus, we have extended and customized the functionality of the *BankAccount* class to create a specialized type of account (*SavingsAccount*) that includes additional behavior specific to savings accounts.

Let us extend this example by considering behavior of a Checking Account where withdrawal can be done from the balance amount and if there is not sufficient fund then an overdraft limit can be availed in case of emergency. So, the withdrawal method of checking account is different from the basic withdrawal method followed by Bank accounts by default.

Here is the code example of Checking Account where withdrawal functionality can be overridden, and its own custom functionality can be implemented.

```
/**
 * Represents a checking bank account.
 */
public class CheckingAccount extends BankAccount {
    private double overdraftLimit;

    /**
     * Constructs a CheckingAccount object with the specified account number, account holder name,
     * initial balance, and overdraft limit.
     *
     * @param accountNumber      the account number
     * @param accountHolderName  the account holder's name
     * @param balance            the initial balance
     * @param overdraftLimit     the overdraft limit for the checking account
     */
}
```

```

public CheckingAccount(int accountNumber, String accountHolderName, double balance, double overdraftLimit) {
    super(accountNumber, accountHolderName, balance);
    this.overdraftLimit = overdraftLimit;
}

/**
 * Withdraws the specified amount from the checking account, allowing overdraft up to the overdraft limit.
 * If the withdrawal exceeds the balance and the available overdraft, an error message is displayed.
 *
 * @param amount the amount to withdraw
 */
public void withdraw(double amount) {
    double balance = getBalance();
    if (balance + overdraftLimit >= amount) {
        balance -= amount;
        setBalance(balance);
    } else {
        System.out.println("Insufficient funds");
    }
}
}

```

2.6.4. Polymorphism

A Lamborghini or a Mustang IS-A car. Car can be used a generic interface to describe properties (attributes) such as wheel, steering, headlight, etc. and behaviors (methods) such as driving, honking, etc. of both Lamborghini and Mustang.

Similarly, A Savings Account or a Checking account IS-A Bank Account. However, if a Bank Account is savings account, it will not allow you to withdraw money if you have zero balance but for a checking account, you can withdraw availing an overdraft limit.



Polymorphism is a feature in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass or interface. It enables a single interface to be used for a general class of actions, providing flexibility and code reuse.

Let us finalize the code for our Banking Example to conclude:

```

public class BankingExample {
    /**
     * @param args
     */
    public static void main(String[] args) {
        BankAccount account1 = new SavingsAccount(123456, "John Doe", 1000.0, 0.05);
        BankAccount account2 = new CheckingAccount(987654, "Jane Smith", 2000.0, 1000.0);

        account1.deposit(500.0);
        account1.withdraw(200.0);
    }
}

```

```
account2.deposit(1000.0);
account2.withdraw(3000.0);

System.out.println("Account 1 balance: " + account1.getBalance());
System.out.println("Account 2 balance: " + account2.getBalance());
}

}
```

In the *BankingExample* class, we create instances of *SavingsAccount* and *CheckingAccount* but store them in variables of type *BankAccount*. Polymorphism allows us to treat instances of *SavingsAccount* and *CheckingAccount* as objects of the *BankAccount* superclass. This flexibility enables us to use a single interface (*BankAccount*) to perform operations on different types of accounts.



Caution

If we instantiate a *SavingsAccount* object using the *BankAccount* class reference, we will only have access to the methods and members that are defined in the *BankAccount* class. Although we can't directly access the specific methods of subclasses when using the superclass reference, polymorphism allows us to override and provide different implementations of methods in the subclasses. We shall look at these design problems in coming chapters.

2.7. Summary: Encapsulation, Inheritance and Polymorphism

In Object-Oriented programming world, Encapsulation, inheritance, and polymorphism combine to produce a programming environment that supports the development of far more robust and scalable programs. Some of the key insights we have covered in this chapter are as follows:

- **Robust and Scalable Programs:** Polymorphism, encapsulation, and inheritance contribute to the development of robust and scalable programs. By using inheritance, we can create a hierarchy of classes that promotes code reuse and organization. This allows us to build upon existing code and extend functionality without starting from scratch. Additionally, encapsulation ensures that the internal implementation details of a class are hidden, reducing dependencies, and making it easier to modify and maintain code over time.
- **Code Reusability:** Inheritance enables code reuse by allowing subclasses to inherit properties and methods from their superclass. This saves time and effort by avoiding the need to rewrite common functionality. Through proper design, we can create a hierarchy of classes where subclasses inherit and extend the behavior of their parent classes. This code reuse enhances productivity and maintainability.
- **Migration and Compatibility:** Encapsulation plays a crucial role in maintaining code compatibility. By encapsulating the internal implementation details of a class, we can modify or improve that implementation without affecting the code that depends on the public interface of the class. This allows for gradual migration and evolution of the software, minimizing the risk of breaking existing code.
- **Clean and Readable Code:** Polymorphism allows us to write clean, sensible, and readable code. By designing our classes and interfaces with polymorphism in mind, we can create code that is more flexible and adaptable. Polymorphism enables us to write code that operates on a general

interface or superclass, making it easier to understand and maintain. It also promotes code that follows the principles of abstraction and separation of concerns.

- **Resilient Code:** Polymorphism aids in the creation of resilient code. With polymorphism, you can write code that can handle various types of objects without explicitly checking their specific types. This leads to more flexible and adaptable code that can handle different scenarios and changes in requirements.

In the next chapter, we shall discuss some of the design patterns used in Object oriented world to provide simple and elegant solutions to specific problems. By studying design patterns, we gain a deeper understanding of various problem domains and their corresponding design considerations.

Design Patterns in Object Oriented Programming

“ “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” - Christopher Alexander

3.1. Design Patterns: Introduction



Design Patterns are the blueprints to solve commonly occurring problems in software design.

However, design patterns are not to be confused with some reusable piece of code. These are generalized descriptions of Objects and Classes to be customized to solve a design problem for a specific domain. Each pattern focuses on a specific design pattern and provides a solution which in general lays down guidelines about how to structure Objects and Classes, make up the design, their relationships, responsibilities, and collaborations. Patterns are programming language agnostic and concept can be applied across different languages while developing enterprise applications.

3.2. Design Patterns: Mandate or Necessity?

Arguments can be drawn that usage of design patterns is not mandatory while programming and some might even showcase examples of how they have managed to develop and deploy an application without incorporating any patterns.

Argument well taken, however, the only constant to a successful software application is the change in its behavior as per the changing requirements over time to maintain an edge over its competitors. So, we are not just talking about building something and forget it. We are talking about a product which is built to last through turbulence of changing and testing times and requirements. And this is where Design Patterns come handy as necessary tools to maintain the software with efficient design, and if this toolkit is utilized modestly right through the initial design and development phase, then you can do away with some of the post release headaches of refactoring or restructuring the entire codebase.

It is true that most of us don't see the problems incoming at the time of coding, but by utilizing design patterns from the initial design and development phase, we can proactively address potential challenges and ensure the software is built with flexibility, adaptability, and maintainability in mind.

As requirements change and new features need to be added, design patterns allow for easier modification and extension of the existing code without introducing unnecessary complexity or risking the stability of the system. They enable developers to make incremental changes to the codebase, rather than resorting to massive refactoring or restructuring efforts. This saves time, reduces risks, and improves the efficiency of software maintenance.

Let us take a simple use case to understand this situation:

We are developing a payment processing system, for an e-commerce website, that handles various payment methods, such as credit cards, debit cards, and mobile wallets. The system needs to support different payment gateways, each with its own specific implementation and integration requirements. Payment system would be utilized while placing an order on the website.

How would a novice developer would attempt to design a solution for this problem?



Let us look at a simple code which can get the job done for us.

```
/*
 * Payment Processor class for processing different payment types
 */
public class PaymentProcessor {
    /**
     * This method processes a payment based on the payment method and amount provided.
     * It uses conditional statements to determine the logic for processing the payment based
     * on the payment method. If the payment method is not one of the three specified options
     * (credit card, debit card, or wallet),
     * additional code can be added to handle those other payment methods.
     * @param paymentMethod
     * @param amount
     */
    public void processPayment(String paymentMethod, double amount) {
        if (paymentMethod.equals("CreditCard")) {
            // Logic for processing credit card payment
        } else if (paymentMethod.equals("DebitCard")) {
            // Logic for processing debit card payment
        } else if (paymentMethod.equals("Wallet")) {
            // Logic for processing wallet payment
        }
        // Additional code for handling other payment methods
    }
}
```

Well, it was pretty simple right? If we have to add additional payment methods, then we will keep extending the if else logic. Do you see anything wrong with this approach?

Let us say, some of the payment methods have additional calculation logic based on some parameters, we will have to keep adding that block of code in specific if-else block of that payment method. With time, the code will get cumbersome and lengthy. May be two years down the line, a new developer joins the team and is handed over the responsibility of the code and told to modify a certain feature.

Imagine the horror on developer's face! Everything in a single place with no segregation at all. If one of the payment methods encounter an error, entire method will throw an exception. Use of conditional statements make the code less readable and harder to maintain. Code duplication is another problem which might introduce unnecessary error.

How to solve this design problem?

In the problem statement described above, we have payment strategy feature which keeps changing with time with respect to Payment Processor class. Main Job of Payment Processor class should be to identify which payment method is to be used, however, it is currently also taking the responsibility of implementation of various payment methods. So, it is doing a lot of heavy lifting. So, the first logical step would be to think of a solution, which would lead the way for segregation of concern in the code structure.

So, you already thinking towards one of the first principles of Object-Oriented Programming, i.e. **Single Responsibility Principle**. We shall be learning about it in the next few pages. Right now, let us focus back to our current example.

We assign the Payment Processor a single responsibility of identifying the payment method for processing payment and extract the part of payment method implementation.



Let us look at the following code to understand how it can be done effectively.

```
/*
 * This is an interface called IPaymentStrategy which declares a method called processPayment
 * that takes in a double value representing the payment amount.
 * It is meant to be implemented by classes that provide different payment strategies.
 */

public interface IPaymentStrategy {
    void processPayment(double amount);
}

// Concrete implementations of payment strategies

public class CreditCardPaymentStrategy implements IPaymentStrategy {
    @Override
    public void processPayment(double amount) {
        // Logic for processing credit card payment
    }
}

public class DebitCardPaymentStrategy implements IPaymentStrategy {
    @Override
    public void processPayment(double amount) {
        // Logic for processing debit card payment
    }
}

public class WalletPaymentStrategy implements IPaymentStrategy {
    @Override
    public void processPayment(double amount) {
        // Logic for processing wallet payment
    }
}
```



The modified *PaymentProcessor* class would look something like this:

```
/*
 * This is a PaymentProcessor class that has a private instance variable of type IPaymentStrategy.
 */
public class PaymentProcessor {
    private IPaymentStrategy paymentStrategy;

    /**
     * This method sets the payment strategy for the current object.
     * It takes an instance of an IPaymentStrategy as a parameter and
     * assigns it to the paymentStrategy field.
     * This allows the object to use the appropriate payment method when needed.
     * @param paymentStrategy
     */
    public void setPaymentStrategy(IPaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }
    /**
     * This method processes a payment using a selected payment strategy.
     * @param amount
     */
    public void processPayment(double amount) {
        // Delegate the payment processing to the selected strategy
        paymentStrategy.processPayment(amount);
    }
}
```



Now, let us see the use of this code to understand its effectivity.

```
public class Main {
    public static void main(String[] args) {
        // Create an instance of PaymentProcessor
        PaymentProcessor paymentProcessor = new PaymentProcessor();

        // Set the payment strategy based on the selected payment method
        paymentProcessor.setPaymentStrategy(new CreditCardPaymentStrategy());

        // Process the payment
        double amount = 100.0;
        paymentProcessor.processPayment(amount);
    }
}
```



We can observe that now we are able to set the payment strategy at run time. Also, the code has become more extensible and maintainable. If we have to add a new payment method, we do not need to touch the code in Payment Processor section. The separation of concerns is achieved by encapsulating the payment

processing logic within each strategy implementation. The *PaymentProcessor* class is modified to use the selected payment strategy and delegate the payment processing to that strategy.

Now, one can argue why did we use Interface of Payment Strategy when we could have leveraged Inheritance by creating a Super Class of Strategy and specific sub-classes extending its functionality.

3.2.1. Limitation of Inheritance

Answer to this question lies in the implementation of Inheritance itself. Following scenarios need to be considered in case of inheritance:

1. We must implement all abstract methods of the parent class even if there is no use of them in child class.
2. We must always make sure that the behavior implemented by the new child class is always in adherence with the functionality of base class or it would result in code breakage. Reason being objects of the subclass may be passed to any code that expects objects of the superclass.
3. There is tight coupling between parent class and child class.
4. When inheritance is used excessively to achieve code reuse, it can lead to the creation of parallel inheritance hierarchies or the explosion of class combinations. This situation occurs when there are multiple dimensions or orthogonal concerns in the system that need to be combined.

Let us elaborate more on the last point.

Suppose we have a base class called "Vehicle" that represents various types of vehicles. Now, let's say we want to introduce another dimension, such as "Color," to represent different colors of vehicles. Here, we have two dimensions: 1. Type of Vehicle 2. Color of Vehicle.

If we approach this problem via inheritance, we will end up create sub classes of multiple combinations.

In case, we want to add more dimensions like size, fuel type, etc. class hierarchy will bloat even further. This approach violates the **Open-Closed Principle (OCP)**, which states that software entities (classes, modules, functions) should be open for extension but closed for modification.

Following [picture](#) depicts the class structure:

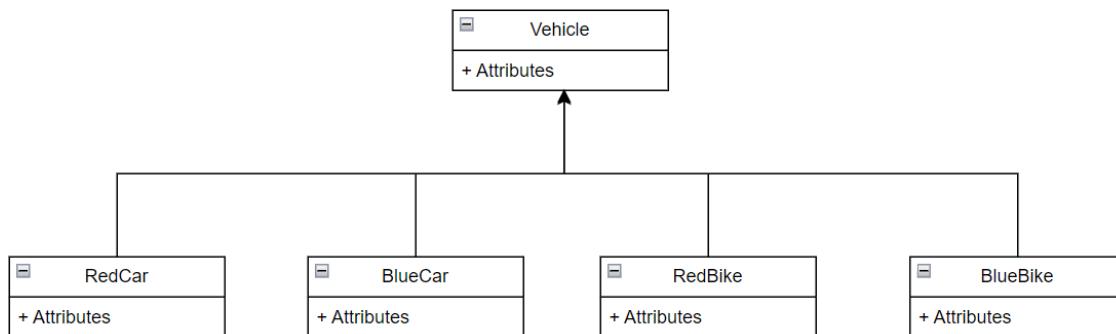


Figure 3-0-1

Hence, to overcome such scenarios, an alternative concept of **Composition** is introduced.

3.2.2. Composition



Composition in object-oriented programming refers to the practice of creating complex objects by combining or composing simpler objects.

It is a design principle that emphasizes building objects by assembling or "composing" them from other objects, rather than inheriting their behavior. In other words, while Inheritance represents IS-A relationship between classes, Composition represents HAS-A relationship between classes.

In our example above, we have used a HAS-A relationship between `PaymentProcessor` and `IPaymentStrategy` instance.

3.3. SOLID Principles

The SOLID Principles are five basic principles of Object-Oriented architecture, first introduced by Robert Martin. They are a set of rules and best practices to follow while designing a class structure.

SOLID is a mnemonic for following five design principles:

1. Single Responsibility Principle
2. Open-Closed Principle
3. Liskov Substitution Principle
4. Interface Segregation Principle
5. Dependency Inversion Principle

We already had an introduction with first two principles while discussing limitations of inheritance, nevertheless, we shall be categorically introduced in the following sections.

3.3.1. The Single Responsibility Principle



A class should have only one reason to change

In the context of this principle, responsibility is the reason for change and " If we can think of more than one motive for changing a class, then that class has more than one responsibility.

3.3.1.1. Example

Let us consider the following scenario in the following picture.

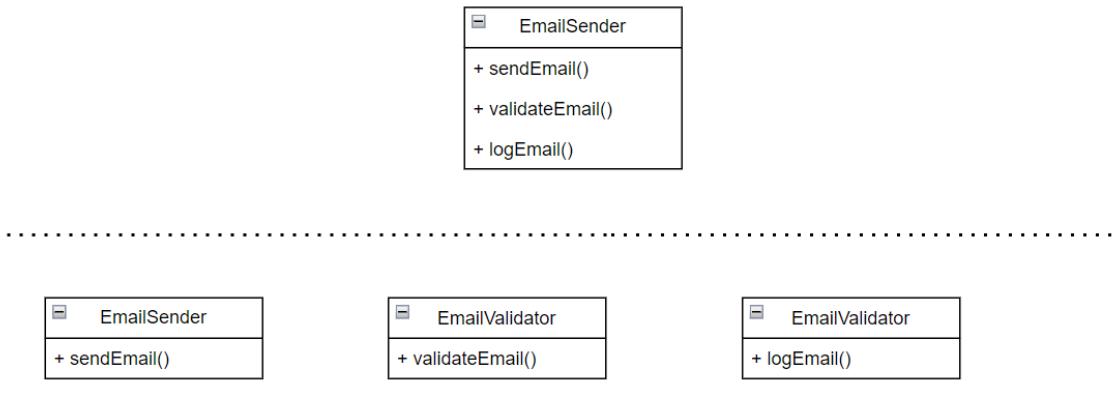


Figure 3-0-1

The `EmailSender` class is responsible for sending emails, validating email addresses, and logging sent emails. However, this violates the SRP because the class has multiple responsibilities. Hence, we have separated the responsibilities into three distinct classes:

1. `EmailSender` is now solely responsible for sending emails.
2. `EmailValidator` is responsible for validating email addresses.
3. `EmailLogger` is responsible for logging sent emails.

By adhering to the SRP, each class has a single responsibility, and if any changes or modifications are needed, they can be made independently without affecting other classes.

3.3.2. Open Close Principle

 *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*

In the vehicle [example](#) in paragraphs above, we discussed how adding new behaviour to classes leads to modification of existing behaviours. The main idea of this principle is to keep existing code from breaking when new behaviours are introduced in the system. A system where a single change triggers a cascade of changes in dependent modules, is called a rigid system. We want our software entities to be loosely coupled while designing the architecture.

When we say, “Open for extension”, it means we can extend a class, produce a sub-class, and do whatever you want with it—add new methods or fields, override base behaviour, etc.

And when we say, “Closed for Modification”, it means whenever we want to extend the behaviour of the class, we shall never be changing its source code.

 **Confused?? How can we change the behaviour of an entity without ever changing its source code?**

 **Abstraction** is the answer to this problem.

Abstract base classes provide a level of abstraction that defines the common interface, behaviour, and attributes shared by a group of related classes. They can serve as a blueprint or contract for derived classes to follow, while allowing each derived class to provide its own implementation details.

By defining abstract base classes, we can create a fixed set of methods and properties that all derived classes must implement. However, the actual behaviour and functionality can vary among the derived classes, effectively representing an unbounded group of possible behaviours.

Modifying existing code, especially in production environments, should be approached with caution and careful consideration. By creating a subclass and overriding specific parts of the original class, we can achieve the desired behaviour without directly modifying the original class. This approach provides a way to extend the functionality of a class without impacting existing clients or breaking their code.



Getting back to the vehicle [example](#), let us write the code in accordance with OCP principle and understand the outcomes.

```
public class Vehicle {  
    // Common vehicle attributes and behavior  
}  
  
public class Car implements Vehicle {  
    // Car-specific behavior and properties  
}  
  
public class Bike implements Vehicle {  
    // Bike-specific behavior and properties  
}  
  
public class Color {  
    // Color-related behavior and properties  
}  
  
public class Red implements Color {  
    // Implementation for red color  
}  
  
public class Blue implements Color {  
    // Implementation for blue color  
}  
  
public class ColoredVehicle {  
    private Vehicle vehicle;  
    private Color color;  
    public ColoredVehicle(Vehicle vehicle, Color color) {  
        this.vehicle = vehicle;  
        this.color = color;  
    }  
    public void setColor(Color color) {  
        this.color = color;  
    }  
    // Additional behavior and methods for the colored vehicle  
}  
  
Vehicle car = new Car();  
Vehicle bike = new Bike();
```

```
Color red = new Red();
Color blue = new Blue();

ColoredVehicle redCar = new ColoredVehicle(car, red);
redCar.setColor(blue); // Changing color of the vehicle dynamically
ColoredVehicle blueBike = new ColoredVehicle(bike, blue);
```

 With this approach, you can dynamically combine different vehicle types with different colours without the need for a massive hierarchy of subclasses. By separating the concerns of vehicle type and color into distinct classes and using composition, we achieve greater flexibility and maintainability. We can easily introduce new colors, vehicle types, or other dimensions without modifying existing code. The combination of dimensions is handled dynamically at runtime, allowing for a more scalable and adaptable solution.

This approach adheres to the principles of composition over inheritance.

3.3.3. Liskov Substitution Principle



Subtypes must be substitutable for their base types.

This principle states that the subclass should remain compatible with the behavior of the superclass. In other words, objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.

If a class B is a subtype of class A, then instances of class B should be able to be used wherever instances of class A are expected, without causing any unexpected behavior or violating the contract of the superclass.

In layman terms, a class's contract tells its clients what to expect. If a subclass extends or overrides the behavior of the superclass in unintended ways, it will break the contract. There can be various ways via which this contract can be violated.

Let us look at the checkpoints, if not adhered, might result in contract violation by sub-class.

3.3.3.1. Sub-Class Checkpoints

- The parameter types in a subclass method should either match exactly or be more general (abstract) than the parameter types in the corresponding superclass method.

Let us understand this via a use case.

Suppose we have a super class A with a method drive to drive the cars: drive(Car c). Now we create a sub class B and it overrides the existing drive functionality to drive all vehicles: drive(Vehicle v). If we pass the object of newly created subclass to the existing clients of the super class, things will work as expected. Reason: Earlier, it was expected to drive the cars however even with the new modification, it can drive all the vehicles which means it can still drive the car. Say, we create another sub class C and override drive method to drive only Lamborghini: drive(Lamborghini l). Now, we have a problem if we pass the object of sub class C to the clients of superclass, it will break the contract as it is expected to drive all the cars not just Lamborghini.

- Returning an object that's compatible with the object returned by the superclass method.

We can think of an inverse of the example above. If method `getName()`: `Car` of Super class `A` is returning an object of type `car` and the sub class `B` overrides this method which returns object of `Lamborghini`, it won't cause any contract violation. However, if sub class `C` overrides this method and returns an object of `Vehicle` then it will break the client contract, as it gets back a generic vehicle which it is not programmed to entertain.

- A method in a child class shouldn't throw types of exceptions which the parent class method isn't expected to throw.

This point is in line with the first two points. Modern day programming compilers also take care of this code smell.

- A child class should not change the semantics or introduce side effects that are not part of the superclass's contract.

This point reflects about not introducing a side effect. A side effect could be anything, even a print or log statement in sub class method which is not present in parent class.

- A sub class method should not change the pre-conditions or post conditions or else the contract expected by the client of super class will be violated.

For example, if a method of super class expects Numerical values in the argument but a sub class method applies a check to only accept positive values, it would lead to contract violation.

3.3.3.2. Importance of Liskov Substitution Principle

When client code cannot freely substitute a superclass reference with a subclass object, it often leads to the introduction of conditional code that checks the type of the object and performs special handling for specific subclasses. This conditional code tends to be scattered across the codebase, resulting in code that is difficult to maintain.

3.3.4. Interface Segregation Principle



Clients should not be forced to depend upon interfaces that they don't use.

Interface is a set of abstractions which the class implementing the interface must follow. The Interface Segregation Principles states that the client should not be exposed to the methods it is not using.

Say, we have a big fat interface with 5 methods abstracted, however, the client, which is implementing this interface, makes usage of only 3 methods. Thus, the program violated Interface Segregation principle. This principle advises that instead of one fat interface many small interfaces should be preferred based on groups of methods, each one serving one submodule.

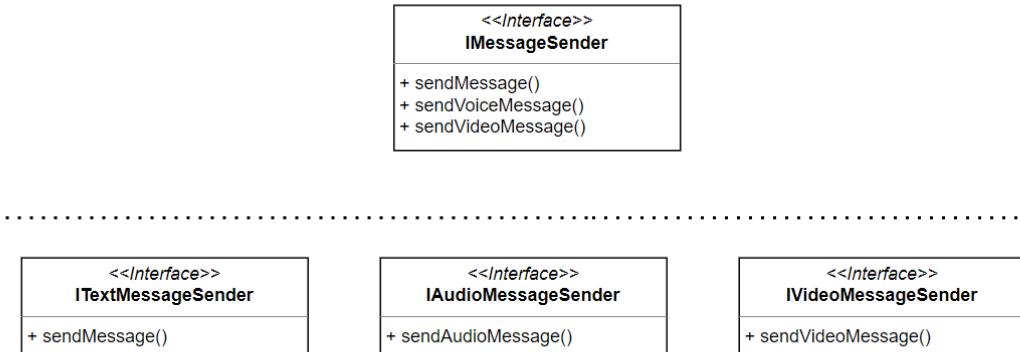


Figure 3-0-2

In the above [diagram](#), we can observe that the fat interface Message Sender has been broken down to three interfaces, each specific to their functionality. So, if a client needs only to implement Audio and Text message sending functionality, it does not longer need to worry about sending video message code, which was earlier abstracted.

3.3.5. The Dependency-Inversion Principle

 *High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.*

Let us first distinguish between High-Level Modules and Low-Level Modules.

High-level modules: They contain the important policy decisions and business models of an application. They contain complex business logics which directs low level modules to do tasks.

Low-Level modules: They deal with basic operations or tasks. We can also say that low level modules are created to manage unit behaviours separately, such as data transfer, database connection, etc.

The principle suggests that both high-level and low-level modules should depend on abstractions rather than concrete implementations. Abstractions provide a generalized interface or contract that can be implemented by various concrete classes. By depending on abstractions, modules become decoupled from specific implementations, making them more flexible and easier to modify or replace.

The principle also emphasizes that abstractions should not depend on implementation details. This means that the definition and behavior of abstractions should remain independent of specific implementation choices. By keeping abstractions free from implementation details, we ensure that they can be reused and the changes in implementation details do not affect the overall system design.

Instead it suggests that the implementation details should be depending on abstractions. This implies that low-level modules, which handle specific implementation details, should be designed in a way that they can be easily substituted or modified to conform to the abstractions defined by higher-level modules.

Let us consider an example of a messaging system that sends notifications through different channels such as email, SMS, and push notifications.

In a traditional approach, the high-level module responsible for sending notifications might directly depend on the low-level modules that handle the specific implementations of email, SMS, and push notification sending.

```
package DIP;

public class MessageSender {
    private EmailSender emailSender;
    private SMSSender smsSender;
    private PushNotificationSender pushNotificationSender;

    public MessageSender() {
        this.emailSender = new EmailSender();
        this.smsSender = new SMSSender();
        this.pushNotificationSender = new PushNotificationSender();
    }

    /**
     * Sends notification via various channels
     * @param recipient
     * @param message
     */
    public void sendNotification(String recipient, String message) {
        // Sending email notification
        if (isValidEmail(recipient)) {
            emailSender.sendNotification(recipient, message);
        }

        // Sending SMS notification
        if (isValidPhoneNumber(recipient)) {
            smsSender.sendNotification(recipient, message);
        }

        // Sending push notification
        pushNotificationSender.sendPushNotification(recipient, message);
    }
}

package DIP;

public class EmailSender {
    public void sendEmail(String recipient, String message) {
        // Implementation for sending an email notification
    }
}

package DIP;

public class MessageSender {
    private EmailSender emailSender;
    private SMSSender smsSender;
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
private PushNotificationSender pushNotificationSender;

public MessageSender() {
    this.emailSender = new EmailSender();
    this.smsSender = new SMSSender();
    this.pushNotificationSender = new PushNotificationSender();
}

/**
 * Sends notification via various channels
 * @param recipient
 * @param message
 */
public void sendNotification(String recipient, String message) {
    // Sending email notification
    emailSender.sendEmail(recipient, message);

    // Sending SMS notification
    smsSender.sendSMS(recipient, message);

    // Sending push notification
    pushNotificationSender.sendPushNotification(recipient, message);
}

}

package DIP;
public class PushNotificationSender {
    public void sendPushNotification(String recipient, String message) {
        // Implementation for sending a push notification
    }
}
```

 We can observe the tight coupling in the send method of high-level module: *MessageSender*. We are performing checks before sending to appropriate channels. If in future, new channels are introduced, we will have to modify existing code, which will violate Open Close Principle.

If any changes are required in the messaging system, such as adding a new notification channel or modifying the behavior of existing channels, the high-level module: *MessageSender* would need to be modified. This violates the principle of separation of concerns and makes the code less maintainable and extensible.



Now, let us modify our code in adherence with DIP principle.

```
package DIP;

public interface INotifier {
    void sendNotification(String recipient, String message);
}

package DIP;
```

```
public class MessageSender {
    private INotifier notifier;

    public MessageSender(INotifier notifier) {
        this.notifier = notifier;
    }

    /**
     * Sends notification via various channels
     * @param recipient
     * @param message
     */
    public void sendNotification(String recipient, String message) {
        // Send notification
        notifier.sendNotification(recipient, message);
    }
}

package DIP;

public class EmailSender implements INotifier {
    @Override
    public void sendNotification(String recipient, String message) {
        // Implementation for sending an email notification
    }
}

package DIP;

public class SMSSender implements INotifier{

    @Override
    public void sendNotification(String recipient, String message) {
        // Implementation for sending an SMS notification
    }
}

package DIP;

public class PushNotificationSender implements INotifier{

    @Override
    public void sendNotification(String recipient, String message) {
        // Implementation for sending a push notification
    }
}
```



Here, we have introduced an abstraction which represents a notifier. Message Sender will have a HAS-A relationship with this notifier. Also, the implementation of low-level modules for sending Emails, SMS or Push notification depend on the abstraction we created. This allows for flexibility and extensibility in the system. Say, if a new notification channel like a chatbot is added, a new class implementing the *Notifier* interface can be easily introduced without modifying the high-level module. There is no tight coupling between high-level modules and low-level modules.

3.4. Cataloguing Design Patterns

In the book “*Design patterns: Elements of reusable Object-Oriented Software*”, authors (historically named as *Gang of Four*) classified design patterns based on two criteria: **Purpose & Scope**.

Purpose reflects on what a pattern does. There are three purposes identified with respect to Objects.

1. **Creational:** deals with process of object creation.
2. **Structural:** deals with composition of classes or objects.
3. **Behavioral:** deals with interaction and responsibility distribution of classes or objects.

Scope defines whether patterns apply to classes or objects.



Design Patterns being followed in software development, are hugely inspired from how humans manage organizations effectively in the real world. These are not new inventions, but a blueprint prepared from learning the segregation behavior effectively employed by us in our daily scenarios for smooth functioning of an organisation or any system, while ensuring the integrity and security constraints.

As we learn about different design patterns in coming chapter, we can start creating a mental model and relate how we have been using or have experienced the usage of some of these patterns in our life already.



Abstraction, Encapsulation, Composition, or Inheritance are concepts, which are not bound to just software world but have been in practice since humans started designing effective organisations like court rooms, military bases, offices, schools, banks, etc. Likewise, the design patterns, rooted in above mentioned concepts, have emerged from decades of software development experience, and they provide proven solutions to common design problems. By applying these patterns, developers can create more maintainable, scalable, and secure software systems. The overlap with organizational behavior is a testament to how effective human problem-solving strategies can inspire robust software development practices.

Let us now discuss the broad categorisation of Design Patterns and then we shall start studying them in detail.

3.4.1. Creational Patterns

Class Patterns: In class-based creational patterns, the responsibility of creating objects is delegated to subclasses. The base class defines the interface or abstract methods for creating objects, while the subclasses provide the specific implementation. Examples of class-based creational patterns include the **Factory Method** pattern.

Object Patterns: In object-based creational patterns, the responsibility of object creation is delegated to another object. Instead of using inheritance, these patterns use composition, where one object holds a reference to another object responsible for creating the desired objects. Examples of object-based creational patterns include the **Abstract Factory**, **Prototype**, **Singleton** and **Builder** patterns.

3.4.2. Structural Patterns

Class Patterns: In class-based structural patterns, inheritance is used to compose classes. The patterns focus on class relationships and how they can be structured to form larger, more complex structures. Examples of class-based structural patterns include the **Adapter** pattern.

Object Patterns: In object-based structural patterns, objects are assembled or composed to create more complex structures. The focus is on object composition rather than inheritance. These patterns describe how objects can be connected and work together to achieve specific functionalities. Examples of object-based structural patterns include the **Bridge, Composite, Façade, Decorator and Proxy** patterns.

3.4.3. Behavioral Patterns

Class Patterns: In class-based behavioral patterns, inheritance is used to describe algorithms and flow of control. The patterns focus on defining common interfaces or base classes that describe the behavior and allow subclasses to override or implement specific algorithms. Examples of class-based behavioral patterns include the **Template Method** and **Interpreter** patterns.

Object Patterns: In object-based behavioral patterns, a group of objects cooperates to perform a task that no single object can accomplish alone. These patterns focus on the interactions and communication between objects to achieve a specific behavior. Examples of object-based behavioral patterns include the **Observer, Command, Mediator, Memento, Flyweight, State, Visitor, Chain of Responsibility, and Strategy** patterns.



It's important to note that these categorizations are not mutually exclusive, and some patterns may fall into multiple categories or have variations in their implementation. Some of the patterns are also used as an alternate to each other. Some of the patterns have lot of similarities in code implementation even though the motive might be different. The intent of each pattern should be considered based on its specific use case and the problem it addresses.

3.5. Summary

In this chapter, we introduced basic design principles of software development in object-oriented programming and catalogued design patterns based on their purpose and scope. In the next few chapters, we shall be looking at some of the key design patterns in details and observe how they can be used.

Objects, Data & AI: Build thought process to develop Enterprise Applications

Creational Design Patterns

 “I think computer viruses should count as life ... I think it says something about human nature that the only form of life we have created so far is purely destructive. We've created life in our own image.” — Stephen Hawking

 **Creational** patterns focus on abstracting the instantiation process, allowing systems to be independent of creation, composition, and representation details.

So far in our coding examples, we have been creating concrete objects using the **new** keyword.

```
// Create an instance of PaymentProcessor
PaymentProcessor paymentProcessor = new PaymentProcessor();
```

However, at the core of all design principles we have studied so far, we are encouraged to program to an interface and not an implementation. And when we instantiate a class using **new** keyword, that is definitely programming to implementation.

 Well, then one might ask, how are we going to instantiate a class or create objects?

 We will still be using **new** keyword for object creation; however, we will encapsulate the process so that we have better flexibility and more code reuse.

Creational patterns help in designing a system independent of Object Creation process. They intend to give more flexibility in object creation by letting us configure at compile time or run time, what gets created, who creates it, how it gets created, and when it gets created.

4.1. Factory Pattern

Factory as the name suggests is used to produce goods. Similarly, in object-oriented world, we create a virtual factory to create objects and ask it to provide us with specific objects whenever required.

 **Factory Pattern** is used to abstract the object creation logic from run time.

Let us look at a relatable scenario to understand this better.

Say, in a manufacturing company, there is a need to create different types of vehicles. The company produces cars, bikes, and trucks. The process of creating these vehicles involves specific manufacturing steps for each type.

 Here is how would traditionally approach to this requirement would look like.

```
package Factory;
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
/**  
 * Vehicle Class  
 */  
  
public abstract class Vehicle {  
    abstract void manufacture();  
    // Other methods can be defined with respect to Vehicles  
}  
  
// Concrete Car class extending the Vehicle  
public class Car extends Vehicle {  
    @Override  
    public void manufacture() {  
        System.out.println("Car is being manufactured.");  
    }  
}  
  
// Concrete Bike class extending the Vehicle  
public class Bike extends Vehicle {  
    @Override  
    public void manufacture() {  
        System.out.println("Bike is being manufactured.");  
    }  
}  
  
// Concrete Truck class extending the Vehicle  
public class Truck extends Vehicle {  
    @Override  
    public void manufacture() {  
        System.out.println("Truck is being manufactured.");  
    }  
}  
  
package Factory;  
/**  
 * Vehicle Customizer class is used to add features to vehicles and manufacture them  
 */  
public class VehicleCustomizer {  
    public void customizeVehicle(String type) {  
        // create specific vehicle instances  
        Vehicle vehicle;  
        if (type.equalsIgnoreCase("car")) {  
            vehicle = new Car();  
        } else if (type.equalsIgnoreCase("bike")) {  
            vehicle = new Bike();  
        } else if (type.equalsIgnoreCase("truck")) {  
            vehicle = new Truck();  
        }  
    }  
}
```

```
// implement customization logic
vehicle.manufacture();
}

}
```



Issues with above solution

1. Whenever we have to introduce a new vehicle type or remove an existing vehicle type, based on the business requirement, we will have to change the code in Customizer class. It breaks OCP principle.
2. Customizer class is first creating objects and then customizing vehicle. It is currently sharing more than one responsibility, hence breaking SRP principle.
3. In current example Customizer is just one client using Vehicle objects, but in application there would be many such clients. Imagine code modification during requirement changes.



So, the factory pattern can be leveraged here and object creation logic can be encapsulated to make code more usable and loosely coupled and adhering to design principles, which in turn will help us in maintenance of the code.



Let us understand Factory Pattern implementation via code example below.

```
package Factory;

/**
 * Vehicle Factory to produce objects of different vehicle types
 */
public class VehicleFactory {
    /**
     * This method returns objects of Different Vehicle subclasses
     * @param type : Vehicle type
     * @return objects of Different Vehicle subclasses
     */
    public Vehicle createVehicle(String type) {
        if (type.equalsIgnoreCase("car")) {
            return new Car();
        } else if (type.equalsIgnoreCase("bike")) {
            return new Bike();
        } else if (type.equalsIgnoreCase("truck")) {
            return new Truck();
        }
        return null;
    }
}

package Factory;
/**
 * Vehicle Customizer class is used to add features to vehicles and manufacture
```

```
/*
public class VehicleCustomizer {
    VehicleFactory vehicleFactory;
    /**
     * This method customizes vehicles based on specific types
     * @param type
     */
    public void customizeVehicle(String type) {
        // create specific vehicle instances
        Vehicle vehicle;
        vehicle = vehicleFactory.createVehicle(type);
        // implement customization logic
        vehicle.manufacture();
    }
}
```



Using **Factory pattern**, we have encapsulated object creation logic and during requirement changes like addition of new vehicle types or removal of older one, changes need to be done only in *VehicleFactory* and not through entire code base.

4.2. Abstract Factory Pattern

Now, it is time to improve the code further. Even though, we have encapsulated object creation logic in the program above, we still have a **tight coupling in the factory** when it comes to create new objects. Yes, you identified it correctly, we are still doing a manual type check before creating object in the **factory**.



Design principles say that we should always program to an interface, not an implementation. We must identify the aspects of your application that vary and separate them from what stays the same.

And we will follow the exact words. In our [example](#), we understand that different vehicle types can be introduced or removed at any given instant. Hence, we would declare interfaces for each distinct vehicle of the vehicle family.



Let us understand this implementation via below code example.

```
package Factory;
/**
 * Abstraction of Vehicle Factory
 */
public interface VehicleFactory {
    Vehicle createVehicle();
}

/**
 * Concrete CarFactory class implementing the VehicleFactory interface
 */
```

```
public class CarFactory implements VehicleFactory {  
    @Override  
    public Vehicle createVehicle() {  
        return new Car();  
    }  
}  
  
/**  
 * Concrete BikeFactory class implementing the VehicleFactory interface  
 */  
public class BikeFactory implements VehicleFactory {  
    @Override  
    public Vehicle createVehicle() {  
        return new Bike();  
    }  
}  
  
/**  
 * Concrete TruckFactory class implementing the VehicleFactory interface  
 */  
public class TruckFactory implements VehicleFactory {  
    @Override  
    public Vehicle createVehicle() {  
        return new Truck();  
    }  
}
```

 We use the **Abstract Factory Pattern** to create families of related objects without specifying their concrete classes, in this case, vehicles.

 The *Vehicle* interface remains the same as before, defining the common functionality for all vehicles. The concrete vehicle classes (*Car*, *Bike*, *Truck*) also remain unchanged. However, instead of a single *VehicleFactory* class, we now have an abstract *VehicleFactory* interface. This interface declares a *createVehicle()* method that returns a *Vehicle* object. The concrete factories (*CarFactory*, *BikeFactory*, *TruckFactory*) implement the *VehicleFactory* interface. Each factory is responsible for creating a specific type of vehicle. For example, the *CarFactory* creates *Car* objects, the *BikeFactory* creates *Bike* objects, and so on.

Now, let us look at how this code would be consumed by *VehicleCustomizer* client.

```
package Factory;  
/**  
 * Vehicle Customizer class is used to add features to vehicles and manufacture  
 */  
public class VehicleCustomizer {  
    VehicleFactory vehicleFactory;
```

```
/**  
 * This method customizes vehicles based on specific types  
 * @param type  
 */  
public void customizeVehicle(String type) {  
    // create specific vehicle instances  
    VehicleFactory carFactory = new CarFactory();  
    Vehicle car = carFactory.createVehicle();  
    car.manufacture();  
  
    VehicleFactory bikeFactory = new BikeFactory();  
    Vehicle bike = bikeFactory.createVehicle();  
    bike.manufacture();  
  
    VehicleFactory truckFactory = new TruckFactory();  
    Vehicle truck = truckFactory.createVehicle();  
    truck.manufacture();  
}  
}
```

 Here in *VehicleCustomizer* client code, we create instances of the concrete factories (*carFactory*, *bikeFactory*, *truckFactory*). Then, we use the factory objects to create the corresponding vehicles by invoking the *createVehicle()* method. The factory internally handles the object creation process and returns the appropriate concrete vehicle object.

 By utilizing the **Abstract Factory Pattern**, the client code is decoupled from the concrete vehicle classes and specific factory implementations. Instead, the client code interacts with the abstract factory interface, allowing for flexibility and easy substitution of different factory implementations. This pattern enables the creation of families of related objects and provides a higher level of abstraction in the code structure.

4.3. Builder Pattern

So far in our [example of vehicle application](#), we have been creating objects of different types of vehicles, however, vehicles had zero or limited attributes assigned. Code in constructor has been clean. Now, let us increase the complexity to some degree so that our simple program resembles more to classes or objects in real world.

Currently, we have three vehicle types into our system: Car, Bike and Truck. Now, let us just focus on one vehicle type, say Car. Business heads want to add more details of a car to be captured by the application, such as engine, wheel, brand, etc. These attributes may have their own attributes being captured by the system. Engine may have attributes like engine type and horsepower. Now, we have case of nested attributes and object creation logic gets complex inside constructor.

```
/**  
 * Constructs a new Car object with the provided parameters.  
 *  
 * @param wheel The wheel of the car.  
 */
```

```
* @param brand The brand of the car.  
* @param color The color of the car.  
*/  
private Car(String wheel, String brand, String color) {  
    this.wheel = wheel;  
    this.brand = brand;  
    this.color = color;  
    Engine engine = new Engine("V8", 500);  
    this.engine = engine;  
}
```

 Now, *Engine* object creation is also happening inside constructor of *Car*. One might argue that this one statement of engine creation can be pulled out of constructor code and constructor be supplied with *Engine* object via arguments. Agreed, this can be a possible approach. However, imagine a situation, when you are creating a *Car* object and engine information is not available. Now, class *Car* is tightly coupled with class *Engine*. Repercussions you are smart enough to figure out.



Builder Pattern comes to our rescue!



Builder Pattern lets us construct complex objects step by step. It allows us to produce different types and representations of an object using the same construction code.



Let us look at the modified code leveraging this pattern and decoupling the dependencies.

```
package VehicleExample;  
  
/**  
 * Concrete Car Class  
 */  
public class Car extends Vehicle {  
    private String wheel;  
    private String brand;  
    private String color;  
    private Engine engine;  
  
    /**  
     * Retrieves the wheel of the vehicle.  
     *  
     * @return The wheel of the vehicle.  
     */  
    public String getWheel() {  
        return wheel;  
    }  
  
    /**  
     * Retrieves the brand of the vehicle.  
     */
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
* @return The brand of the vehicle.
*/
public String getBrand() {
    return brand;
}

/**
 * Retrieves the color of the vehicle.
 *
 * @return The color of the vehicle.
 */
public String getColor() {
    return color;
}

/**
 * Retrieves the engine of the vehicle.
 *
 * @return The engine of the vehicle.
 */
public Engine getEngine() {
    return engine;
}

/**
 * Constructs a new Car object with the provided builder.
 *
 * @param builder The CarBuilder object used to construct the Car.
 */
private Car(CarBuilder builder) {
    this.wheel = builder.wheel;
    this.brand = builder.brand;
    this.color = builder.color;
    this.engine = builder.engine;
}

@Override
public void manufacture() {
    System.out.println("Car with " + this.getWheel() + " wheel is being manufactured.");
}

// CarBuilder class for constructing Car objects
public static class CarBuilder {
    private String wheel;
    private String brand;
    private String color;
    private Engine engine;
```

```
/**  
 * Sets the wheel of the car.  
 *  
 * @param wheel The wheel to set.  
 * @return The CarBuilder instance.  
 */  
public CarBuilder setWheel(String wheel) {  
    this.wheel = wheel;  
    return this;  
}  
  
/**  
 * Sets the brand of the car.  
 *  
 * @param brand The brand to set.  
 * @return The CarBuilder instance.  
 */  
public CarBuilder setBrand(String brand) {  
    this.brand = brand;  
    return this;  
}  
  
/**  
 * Sets the color of the car.  
 *  
 * @param color The color to set.  
 * @return The CarBuilder instance.  
 */  
public CarBuilder setColor(String color) {  
    this.color = color;  
    return this;  
}  
  
/**  
 * Sets the engine of the car.  
 *  
 * @param engine The engine to set.  
 * @return The CarBuilder instance.  
 */  
public CarBuilder setEngine(Engine engine) {  
    this.engine = engine;  
    return this;  
}  
  
/**  
 * Builds and returns a new Car instance with the configured properties.  
 *  
 * @return A new Car instance.  
 */
```

```
 */
public Car build() {
    return new Car(this);
}
}
```

 In this modified version of the code, The *CarBuilder* class is a nested static class within the *Car* class. By making the *CarBuilder* class static and nested within the *Car* class, it is encapsulated within the scope of the *Car* class. This encapsulation allows the builder class to access the private constructor of the *Car* class, which enforces object creation through the builder. Since the *CarBuilder* class is static, it can be accessed without requiring an instance of the *Car* class. This means that the client code can create a builder object and invoke its methods directly without needing to instantiate the *Car* class first.

Irony: To remove the tight coupling during object creation with nested properties, we have introduced a tight coupling of our own. A necessary trade off in this case.

```
package VehicleExample;

public class Client {
    public static void main(String[] args) {
        Engine engine = new Engine("V8", 500);
        Car car = new Car.CarBuilder()
            .setWheel("MRF")
            .setBrand("Toyota")
            .setColor("Red")
            .setEngine(engine)
            .build();

        car.manufacture(); // output: Car with MRF wheel is being manufactured.
        System.out.println("Engine of Car: " + car.getEngine()); // output: Engine of Car: 2LTurbo
    }
}
```

 In the client code, we create an *Engine* object with the desired engine details. Then, using the *Car.Builder*, we set the car properties including the *Engine* object. Finally, we build the *Car* object and utilize it by invoking the manufacture method and retrieving the car and engine details.

 The **Builder Pattern** provides a way to construct complex objects step-by-step, allowing for flexible and readable object creation. It separates the construction logic from the object's representation, enabling the construction of objects with different configurations using the same building process. It simplifies the object creation process, supports optional and default property values, and ensures the resulting objects are in a consistent state.

4.4. Prototype Pattern

Different requirements in application arise need for different patterns in the solution. One pattern might be suitable for a specific scenario but won't fit at all for some other. With time and experience, we gain enough expertise to get an instinct about when to use a particular pattern.

In the above three creational patterns, we learnt how we can encapsulate object creation logic to suit our business needs. Carrying forward the discussion on similar lines, let us entertain another business scenario.

In the automotive industry, there are often standard configurations or models of vehicles that have slight variations based on customer preferences or market demands. Customers often have specific preferences for their vehicles, such as color, interior features, or additional accessories. In the vehicle development process, it is common to create prototypes for testing and validation purposes.

 Now, to replicate this scenario in our system, which enables to vary type and configuration of vehicles easily, we use **Prototype Pattern**.

 *Prototype pattern that lets us specify the kind of objects we want to create using a prototypical instance and cloning existing objects (prototypes) without making our code dependent on their classes.*

 Applying this pattern in our code, we add a `clone()` method in our `Car` class, when called via instance of the class, returns a new object with configuration similar to the instance of the object which has called the function.

```
public Vehicle clone() {  
    return new Car(this.type, this.brand, this.color);  
}
```

 Client code uses the returned original object as a prototype, clones and creates a new object and adds necessary configuration as per requirement.

```
Car carPrototype = new Car("Car", "Toyota", "Red");  
carPrototype.manufacture();  
  
Car clonedCar = (Car) carPrototype.clone();  
clonedCar.manufacture();  
  
// Modify the properties of the cloned car  
clonedCar.setBrand("Honda");  
clonedCar.setColor("Blue");  
clonedCar.manufacture();
```



The Prototype pattern delegates the cloning process to the actual objects that are being cloned. The `clone` method creates an object of the current class and carries over all of the field values of the old object into the new one. This pattern can be used when a system needs to be independent of how objects are created, composed or represented and class instantiation is specified at run time. It helps to avoid building a class hierarchy of factories that is parallel to hierarchy of products. Cloning avoids the costly process of initializing objects from scratch, as it creates a replica of an existing object with pre-initialized properties. The Prototype pattern can greatly reduce the number of classes a system needs.



Cloning can be a difficult process to implement if a class internally includes objects that do not support copying or have circular reference.

4.5. Singleton Pattern

Continuing with our vehicle context, manufacturers are outsourced by vehicle makers for different parts. There can be a scenario that a particular manufacturer is utilized to supply car with certain basic configuration, say a specific engine type. Other properties of the model may vary and can be supplied, which the manufacturer would utilize in addition with its own specification of engine, to produce a new car model. Since, this is a unique manufacturer, system requires to have only one global instance of it throughout the application.

 This is where **Singleton Pattern** might come handy. We can limit resource allocation to just one instance of manufacturer throughout the application, and it will help ensure synchronization and adherence to predefined protocols. The singleton instance can act as a central control point for the manufacturing system. It allows for centralized configuration and management of various aspects, such as production rules, quality standards, or system-wide settings. This simplifies the maintenance and updates of these configurations.

 **Singleton Pattern** makes sure that a class has only one instance throughout application and a global access point is provided to access functionality of the class.

 Let us look the code snippet below.

```
package VehicleExample;

public class CarManufacturer {
    private static CarManufacturer instance;

    private CarManufacturer() {
        // Perform initialization here
    }

    public static CarManufacturer getInstance() {
        if (instance == null) {
            instance = new CarManufacturer();
        }
        return instance;
    }

    public Car manufactureCar(String wheel, String brand, String color) {
        // Manufacturing logic here
        Engine engine = new Engine("V8", 500);
        Car car = new Car.CarBuilder()
            .setWheel(wheel)
            .setBrand(brand)
            .setColor(color)
            .setEngine(engine)
            .build();
        return car;
    }
}
```

```
    }  
}
```

 Here in the above code example, I have incorporated the code of Builder pattern for *Car* object creation inside *CarManufacturer*, which has a *private static instance* variable instance to hold the single instance of the class. The constructor is made private to prevent direct instantiation from outside the class. The *getInstance()* method provides access to the singleton instance. If the instance variable is *null*, it creates a new instance of *CarManufacturer*. If the instance variable is already set, it simply returns the existing instance.

The *CarManufacturer* class also includes a *manufactureCar()* method, which represents the manufacturing process for cars. It takes parameters such as the wheel, color, and brand, and adds the specific engine type, for which this manufacturer specializes and returns the new created *Car* object.



Singleton Pattern helps in limiting the instance of a class to one, which ensures efficient allocation and utilization of available resources. Having multiple instances could lead to resource wastage or conflicts in resource allocation. Since the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it. Use of this pattern is strictly in adherence to business requirements. Here, we are limiting the accessibility of a class to one global instance but in other cases, where multiple instances of a class is desired, we must avoid Singleton Pattern.

4.6. Summary

In this chapter, we discussed different creational patterns at length and gained an insight into when it is suitable to use them and when we must avoid their use.

- **Factory Pattern:** The Factory Pattern provides an interface for creating objects, but the specific class to instantiate is determined by the factory class. It encapsulates object creation and decouples the client code from the specific object classes. It allows for flexible object creation based on different criteria.
- **Abstract Factory Pattern:** The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. It encapsulates a group of factory methods for creating different types of objects. It promotes the creation of objects that are compatible and work together.
- **Builder Pattern:** The Builder Pattern separates the construction of a complex object from its representation. It allows step-by-step creation of an object by providing a builder class that handles the construction process. It provides flexibility in constructing objects with optional or default property values.
- **Prototype Pattern:** The Prototype Pattern involves creating new objects by cloning existing objects, known as prototypes. It allows objects to be copied or cloned to create new instances with the same properties. This pattern is useful when creating new objects is costly or complex, and when objects need to be created dynamically at runtime.
- **Singleton Pattern:** The Singleton Pattern ensures that there is only one instance of a class throughout the application. It provides a global access point to the same instance, allowing centralized control and coordination. It is useful when we need to manage a shared resource or when having multiple instances would be redundant or inefficient.

Structural Design Patterns

“ “ *I am wary of the whole dreary deadening structured mess that we have built into such a glittering top-heavy structure that there is nothing left to see but the glitter, and the brute routines of maintaining it.*” — John D. MacDonald

 **Structural design patterns** in software engineering focus on the composition and organization of classes and objects to create larger, more complex structures, utilizing the concept of inheritance to combine interfaces or implementations, allowing for greater flexibility and efficiency in the overall design.

Creational Design Patterns dealt with object creation mechanisms and we discussed various such patterns in this section, which help us to create objects and use them efficiently based on business requirements. But, till now we have only used simpler class structures and as we move towards complex class structures, we would be leveraging Structural Design Patterns. These patterns provide guidelines on how to create objects in a flexible and reusable manner, promoting decoupling and enhancing the scalability of our systems. Now, as we transition towards structural design patterns, our focus shifts from the creation of individual objects to the composition and organization of these objects to form larger structures. These patterns help us understand how to assemble classes and objects to build robust architectures that are adaptable, efficient, and maintainable. These patterns utilize **inheritance** and **composition** to create **interfaces** and **implementations** that enable the construction of flexible software systems.

Different business use cases will call for usage of different structural patterns implementation. Let us start with Adapter Patterns.

5.1. Adapter Pattern

 **Adapter Patterns** are used to allow classes work together, which would not have been possible due to incompatible interfaces. It converts interface of a class into another interface as expected by client.

In Software industry, applications frequently encounter the need to seamlessly integrate with diverse third-party applications, enriching their functionality and enhancing user experiences. Moreover, as technology evolves and businesses strive to stay ahead, the process of modernization becomes inevitable.

Third-party applications may have their own unique interfaces, protocols, or data formats that differ from the expected interface of the application.

A crucial aspect of modernization involves addressing legacy behaviours and attributes. These aspects, which may have served the application well in the past, require thoughtful consideration and refinement to meet the evolving requirements of the present. The process involves implementing updated functionalities, optimizing performance, and adhering to the latest architectural patterns.



One of the key principles in software development is the idea of designing for **reusability**. However, there are situations where a toolkit class that is intended to be reusable may not fulfil its purpose because its interface doesn't align with the specific requirements of a particular application or domain.

Carrying on the vehicle example context, let us assume that a business requirement arises which demands conversion of Legacy Vehicle Data format into Modern Data Format. There might be changes in attribute names and how some of the attribute values are perceived in modern system. This is where Adapter Pattern will come handy.

Here, we have a legacy system data format which contains make, model, manufacturing year and engine used in a vehicle. However, in new data format, these attribute names are changed, and engine names have changed slightly. No doubt, legacy interface will not compatible with modern interface at compilation.



Hence, to solve this, we implement an adapter between the two data formats, which encapsulates the data conversion logic from the client.



Let us look at code snippets to understand better.

```
package Adapter;

public class LegacyVehicleData {
    private String make;
    private String model;
    private int manufacturingYear;
    private String engineModel;

    // getter and setter method implementation
}

package Adapter;

public class ModernVehicleData {
    private String brand;
    private String type;
    private int year;
    private String engineType;

    // getter and setter method implementation
}

package Adapter;

public interface EngineAdapter {
    String convertToModernEngineFormat();
}

package Adapter;

public interface LegacyVehicleDataAdapter {
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
ModernVehicleData convertToModernFormat();  
}  
package Adapter;  
  
public class EngineAdapterImpl implements EngineAdapter {  
    private LegacyVehicleData legacyVehicleData;  
  
    public EngineAdapterImpl(LegacyVehicleData legacyVehicleData) {  
        this.legacyVehicleData = legacyVehicleData;  
    }  
  
    @Override  
    public String convertToModernEngineFormat() {  
        // Perform engine data conversion/transformation from legacy to modern format  
        String engineModel = legacyVehicleData.getEngineModel();  
        // Example conversion logic  
        if (engineModel.equals("V4")) {  
            return "Inline-4";  
        } else if (engineModel.equals("V6")) {  
            return "V6";  
        } else {  
            return "Unknown";  
        }  
    }  
}  
  
}  
  
package Adapter;  
  
/**  
 * LegacyVehicleDataAdapterImpl  
 */  
public class LegacyVehicleDataAdapterImpl implements LegacyVehicleDataAdapter {  
    private LegacyVehicleData legacyVehicleData;  
    private EngineAdapter engineAdapter;  
  
    public LegacyVehicleDataAdapterImpl(LegacyVehicleData legacyVehicleData) {  
        this.legacyVehicleData = legacyVehicleData;  
        this.engineAdapter = new EngineAdapterImpl(legacyVehicleData);  
    }  
  
    @Override  
    public ModernVehicleData convertToModernFormat() {  
        String brand = legacyVehicleData.getMake();  
        String type = legacyVehicleData.getModel();  
        int year = legacyVehicleData.getManufacturingYear();  
        String engineType = engineAdapter.convertToModernEngineFormat();  
    }  
}
```

```
        return new ModernVehicleData(brand, type, year, engineType);
    }

}

package Adapter;

// Client code
public class Client {
    public static void main(String[] args) {
        // Assume we have a legacy vehicle data instance
        LegacyVehicleData legacyData = new LegacyVehicleData("LegacyMake", "LegacyModel", 2000, "V4");

        // Create an adapter instance and convert legacy data to modern format
        LegacyVehicleDataAdapter adapter = new LegacyVehicleDataAdapterImpl(legacyData);
        ModernVehicleData modernData = adapter.convertToModernFormat();

        System.out.println("Engine details: ---" + modernData.getEngineType());
    }
}
```

 In the code above, we are using two data formats *LegacyVehicleData* and *ModernVehicleData*. Both classes have similar attributes with different names though. Client which has been using Legacy format yet, now leverages *LegacyVehicleDataAdapter* to map the data from older format to latest format. *LegacyVehicleDataAdapter* class encapsulates the logic from client code about conversion of data format and makes the code loosely coupled. We can further reuse this code in other parts of the application, which has been using older data format to migrate towards latest format. *LegacyVehicleDataAdapter* internally uses another adapter named *EngineAdapter* which encapsulates the logic of mapping older engine types to new types.

 **Adapter pattern** allows for the reuse of existing classes or components that have incompatible interfaces. By creating adapters, these classes can be integrated into new systems without requiring modifications to their original code. It bridges the gap between disparate interfaces, ensuring that they can communicate and collaborate effectively. This promotes code reuse and avoids the need to duplicate functionality.

 Following are some of the considerations we must keep in mind while using Adapter Pattern:

1. **Cost vs. Benefit:** When deciding whether to use an adapter, weigh the benefits it provides against the associated costs. If the incompatibility between interfaces is causing significant issues or hindering the development process, the benefits of using an adapter may outweigh the costs. However, if the differences are minimal or the integration is straightforward, an adapter may not be necessary.
2. **Performance Impact:** Adapters introduce an extra layer of indirection, which can potentially impact performance. The delegation from the adapter to the adapted object adds a small overhead in terms of execution time and memory. If performance is a critical concern in your specific use case, carefully evaluate the impact of using an adapter and consider alternatives if necessary.

5.2. Bridge Pattern



Bridge Pattern is used to decouple the abstraction and implementation hierarchies, enabling them to evolve independently. These hierarchies are then connected to each other via object composition, forming a bridge-like structure. This promotes loose coupling between the two, making it easier to modify or extend each hierarchy without affecting the other.

It achieves decoupling of the abstraction and implementation hierarchies by creating a bridge between the abstraction (abstraction interface) and its concrete implementations (implementor interface). It allows for a flexible relationship between the abstraction and the implementation. The abstraction can use different implementations at runtime by referencing different concrete implementor objects. This enables dynamic switching or selection of implementations based on the requirements or configuration.

Okay, enough of talking in silos. Let us understand the use case in terms of a business scenario.

We are developing a messaging system app which aims to provide a flexible and extensible platform for sending messages across different channels, such as emails and SMS. The system needs to support various formatting options for the messages, such as plain text and HTML. The Bridge pattern is used to separate the message sending functionality from the formatting logic, enabling easy integration of new message formats and channels.

If we try to implement the above scenario in an intuitive approach, we may create a base class *MessageSender* which would be extended by SMS and Email Sender classes, and SMS sender would use Text Message Formatter and Email Sender would use HTML Message Formatter (see [Figure 5-1](#)). Although, at first look, there is nothing wrong with this approach, however at looking closely we find a tight coupling scenario between Text Message Formatter and SMS sender and similarly for Email Sender use case with HTML formatter.



This approach lacks flexibility as each sender class is directly responsible for both sending the message and formatting it. This means that any changes or additions to the formatting logic would require modifications in each sender class, leading to code duplication and potential issues when maintaining or extending the system.



Bridge Pattern promotes **composition over inheritance** and helps decouple high level functionality from low level implementation.

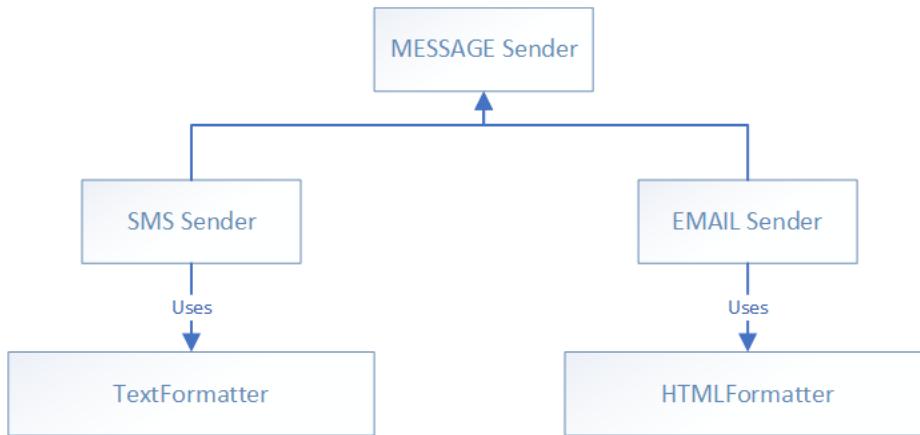


Figure 5-0-1

Now, let us modify the architecture (see [Figure 5-2](#)) using **Bridge Pattern** and checkout following code example.

```

/**
 * This abstract class represents a message sender.
 */
public abstract class MessageSender {
    protected MessageFormatter formatter;

    /**
     * Constructs a MessageSender object with the given formatter.
     *
     * @param formatter the formatter to be used for formatting messages
     */
    public MessageSender(MessageFormatter formatter) {
        this.formatter = formatter;
    }

    /**
     * Sends a message.
     *
     * @param message the message to be sent
     */
    public abstract void sendMessage(String message);
}

public interface MessageFormatter {
    String formatMessage(String message);
}

public class HTMLFormatter implements MessageFormatter {
    @Override
    public String formatMessage(String message) {

```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
// Perform plain text formatting
    return " <html><body>" + message + "</body></html>";
}
}

public class PlainTextFormatter implements MessageFormatter {
    @Override
    public String formatMessage(String message) {
        // Perform plain text formatting
        return "[Plain Text]" + message;
    }
}

/**
 * The SmsSender class is a concrete implementation of the MessageSender abstract class.
 * It is responsible for sending SMS messages using a specified message formatter.
 */
public class SmsSender extends MessageSender {
    public SmsSender(MessageFormatter formatter) {
        super(formatter);
    }

    @Override
    public void sendMessage(String message) {
        String formattedMessage = formatter.formatMessage(message);

        // Send SMS with the formatted message
        System.out.println("Sending SMS: " + formattedMessage);
    }
}

/**
 * The EmailSender class is a concrete implementation of the MessageSender abstract class.
 * It is responsible for sending email messages using a specified message formatter.
 */
public class EmailSender extends MessageSender {
    public EmailSender(MessageFormatter formatter) {
        super(formatter);
    }

    @Override
    public void sendMessage(String message) {
        String formattedMessage = formatter.formatMessage(message);

        // Send email with the formatted message
        System.out.println("Sending email: " + formattedMessage);
    }
}
```

 In this example, we have the *MessageSender* abstraction representing the functionality of sending messages. The *MessageFormatter* interface acts as the implementor, defining the low-level formatting functionality. The concrete implementor classes, *PlainTextFormatter* and *HtmlFormatter*, provide specific implementations of the formatting logic. The *EmailSender* and *SmsSender* classes act as refined abstractions, extending the *MessageSender* class and implementing the *sendMessage()* method using the appropriate formatting strategy. The *MessagingSystem* class demonstrates the usage of the Bridge pattern by creating instances of different senders with their respective formatters and invoking the *sendMessage()* method. The messages are formatted and sent according to the chosen formatting strategy. Bridge pattern enables the separation of the message sending functionality (*MessageSender*) from the formatting logic (*MessageFormatter*), allowing them to vary independently and providing flexibility and maintainability in the messaging system.

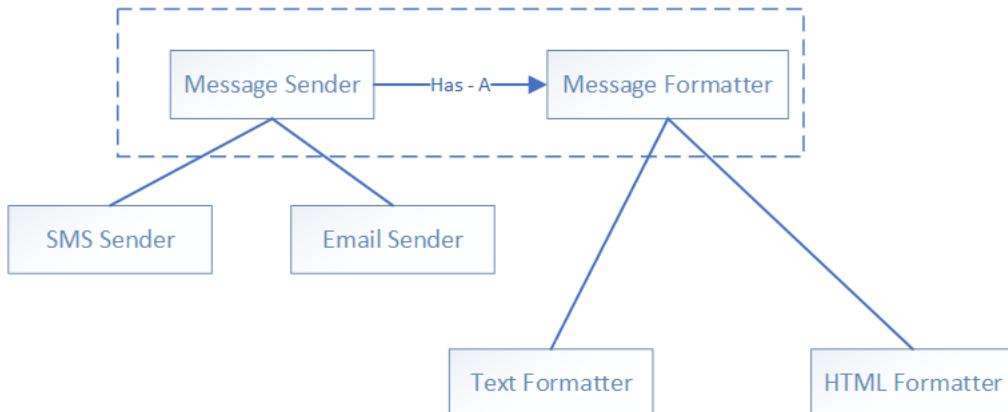


Figure 5-0-2

 Bridge pattern decouples the abstraction (high-level functionality) from the implementation (low-level details). It achieves this by using composition, which establishes a "**Has-A**" relationship between the abstraction and the implementation. It allows both to vary independently, reducing the dependencies between them. This promotes flexible design and makes it easier to modify or extend the system without impacting other parts of the codebase. The Bridge pattern aligns with the **Single Responsibility Principle (SRP)** by separating the responsibilities of the abstraction and implementation into separate classes. Each class has a clear and focused responsibility, making the codebase more modular and easier to understand. It also facilitates the adherence to other **SOLID** principles such as **Open-Closed Principle (OCP)** and **Dependency Inversion Principle (DIP)**.

 Designing the abstraction (abstraction interface or abstract class) requires careful consideration. It should focus on the essential high-level functionality and not become overly specific or too granular. Defining a proper abstraction is crucial to ensure a clear separation between the abstraction and implementation, facilitating future modifications and extensions.

 While the Bridge pattern and Object Adapter pattern may share some structural similarities, they have distinct intents and purposes. The Bridge pattern focuses on decoupling the abstraction (interface) from its implementation, allowing them to vary independently. It aims to provide a bridge between the abstraction and implementation, enabling them to evolve and change separately. The Bridge pattern promotes flexibility, extensibility, and the ability to switch implementations at runtime.

Adapter pattern is designed to adapt the interface of an existing object to make it compatible with another interface. It allows objects with incompatible interfaces to work together by wrapping the existing object with an adapter class. The adapter class translates the requests from the target interface into calls to the adapted object's interface.

5.3. Composite Pattern



Composite Pattern is used to arrange objects in tree like structure to represent part-whole hierarchies, where the whole can be composed of sub-parts, which in turn can be composed of further sub-parts, forming a recursive structure.

We often deal with hierarchical systems in our daily life. If we are going to Bank or Post office, they have different hierarchical system in place. A sports team will have its own hierarchy, even computers have directories and subdirectories. Graphic User Interfaces (GUIs) have components which can be structured in hierarchy. Hierarchies can be simple or complex and are often represented using tree like structure in diagrams. How do we represent such a system in Software Programs?

Consider a scenario where we are developing an employee management system for a large organization. The system needs to handle hierarchical relationships among employees, such as managers and their subordinates. Each employee can have their own set of properties and methods specific to their role.

Initially, we might design separate classes for each role, such as Manager, Supervisor, and Employee, with their respective attributes and behaviours. However, as the organization grows and the hierarchy becomes more complex, we realize that the existing design is not scalable and leads to tightly coupled code. Say, if we wanted to perform an operation on a group of employees, we would need to write separate code to handle each level of the hierarchy, leading to tightly coupled and duplicated logic.



Hence, we leverage **composite pattern** to represent the hierarchical structure of employees as a tree-like structure, where we can perform operations on individual employees or groups of employees without the need for separate code paths.



Let us look at the code snippet below to understand better.

```
import java.util.ArrayList;
import java.util.List;

// Component interface
interface EmployeeComponent {
    void displayInformation();
}

// Leaf class representing an individual employee
class Employee implements EmployeeComponent {
    private String name;
    private String role;

    public Employee(String name, String role) {
        this.name = name;
    }

    @Override
    public void displayInformation() {
        System.out.println("Employee Name: " + name + ", Role: " + role);
    }
}
```

```
this.role = role;
}

public void displayInformation() {
    System.out.println("Employee: " + name + ", Role: " + role);
}
}

// Composite class representing a supervisor and their subordinates (employees)
class Supervisor implements EmployeeComponent {
    private String name;
    private List<EmployeeComponent> subordinates;

    public Supervisor(String name) {
        this.name = name;
        this.subordinates = new ArrayList<>();
    }

    public void addSubordinate(EmployeeComponent subordinate) {
        subordinates.add(subordinate);
    }

    public void removeSubordinate(EmployeeComponent subordinate) {
        subordinates.remove(subordinate);
    }

    public void displayInformation() {
        System.out.println("Supervisor: " + name);
        System.out.println("Subordinates:");
        for (EmployeeComponent subordinate : subordinates) {
            subordinate.displayInformation();
        }
    }
}

// Composite class representing a manager and their subordinates (supervisors)
class Manager implements EmployeeComponent {
    private String name;
    private List<EmployeeComponent> subordinates;

    public Manager(String name) {
        this.name = name;
        this.subordinates = new ArrayList<>();
    }

    public void addSubordinate(EmployeeComponent subordinate) {
        subordinates.add(subordinate);
    }
}
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
public void removeSubordinate(EmployeeComponent subordinate) {
    subordinates.remove(subordinate);
}

public void displayInformation() {
    System.out.println("Manager: " + name);
    System.out.println("Subordinates:");
    for (EmployeeComponent subordinate : subordinates) {
        subordinate.displayInformation();
    }
}

// Client code
public class EmployeeManagementSystem {
    public static void main(String[] args) {
        // Creating individual employees
        Employee john = new Employee("John Smith", "Developer");
        Employee anna = new Employee("Anna Johnson", "QA Engineer");

        // Creating a supervisor and adding employees
        Supervisor supervisor = new Supervisor("Michael Scott");
        supervisor.addSubordinate(john);
        supervisor.addSubordinate(anna);

        // Creating a manager and adding a supervisor
        Manager manager = new Manager("David Wallace");
        manager.addSubordinate(supervisor);

        // Displaying information
        manager.displayInformation();
    }
}
```

 If we look at the code above closely, we will observe that `displayInformation()` method, when called on a manager object, would recursively call the same method on all its subordinate in a hierarchical order. Also, this tree structure allows flexibility for a manager to add either employee or Supervisor as its child

components (see [Figure 5-3](#)).

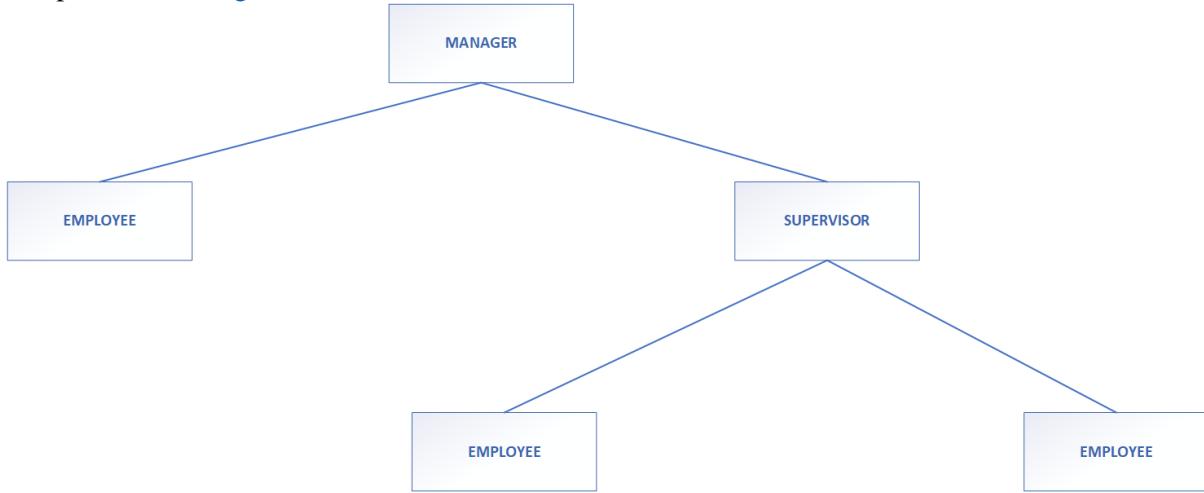


Figure 5-0-3

Composite Pattern has three key elements:

1. **Component:** The Component represents the common interface or base class for all elements in the hierarchy, whether they are individual objects or compositions of objects. It defines the basic operations that can be performed on the elements, such as adding, removing, or accessing child elements. In above example, *EmployeeComponent* is component element.
2. **Leaf:** The Leaf is the implementation of the Component interface for individual objects that do not have any child elements. Leaf objects represent the building blocks or basic elements of the hierarchy. In above example, *Employee* is a Leaf element.
3. **Composite:** The Composite represents the implementation of the Component interface for composite objects. Composite objects are containers that can hold other components, including both leaf objects and other composite objects. They delegate operations to their child components, recursively traversing the tree structure. In above example, *Manager* and *Supervisor* are composite elements.



Composite Pattern enables the creation of complex structures that can be treated uniformly, simplifying the code, and making it more flexible and scalable. It promotes code reusability by allowing the use of the same operations and interfaces for individual objects and compositions. Additionally, the Composite pattern supports recursive traversal and operations on the entire structure, enabling efficient manipulation and processing of complex hierarchies. It also allows for dynamic additions or removals of objects from the structure, providing flexibility in managing and modifying the composition.

5.4. Decorator Pattern



Decorator pattern is a structural design pattern that allows behavior to be added to an individual object dynamically and offer a flexible alternative to subclassing when it comes to extending the functionality of an object.

Extension of functionality is a common business use case in real world. Successful products keep extending their functionalities to be ahead in the game. Small businesses keep extending their functionality to survive and thrive. And when we integrate their requirements in software development,

we encounter frequent request of functionality addition or deletion. If something is not working, it should be removed quickly and if there is something new, which when added upon existing functionality, boosts sales, it should be integrated in a jiffy.



And this is where Decorator Patterns come in handy. They help in extending functionality of an object without altering or affecting the existing behavior. They also help in giving an option for mix and match for choosing functionality on run time.

Okay, enough of theory and let us consider a potential business scenario and then see its implementation use case.

Imagine a coffee shop wants to implement a custom coffee ordering system to cater to the unique preferences of its customers. The system should allow customers to create personalized coffee orders by selecting various options such as type of coffee, additional ingredients, and toppings.

So, we have a base entity *Coffee* which needs to be customized and as per customization its features such as cost and description would change. It makes sense, right? A simple coffee would cost differently than a milk coffee or a cream whipped coffee.



If we adhere to traditional approach to address this requirement (see [Figure 5-4](#)), we may end up creating new subclasses, say *MilkCoffee* and *WhippedCreamCoffee* in our system. And when a customer asks for Milk Coffee with whipped cream, then what do we do? Create another subclass? And when our system has many such subclasses, then imagine the chaos of subclasses to be maintained. Something is wrong with this approach and we will fix it right after a coffee break! ☕

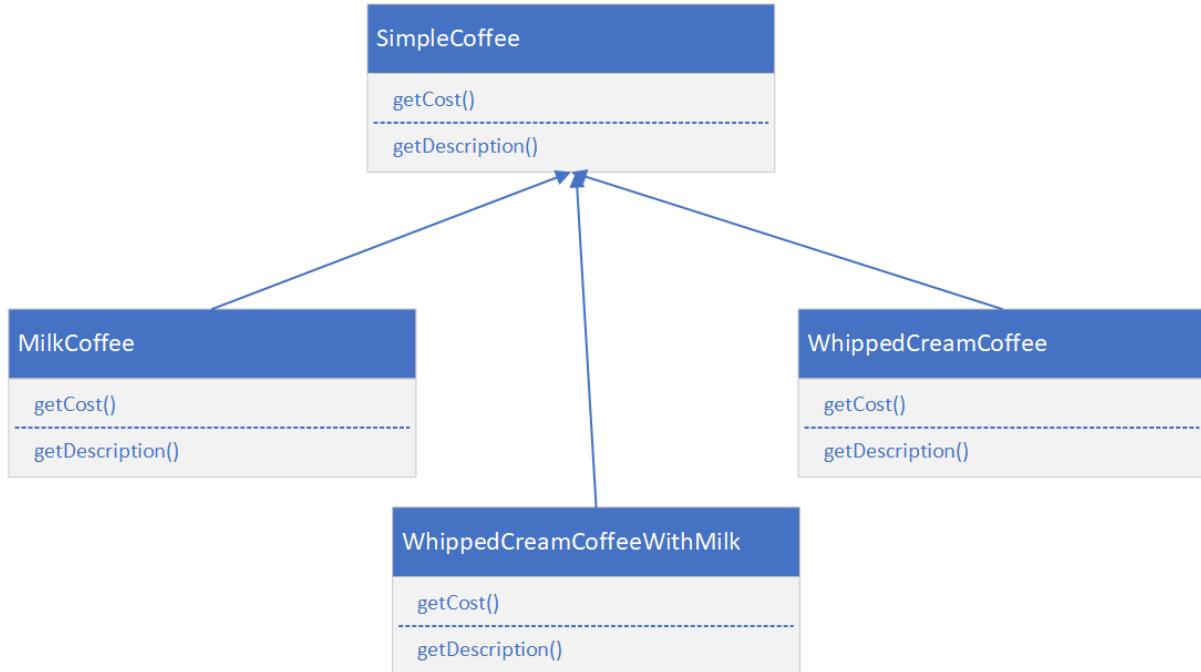


Figure 5-0-4

 To allow dynamic addition of functionalities at run time, **Decorator Pattern** is used. By using decorators, we can wrap the original object with one or more decorators, each providing a specific extension or modification to the object's behavior. This approach avoids the need for an excessive number of subclasses and reduces the complexity of the class hierarchy.

Furthermore, decorators follow the principle of **composition over inheritance**, as they allow objects to be composed of multiple decorators, each responsible for a specific aspect of functionality. This composition-based approach promotes code reusability, modularity, and maintainability.

 We will see a code snippet which leverages Decorator Pattern to give us flexibility to extend functionality of our system dynamically and it also does not alter existing functionality. Cool!

```
/*
 * The Coffee interface represents a type of coffee.
 * It provides methods to get the cost and description of the coffee.
 */
public interface Coffee {
    /**
     * Returns the cost of the coffee.
     *
     * @return the cost of the coffee as a double value
     */
    double getCost();

    /**
     * Returns the description of the coffee.
     *
     * @return the description of the coffee as a String
     */
    String getDescription();
}

/**
 * The SimpleCoffee class represents a simple type of coffee.
 * It implements the Coffee interface and provides the implementation for its methods.
 */
public class SimpleCoffee implements Coffee {
    @Override
    public double getCost() {
        return 1.0;
    }

    @Override
    public String getDescription() {
        return "Simple Coffee";
    }
}

/*
 * The CoffeeDecorator class is an abstract class that represents a decorator for a Coffee object.
*/
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
* It implements the Coffee interface and provides a common implementation for its methods.
* Subclasses of CoffeeDecorator can add additional behavior to the decorated coffee object.
*/
public abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee decoratedCoffee) {
        this.decoratedCoffee = decoratedCoffee;
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }
}

/**
 * The MilkDecorator class represents a decorator that adds milk to a Coffee object.
 * It extends the CoffeeDecorator class and provides additional behavior for the decorated coffee.
*/
public class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    @Override
    public double getCost() {
        return super.getCost() + 0.5;
    }

    @Override
    public String getDescription() {
        return super.getDescription() + ", Milk";
    }
}

/**
 * The WhippedCreamDecorator class represents a decorator that adds WhippedCream to a Coffee object.
 * It extends the CoffeeDecorator class and provides additional behavior for the decorated coffee.
*/
public class WhippedCreamDecorator extends CoffeeDecorator {
    public WhippedCreamDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }
}
```

```
@Override
public double getCost() {
    return super.getCost() + 0.75;
}

@Override
public String getDescription() {
    return super.getDescription() + ", Whipped Cream";
}
}

/**
 * The Client class demonstrates the usage of the Coffee decorators.
 * It creates instances of different coffee objects and prints their cost and description.
 */
public class Client {
    public static void main(String[] args) {
        Coffee simpleCoffee = new SimpleCoffee();
        System.out.println("Cost: " + simpleCoffee.getCost() + ", Description: " + simpleCoffee.getDescription());

        Coffee milkCoffee = new MilkDecorator(simpleCoffee);
        System.out.println("Cost: " + milkCoffee.getCost() + ", Description: " + milkCoffee.getDescription());

        Coffee whippedCreamCoffee = new WhippedCreamDecorator(milkCoffee);
        System.out.println(
            "Cost: " + whippedCreamCoffee.getCost() + ", Description: " + whippedCreamCoffee.getDescription());
    }
}
```

 In the code example given above, the Decorator pattern is implemented using the *CoffeeDecorator* abstract class and its concrete subclasses such as *MilkDecorator* and *WhippedCreamDecorator*.

The *CoffeeDecorator* class acts as a wrapper for the Coffee object and implements the Coffee interface. It holds a reference to the decorated coffee object (*decoratedCoffee*) and delegates calls to the underlying object for methods such as *getCost()* and *getDescription()*. This allows the decorator to modify or extend the behavior of the decorated object.

Each concrete decorator class, such as *MilkDecorator* and *WhippedCreamDecorator*, extends the *CoffeeDecorator* class. These decorators add specific functionalities by overriding the *getCost()* and *getDescription()* methods and providing their own behavior. They also make use of the **super** keyword to call the corresponding methods of the decorated object.

 **Decorators** allow for the dynamic addition of new functionalities to an object at runtime. We can wrap the original *Coffee* object with one or more decorators, adding or removing functionality as needed. This dynamic composition provides flexibility without modifying the original object or affecting other objects using the same interface. This enables fine-grained control over the behavior and allows for flexible customization. Adhering to SRP, each decorator class has a single responsibility, adding or

modifying a specific behavior. This promotes code modularity and separation of concerns, making the system easier to understand, modify, and maintain.



Decorators wrap the original object, adding or modifying its behavior, but they do not change the underlying object's identity. The decorated component and the decorator are distinct objects, even though the decorator delegates most of its functionality to the underlying component.



When using the Decorator pattern extensively, it is common for systems to be composed of many small objects that appear similar. The differences between these objects lie primarily in their interconnections and not in their class or variable values.

This composition of numerous interconnected objects can make the system complex and potentially challenging to understand, learn, and debug, especially for developers who are new to the codebase. The similarities between the objects, combined with the dynamic nature of decorators, can introduce additional complexity and make it harder to trace and debug issues.

5.5. Facade Pattern



Façade Pattern provides a simplified interface to a library, a framework, or complex subsystem of classes, making it easier to use and understand. It acts as a high-level interface that hides the complexities of the underlying system and provides a unified and simplified interface for clients to interact with.

Imagine using a coffee vending machine after a tiring day to treat yourself to a cup of coffee. Instead of having to go through the individual steps of pouring milk, adding coffee, and selecting flavors, the vending machine provides a simplified interface through a single button on its GUI. By pressing the button for a specific flavored coffee, the vending machine automatically handles the process of pouring milk, adding coffee, and incorporating the chosen flavor. This simplified interface acts as a **Facade**, abstracting away the complexities of the internal workings of the machine and providing a convenient and streamlined experience for the user.

Let us take another example in terms of software development.

Imagine we have a powerful and complex sound editing software that offers a vast array of features for professional audio production. However, we are working on a simple mobile app that only needs to perform basic audio trimming and volume adjustment for user-generated content.



In this scenario, integrating the entire sound editing software directly into our mobile app would be overkill and could lead to unnecessary complexity. Users of mobile app would be exposed with lot many features which they have no use of and may end getting confused. We want to keep our interface clean and simple to use and focus on the main goal which is basic audio trimming and volume adjustment.



Façade Pattern can be used here to shift focus of the app back to the specific functionalities it needs, making it more lightweight, easier to maintain, and user-friendly. It shields the app from the complexities of the extensive library, acting as a convenient interface for the limited scope of audio editing needed for our app.



Let us look at the sample code snippet to understand better.

```
/*
 * This class represents a sound editing software that provides various functionalities
 * for manipulating audio files.
 */

public class SoundEditingSoftware {
    /**
     * Opens the specified audio file.
     *
     * @param filePath the path of the audio file to be opened
     */
    public void openFile(String filePath) {
        System.out.println("Opening file: " + filePath);
        // Perform complex operations to open the audio file
    }

    /**
     * Trims the audio file from the specified start time to the specified end time.
     *
     * @param startTime the start time in seconds
     * @param endTime   the end time in seconds
     */
    public void trimAudio(double startTime, double endTime) {
        System.out.println("Trimming audio from " + startTime + "s to " + endTime + "s");
        // Perform complex operations to trim the audio
    }

    /**
     * Adjusts the volume of the audio by the specified percentage.
     *
     * @param percentage the percentage by which to adjust the volume
     */
    public void adjustVolume(int percentage) {
        System.out.println("Adjusting volume by " + percentage + "%");
        // Perform complex operations to adjust the audio volume
    }

    // Other complex functionalities of the sound editing software
}

/*
 * This class represents a facade for a sound editing software, providing simplified methods
 * for trimming audio files and adjusting their volume.
 */

public class SoundEditingFacade {
    private SoundEditingSoftware software;

    /**
     * Constructs a new SoundEditingFacade object.
     * Initializes the underlying SoundEditingSoftware instance.
    
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
/*
public SoundEditingFacade() {
    software = new SoundEditingSoftware();
}

/**
 * Trims the audio file located at the specified filePath from the given startTime to the endTime.
 *
 * @param filePath the path of the audio file to be trimmed
 * @param startTime the start time in seconds
 * @param endTime the end time in seconds
 */

public void trimAudio(String filePath, double startTime, double endTime) {
    software.openFile(filePath);
    software.trimAudio(startTime, endTime);
}

/**
 * Adjusts the volume of the audio file located at the specified filePath by the given percentage.
 *
 * @param filePath the path of the audio file to have its volume adjusted
 * @param percentage the percentage by which to adjust the volume
 */

public void adjustVolume(String filePath, int percentage) {
    software.openFile(filePath);
    software.adjustVolume(percentage);
}

/**
 * This class represents a mobile app that demonstrates the usage of the SoundEditingFacade class
 * to trim audio files and adjust their volume.
 */

public class MobileApp {
    /**
     * The main method that is executed when running the MobileApp.
     * It creates an instance of SoundEditingFacade and demonstrates the usage of its methods.
     *
     * @param args the command line arguments (not used in this example)
     */
    public static void main(String[] args) {
        SoundEditingFacade soundEditor = new SoundEditingFacade();
        String filePath = "audio_file.mp3";
        double startTime = 10.5;
        double endTime = 30.0;
        soundEditor.trimAudio(filePath, startTime, endTime);
        int volumePercentage = 50;
        soundEditor.adjustVolume(filePath, volumePercentage);
    }
}
```

 In this code example, we have a *SoundEditingSoftware* class representing a complex sound editing library with various functionalities. The *SoundEditingFacade* acts as a simplified interface to the library, providing methods like *trimAudio()* and *adjustVolume()* that internally delegate the operations to the *SoundEditingSoftware*.

The *MobileApp* class demonstrates the usage of the Facade pattern in the mobile app. It creates an instance of *SoundEditingFacade* and uses its methods to perform audio trimming and volume adjustment. The app can utilize the necessary sound editing features without being exposed to the complexities of the underlying sound editing software.

 **Facade pattern** is a useful design pattern in various scenarios where you need to simplify and provide a unified interface to a complex subsystem. By encapsulating the complexities behind a Facade, clients can interact with the system using a straightforward and intuitive API, without needing to understand the intricate details of the underlying components. It improves code maintainability by providing a single-entry point for accessing the subsystem functionality. Changes or updates to the subsystem can be localized within the Facade implementation, without requiring modifications to the client code.

 When used in conjunction with the Facade pattern, the Abstract Factory pattern helps to provide an interface for creating subsystem objects in a subsystem-independent manner. The Facade acts as a simplified interface to the complex subsystem, while the Abstract Factory abstracts the creation of subsystem objects behind an interface. This combination allows clients to interact with the subsystem through the Facade without being concerned about the specific implementation details or platform-specific classes. And in cases, when only one Façade object is required, we can create Façade objects as Singletons.

5.6. Flyweight Pattern

 **Flyweight Pattern** aims to optimize memory usage by sharing common data between multiple objects.

Consider a game that requires rendering a large number of trees on a game map. The trees can be of different types, such as pine trees, oak trees, and birch trees. Each tree type has a specific appearance, but they share common properties like size and position on the map. hence, each individual tree object would store its complete state, including properties such as size, position, and appearance. Initially we go ahead and create objects as and when required in the game and it works pretty well on our computer or phone. However, once we ship it for other users to explore this game and play, we get reports of system crashes. What went wrong?

 As the game map expands and more trees are added, the memory usage of the system would skyrocket. Each tree instance would consume a significant amount of memory, resulting in unnecessary duplication of data. The performance of the system would suffer due to the increased memory overhead and the need to create and manage many individual tree objects. Rendering each tree would require redundant computations and memory accesses, leading to slower frame rates and a less responsive gameplay experience. If we extend the system with new tree types, system will become more complex and error-prone, as each new type would require the creation of additional individual tree objects.



Flyweight pattern can be used to optimize memory usage and improve performance. The key idea behind the Flyweight pattern is to split an object into two parts: **intrinsic state** and **extrinsic state**. The intrinsic state represents the shared data that can be shared among multiple objects, while the extrinsic state represents the unique data that varies for each object.

By separating the intrinsic and extrinsic state, the Flyweight pattern allows multiple objects to share the same intrinsic state, reducing memory consumption. The intrinsic state is typically stored in a flyweight factory or flyweight pool, which manages the creation and sharing of flyweight objects. The client objects use the flyweight objects by providing or manipulating the extrinsic (unique) state. Clients typically interact with the flyweight factory to obtain the required flyweight objects.



Let us see a code snippet to understand it further.

```
import java.awt.Color;
import java.util.HashMap;
import java.util.Map;

/**
 * Flyweight interface representing a tree.
 */
interface Tree {
    /**
     * Renders the tree at the specified coordinates.
     *
     * @param x The x-coordinate of the tree.
     * @param y The y-coordinate of the tree.
     */
    void render(int x, int y);
}

/**
 * Concrete flyweight class representing a specific type of tree (PineTree).
 */
class PineTree implements Tree {
    private final Color color;

    /**
     * Constructs a PineTree object.
     * Assume pine trees have a specific color (GREEN).
     */
    public PineTree() {
        color = Color.GREEN;
    }

    /**
     * Renders the pine tree at the specified coordinates with its color.
     *
     * @param x The x-coordinate of the tree.
     * @param y The y-coordinate of the tree.
     */
    @Override
```

```
public void render(int x, int y) {
    System.out.println("Rendering a pine tree at (" + x + ", " + y + ") with color " + color);
}

/*
 * Flyweight factory class responsible for creating and managing flyweight objects.
 */

class TreeFactory {
    private final Map<String, Tree> treeCache;

    /**
     * Constructs a TreeFactory object.
     * Initializes the tree cache.
     */
    public TreeFactory() {
        treeCache = new HashMap<>();
    }

    /**
     * Retrieves a tree object from the cache based on the specified type.
     * If the tree object does not exist in the cache, a new one is created and added to the cache.
     *
     * @param type The type of tree to retrieve.
     * @return The tree object.
     */
    public Tree getTree(String type) {
        Tree tree = treeCache.get(type);
        if (tree == null) {
            // Create a new tree instance and add it to the cache
            if (type.equals("pine")) {
                tree = new PineTree();
            }
            // Additional tree types can be added here
            treeCache.put(type, tree);
        }
        return tree;
    }
}

/*
 * Client class that uses flyweight objects.
 */

class Game {
    private final TreeFactory treeFactory;

    /**
     * Constructs a Game object with a TreeFactory.
     *
     * @param treeFactory The TreeFactory object to use for creating trees.
     */
    public Game(TreeFactory treeFactory) {
        this.treeFactory = treeFactory;
    }
}
```

```

}

/**
 * Renders a tree of the specified type at the specified coordinates using the TreeFactory.
 *
 * @param type The type of tree to render.
 * @param x The x-coordinate of the tree.
 * @param y The y-coordinate of the tree.
 */
public void renderTree(String type, int x, int y) {
    Tree tree = treeFactory.getTree(type);
    tree.render(x, y);
}
}

/**
 * An example usage of the Flyweight pattern.
 */
public class FlyweightExample {
    /**
     * The main entry point of the program.
     *
     * @param args The command line arguments.
     */
    public static void main(String[] args) {
        TreeFactory treeFactory = new TreeFactory();
        Game game = new Game(treeFactory);
        // Render multiple pine trees at different locations
        game.renderTree("pine", 10, 20);
        game.renderTree("pine", 30, 40);
        game.renderTree("pine", 50, 60);
        // Additional tree types can be rendered here
    }
}

```

 In above code example, we have a *Tree* interface representing the flyweight objects, and a concrete flyweight class *PineTree* that implements the interface. The *TreeFactory* class acts as the flyweight factory, responsible for creating and managing the flyweight objects. The *Game* class is the client that uses the flyweight objects to render trees.

When the *renderTree* method is called in the *Game* class, it retrieves the flyweight object from the *TreeFactory* based on the tree type. If the object already exists in the cache (see usage of *HashMap*), it is retrieved and used. If not, a new flyweight object is created, added to the cache, and then used. The flyweight objects are then rendered with their respective locations.

 By utilizing the Flyweight pattern, the common state (in this case, the color of the tree) is shared among multiple tree objects, resulting in memory efficiency. The flyweight objects are created and cached by the factory, allowing them to be reused when needed. It also reduced garbage collection overhead. It promotes a clear separation of concerns by separating the intrinsic and extrinsic state. This makes the codebase more maintainable and easier to understand.



Since, we are always returning the used instance of a flyweight object, we might confuse it with Singleton pattern. However, the Flyweight pattern and the Singleton pattern are distinct design patterns with different purposes and behaviours. The Flyweight pattern is focused on optimizing memory usage by sharing common data among multiple objects. On the other hand, the Singleton pattern is concerned with ensuring that only a single instance of a class exists throughout the application's lifetime. While both patterns can have a single instance, their purposes and usage differ.

The combination of the Flyweight and Composite patterns can be seen in scenarios where the Composite pattern is used to represent a logically hierarchical structure, and the Flyweight pattern is applied to share leaf nodes within that structure. This combination helps to minimize memory usage by reusing shared leaf nodes across different composite nodes.

By utilizing the Flyweight pattern for the leaf nodes, we can ensure that common leaf nodes are shared among multiple composite nodes, reducing memory consumption. This is particularly beneficial in cases where the leaf nodes have significant shared state or when the number of leaf nodes is large. It optimizes memory usage by reusing common leaf nodes and provides a unified interface for working with both individual leaf nodes and composite nodes.

5.7. Proxy Pattern



Proxy Pattern, as the name suggests, provides with a substitute or placeholder for another object. It allows us to control the access to the target object, providing additional functionalities or behaviors without changing the original object's interface.



But why would we proxy or substitute another object?

Answer is quite intuitive. If we could afford to carry an original object all the time, then we would have carried it in first place rather than proxying. But the reason we are proxying it suggests that it is costly to carry the burden of the object all the time. Still confused?

Say in a remote communication scenario, a client needs to interact with a remote object over a network. In a distributed environment, where different components are located on different servers, we take example of one such server, File Server that stores and manages files. The clients need to interact with the file server to perform file operations like reading, writing, and deleting files. It consumes good amount of system resources. We need instance of file server from time to time but not all the time. How do we manage to do that?



We can do **lazy initialization**, an approach where an object is created or initialized only when it is actually needed. However, this initialization code would need to be implemented in all the clients of File Server, which would result in code duplication.



Proxy pattern help us by letting us implement the same interface as the original service object. However, the proxy does not create a new real service object upon receiving a request from a client. Instead, the proxy intercepts the client's request and can perform additional operations before or after delegating the actual work to the real service object.

The Proxy pattern involves the following components:

1. **Subject:** This is the interface or abstract class that defines the common interface between the Proxy and the target object. It specifies the methods that the Proxy should implement to mimic the behavior of the target object.
2. **Real Subject:** This is the actual object that the Proxy represents. It implements the Subject interface and contains the core functionality that the Proxy wraps.
3. **Proxy:** This is the surrogate object that acts as a substitute for the Real Subject. It implements the Subject interface and maintains a reference to the Real Subject. The Proxy intercepts client requests and performs additional actions if required, and then delegates the request to the Real Subject.



Let us check a code snippet to understand this implementation.

```
/**  
 * This interface represents a file server that allows reading, writing, and deleting files.  
 */  
public interface FileServer {  
    /**  
     * Reads the contents of a file with the specified filename.  
     *  
     * @param filename the name of the file to be read  
     */  
    void readFile(String filename);  
    /**  
     * Writes the given data to a file with the specified filename.  
     *  
     * @param filename the name of the file to be written  
     * @param data the data to be written to the file  
     */  
    void writeFile(String filename, byte[] data);  
    /**  
     * Deletes the file with the specified filename.  
     *  
     * @param filename the name of the file to be deleted  
     */  
    void deleteFile(String filename);  
}  
/**  
 * This class represents a remote file server that implements the {@link FileServer} interface.  
 * It provides methods to read, write, and delete files on the remote server.  
 */  
public class RemoteFileServer implements FileServer {  
    /**  
     * Reads the contents of a file from the remote file server.  
     *  
     * @param filename the name of the file to be read  
     */  
    @Override  
    public void readFile(String filename) {
```

```
// Connect to the remote file server and perform the read operation
System.out.println("Reading file: " + filename);
}

/**
 * Writes the given data to a file on the remote file server.
 *
 * @param filename the name of the file to be written
 * @param data the data to be written to the file
 */
@Override
public void writeFile(String filename, byte[] data) {
    // Connect to the remote file server and perform the write operation
    System.out.println("Writing file: " + filename);
}

/**
 * Deletes a file from the remote file server.
 *
 * @param filename the name of the file to be deleted
 */
@Override
public void deleteFile(String filename) {
    // Connect to the remote file server and perform the delete operation
    System.out.println("Deleting file: " + filename);
}

/**
 * This class represents a proxy for the {@link FileServer} interface.
 * It provides a convenient way to interact with a remote file server located at the specified server address.
 * The proxy lazily initializes the remote file server and forwards read, write, and delete operations to it.
 */
public class FileServerProxy implements FileServer {
    private final String serverAddress;
    private RemoteFileServer remoteFileServer;

    /**
     * Constructs a new FileServerProxy with the specified server address.
     *
     * @param serverAddress the address of the remote file server
     */
    public FileServerProxy(String serverAddress) {
        this.serverAddress = serverAddress;
    }

    /**
     * Reads the contents of a file from the remote file server.
     *
     * @param filename the name of the file to be read
     */
    @Override
    public void readFile(String filename) {
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
// Lazy initialization of the remote file server
if (remoteFileServer == null) {
    remoteFileServer = new RemoteFileServer();
}
// Forward the read operation to the remote file server
remoteFileServer.readFile(filename);
}

/**
 * Writes the given data to a file on the remote file server.
 *
 * @param filename the name of the file to be written
 * @param data the data to be written to the file
 */
@Override
public void writeFile(String filename, byte[] data) {
    // Lazy initialization of the remote file server
    if (remoteFileServer == null) {
        remoteFileServer = new RemoteFileServer();
    }
    // Forward the write operation to the remote file server
    remoteFileServer.writeFile(filename, data);
}

/**
 * Deletes a file from the remote file server.
 *
 * @param filename the name of the file to be deleted
 */
@Override
public void deleteFile(String filename) {
    // Lazy initialization of the remote file server
    if (remoteFileServer == null) {
        remoteFileServer = new RemoteFileServer();
    }
    // Forward the delete operation to the remote file server
    remoteFileServer.deleteFile(filename);
}

/**
 * This class represents a client that interacts with a file server.
 * It demonstrates the usage of the {@link FileServer} interface and its proxy implementation.
 */
public class Client {
    /**
     * The entry point of the client application.
     *
     * @param args the command line arguments
     */
    public static void main(String[] args) {
```

```
// Create a file server proxy with the specified server address
FileServer fileServer = new FileServerProxy("192.168.0.100");
// Read a file from the file server
fileServer.readFile("document.txt");
// Write data to a file on the file server
fileServer.writeFile("data.txt", new byte[] { 1, 2, 3 });
// Delete a file from the file server
fileServer.deleteFile("image.jpg");
}
```

}

}

 In the code example above, *FileServer* is the **Subject** interface defining the file operations that clients can perform. *RemoteFileServer* is the **Real Subject**, which represents the actual file server running on a remote machine. *FileProxyServer* is the **Proxy**, which acts as a local representative for the remote file server. The Proxy lazily initializes the remote file server and forwards the file operations to the real server when requested by clients. The clients can interact with the file server through the Proxy, without being aware of the network communication and remote object details.



By using the **Proxy pattern**, we can centralize the initialization logic and avoid code duplication unless caused by lazy initialization. The Proxy can handle the deferred initialization code, ensuring that it is executed only when the object is actually needed.

Key benefit of using Proxy is reducing the overhead of object creation or resource allocation until necessary. It adds access control mechanisms to the Real Subject, ensuring that certain operations are performed only by authorized clients.

It provides an additional layer of security by enforcing authentication, authorization, or other security checks. It captures method invocations and performs logging or auditing operations, such as recording method calls, collecting performance metrics, or logging error conditions.

It enhances the system's observability and allows for monitoring and analysis of the target object's behavior. Caching mechanism can be employed along side proxy to cache the results of expensive operations performed by the Real Subject, allowing subsequent requests with the same inputs to be served from the cache.



Even though Decorator pattern and proxy Pattern may have similar implementation, their purpose is entirely different. Decorator is focused on adding additional responsibilities or behaviors to an object dynamically. Whereas, Proxy is primarily concerned with controlling access to an object. Both patterns involve wrapping an object and providing additional functionality, but their intentions and primary focuses differ.

5.8. Summary

Structural design patterns are concerned with organizing classes and objects to form larger structures, focusing on relationships and interactions between them. These patterns help in achieving flexibility, modularity, and extensibility in software systems.

1. **Adapter Pattern:** Converts the interface of a class into another interface that clients expect. It allows incompatible classes to work together by acting as a bridge between them.

2. **Bridge Pattern:** Separates an abstraction from its implementation, allowing them to vary independently. It decouples the abstraction and implementation hierarchies, promoting flexibility and extensibility.
3. **Composite Pattern:** Composes objects into tree-like structures to represent part-whole hierarchies. It allows clients to treat individual objects and groups of objects uniformly.
4. **Decorator Pattern:** Dynamically adds responsibilities or behaviors to objects without modifying their underlying classes. It provides a flexible alternative to subclassing for extending functionality.
5. **Facade Pattern:** Provides a simplified interface to a complex subsystem, acting as a high-level interface that makes the subsystem easier to use. It hides the complexities of the subsystem and provides a unified interface for clients.
6. **Flyweight Pattern:** Shares common state among multiple objects to minimize memory usage. It aims to optimize performance and memory consumption by reusing shared state instead of creating new instances.
7. **Proxy Pattern:** Controls access to an object by acting as a surrogate or placeholder. It provides a level of indirection, allowing additional operations to be performed before or after delegating to the real object.

Behavioral Design Patterns

“

“Behaviour is a mirror in which everyone displays his own image.” — Johann Wolfgang von Goethe



Behavioral design patterns are concerned with the interaction and communication between objects, focusing on how objects collaborate and fulfil specific behaviors or responsibilities.

In the study of design patterns, we have explored both creational and structural design patterns in relation to objects. Creational design patterns focus on creating objects in a flexible and efficient manner. These patterns provide us with strategies to instantiate objects, manage their lifecycle, and ensure their proper creation and initialization.

Moving on to structural design patterns, we focused on how classes and objects are composed to form larger structures. These patterns help us organize objects and classes, manage relationships between them, and add additional functionalities while maintaining flexibility and modularity.

Now our focus will shift to understanding **how objects interact, communicate, and behave** in different scenarios.

Say, in a workflow management system, there are often multiple steps or stages that need to be executed in a specific order, with various conditions and dependencies between them. Managing this control flow and coordinating the interactions between different components can become quite complex.

Behavioral design patterns can be applied to simplify the control flow and handle the interconnections between objects in a more structured manner.

By exploring behavioral design patterns, we will gain insights into effectively managing object interactions, designing flexible systems, and encapsulating behavior. These patterns will equip us with techniques to enhance the communication and behavior of our objects, resulting in more modular, maintainable, and adaptable software designs.

6.1. Chain of Responsibility



Chain of Responsibility pattern is a behavioral design pattern that allows an object to pass a request along a chain of potential handlers until the request is handled or processed by an appropriate handler.

Have you ever been to a bank for approval of a Loan or to a government office for getting some paperwork done? If yes, you have already observed Chain of Responsibility in action. Whenever a request is initiated without mentioning an explicit receiver, this pattern is utilized.

Say an organization may have a policy that requires purchases above a certain threshold to go through a hierarchical approval process. When an employee submits a purchase request, the approval system is triggered. The request flows through a chain of approvers, starting from the lowest level (e.g., Manager),

then progressing to higher levels (e.g., Director, CEO) based on the purchase amount. Each approver in the chain reviews the request, checks its compliance with procurement guidelines, and approves or rejects it accordingly. The system tracks the progress of the approval process, maintains an audit trail, and notifies the relevant parties at each step.

 If we go ahead and implement it traditionally then the client code would need to have direct knowledge of the approval limits and the order in which the approvers should be consulted. This tight coupling can make the code harder to maintain and modify, as any changes in the approval process would require modifications to the client code. The client code would need to explicitly handle each purchase request and determine the approver based on their approval limit. This could lead to duplicated code across different parts of the system where purchase requests are processed. As the number of approvers or complexity of the approval process increases, the client code would become more complex and harder to manage.

 By applying the **Chain of Responsibility** pattern, these drawbacks are mitigated. It provides a more modular and flexible approach to handling purchase requests. The client code is decoupled from the specific details of the approval process, making it easier to extend, modify, and maintain the system. It enables dynamic chaining of handlers, where each handler focuses on its specific responsibility without the need for the client to know the specifics of each handler. The client only needs to interact with the first handler in the chain, and the rest of the processing is delegated through the chain.

 Okay, its time to checkout **Chain of Responsibility (CoR)** in action. Study the code snippet below.

```
/**  
 * The Approver abstract class represents an approver in a purchase approval chain.  
 * It provides a method to set the next approver and process purchase requests.  
 */  
public abstract class Approver {  
    protected Approver nextApprover;  
    protected double approvalLimit;  
  
    /**  
     * Sets the next approver in the chain.  
     *  
     * @param nextApprover The next approver to be set.  
     */  
    public void setNextApprover(Approver nextApprover) {  
        this.nextApprover = nextApprover;  
    }  
    /**  
     * Processes a purchase request.  
     * If the request amount is within the approval limit, the current approver approves the purchase.  
     * If the request amount exceeds the approval limit and a next approver is available, the request is passed to the next approver.  
     * If no approver can handle the request, a message is displayed.  
     *  
     * @param request The purchase request to be processed.  
     */  
    public void processRequest(PurchaseRequest request) {
```

```
if (request.getAmount() <= approvalLimit) {
    System.out.println(this.getClass().getSimpleName() + " approved the purchase.");
} else if (nextApprover != null) {
    nextApprover.processRequest(request);
} else {
    System.out.println("None of the approvers can handle the purchase request.");
}
}

/**
 * The Manager class represents a manager approver.
 * It has an approval limit of 1000.0.
 */
public class Manager extends Approver {
    public Manager() {
        this.approvalLimit = 1000.0;
    }
}

/**
 * The Director class represents a director approver.
 * It has an approval limit of 5000.0.
 */
public class Director extends Approver {
    public Director() {
        this.approvalLimit = 5000.0;
    }
}

/**
 * The CEO class represents a CEO approver.
 * It has an approval limit of 10000.0.
 */
public class CEO extends Approver {
    public CEO() {
        this.approvalLimit = 10000.0;
    }
}

/**
 * The PurchaseRequest class represents a purchase request.
 * It holds the amount and description of the request.
 */
public class PurchaseRequest {
    private double amount;
    private String description;
    /**
     * Constructs a new PurchaseRequest with the specified amount and description.
     *
     * @param amount The amount of the purchase request.
     * @param description The description of the purchase request.
     */
}
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
/*
public PurchaseRequest(double amount, String description) {
    this.amount = amount;
    this.description = description;
}
/** 
 * Returns the amount of the purchase request.
 *
 * @return The amount of the purchase request.
 */
public double getAmount() {
    return amount;
}
/** 
 * Sets the amount of the purchase request.
 *
 * @param amount The amount to be set.
 */
public void setAmount(double amount) {
    this.amount = amount;
}
/** 
 * Returns the description of the purchase request.
 *
 * @return The description of the purchase request.
 */
public String getDescription() {
    return description;
}
/** 
 * Sets the description of the purchase request.
 *
 * @param description The description to be set.
 */
public void setDescription(String description) {
    this.description = description;
}
*/
/** 
 * The Client class demonstrates the usage of the approval chain.
 * It creates instances of different approvers, sets up the chain, and processes purchase requests.
 */
public class Client {
    public static void main(String[] args) {
        Approver manager = new Manager();
        Approver director = new Director();
        Approver ceo = new CEO();
        manager.setNextApprover(director);
    }
}
```

```
director.setNextApprover(ceo);
PurchaseRequest request1 = new PurchaseRequest(500.0, "Office supplies");
manager.processRequest(request1);
PurchaseRequest request2 = new PurchaseRequest(7000.0, "New equipment");
manager.processRequest(request2);
PurchaseRequest request3 = new PurchaseRequest(15000.0, "Large-scale project");
manager.processRequest(request3);
}
}
```

 In the above code snippet, the different types of Approvers (Manager, Director, CEO) form a chain where each approver has a specific approval limit. The *PurchaseRequest* object is processed through the chain by invoking the *processRequest()* method on the first approver (Manager). If the request amount exceeds the approval limit of the current approver, the request is passed to the next approver until it is approved or no more approvers are available.



Let us discuss some of the key benefits of using Chain of Responsibility pattern.

- Chain of Responsibility pattern promotes loose coupling between the sender and receiver of a request. The sender doesn't need to know the specific handler that will process the request, and the handler doesn't need to know the sender.
- The chain of handlers can be dynamically modified at runtime. Handlers can be added or removed without affecting the rest of the chain, providing dynamic flexibility in handling requests.
- It allows for multiple handlers to be involved in processing a request.
- Control is decentralized as the decision of which handler processes the request is determined at runtime based on the chain's configuration. This allows for more dynamic and adaptable request handling.
- This pattern supports easy extension by adding new handlers to the chain. New handlers can be introduced without modifying the existing code, promoting the **Open-Closed Principle**.



The **Chain of Responsibility** pattern is often combined with the **Composite pattern** to form a powerful and flexible design solution. In this combination, the parent component in the Composite pattern can act as the successor in the Chain of Responsibility.

The Composite pattern allows to compose objects into tree-like structures to represent part-whole hierarchies. Each component in the composite structure can have child components, forming a recursive structure.

If a component at a lower level in the composite structure cannot handle a request, it passes the request to its parent component, which acts as its successor in the chain. The parent component then can handle the request or pass it along to its own parent, continuing up the hierarchy until a suitable handler is found.

This combination allows for a flexible and dynamic handling of requests in complex composite structures. Each component can choose to handle the request or delegate it to its parent, enabling different levels of responsibility and behavior in the composite hierarchy.

6.2. Command Pattern

 **Command Pattern** turns a request into a standalone object, which can be parameterized, queued, and executed at different points in time.

 But why would we turn a request into an object in the first place? Request is issued as a command to perform some operation on objects, but here we are talking about turning a request into an object itself. What would be the benefit from this approach and why would we even think of doing this?

Let us consider a scenario. Say we are working on developing GUI (Graphical User Interface) for a document editor software. We have requirement of different buttons for different document related tasks such as opening the file, saving the file to disk, closing the file, etc. We go ahead and create a base Button class and then we create multiple subclasses for different operations, such as OpenButton, CloseButton, and so on. These subclasses would contain the code that would be executed on a button click.

 It would not be long before we realize the flaws of this approach. As the application grows and GUI expands, we would need different buttons for different tasks and for each specific task we would be creating specific button subclass. Also, there would be various document types and there would be tasks specific to a document type, and we will have to support those tasks on button click by creating more button subclasses.

 To avoid this mess of subclasses, we leverage **Command Pattern**, which enables the GUI interface to handle multiple actions independently, allowing each button to have its own **command object** and associated behavior.

The Command Pattern involves following components:

1. **Command:** Defines the interface for executing a specific action or operation. It typically includes an execute() method.
2. **Concrete Command:** Implements the Command interface and represents a specific command or action. It encapsulates the receiver object and invokes the corresponding operation.
3. **Receiver:** Defines the interface or implementation of the object that performs the actual work when a command is executed.
4. **Invoker:** Requests the execution of a command and maintains a reference to the command object. It does not know the specifics of the command, only that it needs to execute it.
5. **Client:** Creates and configures the command objects and assigns them to specific receivers. It initiates the requests by invoking the execute() method on the command.

 Let us look at the following code implementation to understand Command Pattern.

```
/**  
 * The Command interface represents a command that can be executed.  
 */  
public interface Command {  
    /**  
     * Executes the command.  
     */
```

```
void execute();
}

/**
 * The SaveCommand class represents a concrete command to save a document.
 */
public class SaveCommand implements Command {
    private Document document;
    /**
     * Constructs a SaveCommand object with the specified document.
     *
     * @param document the document to be saved
     */
    public SaveCommand(Document document) {
        this.document = document;
    }
    /**
     * Executes the save command by calling the document's save method.
     */
    public void execute() {
        document.save();
    }
}
/**
 * The OpenCommand class represents a concrete command to open a document.
 */
public class OpenCommand implements Command {
    private Document document;
    /**
     * Constructs an OpenCommand object with the specified document.
     *
     * @param document the document to be opened
     */
    public OpenCommand(Document document) {
        this.document = document;
    }
    /**
     * Executes the open command by calling the document's open method.
     */
    public void execute() {
        document.open();
    }
}
/**
 * The Document class represents the receiver that can save and open documents.
 */
public class Document {
    /**
     * Saves the document.

```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
/*
public void save() {
    System.out.println("Document saved.");
}

/**
 * Opens the document.
 */

public void open() {
    System.out.println("Document opened.");
}

*/
/** 
 * The Button class represents the invoker that triggers the execution of a command.
 */

public class Button {
    private Command command;

    /**
     * Sets the command to be executed when the button is clicked.
     *
     * @param command the command to be set
     */
    public void setCommand(Command command) {
        this.command = command;
    }

    /**
     * Simulates a button click by executing the assigned command.
     */
    public void click() {
        command.execute();
    }
}

*/
/** 
 * The Client class demonstrates the usage of commands, buttons, and documents.
 */

public class Client {
    public static void main(String[] args) {
        Document document = new Document();
        // Creating commands
        Command saveCommand = new SaveCommand(document);
        Command openCommand = new OpenCommand(document);
        // Creating buttons and associating commands
        Button saveButton = new Button();
        saveButton.setCommand(saveCommand);
        Button openButton = new Button();
        openButton.setCommand(openCommand);
        // Simulating button clicks
        saveButton.click(); // Save button clicked
        openButton.click(); // Open button clicked
    }
}
```

```
    }  
}
```

 In this code example, we have two different buttons, each associated with its own command. *SaveCommand* represents the action of saving a document, and *OpenCommand* represents the action of opening a document. When the buttons are clicked, the respective commands are executed, and the associated actions are performed by the Document receiver.

 Command Pattern promotes modularity, flexibility, and reusability. It becomes easy to add new buttons with different commands without modifying the existing code (**Open Closed Principle**). Additionally, the commands can be parameterized, allowing for more flexibility in configuring and customizing the actions performed by each button. Since commands are objects, they can support additional features like undo/redo functionality or logging of executed commands. This allows for easy reversal of actions and auditing of operations.

6.3. Iterator Pattern

 **Iterator Pattern** provides a way to access the elements of an aggregate object sequentially without exposing its underlying structure.

You receive a box of chocolates; you decide to count how many chocolates are there in the box. You don't need to open the wrapper of chocolate or look inside what chocolate is made of. You just count. Box represents a collection and chocolate represents the objects of collection. Since, you are counting here, so you are the iterator.

Okay, let us take an example in context of software programming.

Imagine, we are working on a music player application. We want to create an iterator for a music playlist that allows us to iterate over the songs in the playlist and perform various operations.

If we approach this problem in traditional way of programming, we will engage client code to directly access the internal representation of playlist, which could be list of songs in string data type. Client Implementation code might look something like below.

```
MusicPlaylist musicPlaylist = new MusicPlaylist();  
musicPlaylist.addSong("Bohemian Rhapsody");  
musicPlaylist.addSong("Hotel California");  
musicPlaylist.addSong("Imagine");  
List<String> songs = musicPlaylist.getSongs();  
for (String song : songs) {  
    System.out.println(song);  
}
```

 It leads to tight coupling between client code and internals of playlist. Say, at some point of time, we decide to use another data structure inside playlist, we would have to modify the client code as well. Basically, it violates the design principles and leads to potential issues and increased coupling.



Iterator Pattern can be used to avoid this scenario. It can separate the traversal logic from the playlist object, without exposing the underlying structure or implementation details.

Iterator Pattern incorporates following components:

1. **Iterator:** Defines the interface for accessing and traversing elements in a collection.
2. **Concrete Iterator:** Implements the Iterator interface and provides the implementation for traversing the collection.
3. **Aggregate:** Defines the interface for creating an Iterator object.
4. **Concrete Aggregate:** Implements the Aggregate interface and provides the implementation for creating an Iterator object.
5. **Client:** Utilizes the Iterator to traverse and access the elements of a collection.



You must have guessed by now, its time to play with code snippets adhering to Iterator Pattern.

```
import java.util.ArrayList;
import java.util.List;
import java.util.NoSuchElementException;

/**
 * The Iterator interface represents the contract for iterating over a collection of elements.
 *
 * @param <T> the type of elements in the iterator
 */
public interface Iterator<T> {
    /**
     * Checks if there are more elements in the iterator.
     *
     * @return true if there are more elements, false otherwise
     */
    boolean hasNext();

    /**
     * Retrieves the next element in the iterator.
     *
     * @return the next element
     * @throws NoSuchElementException if there are no more elements
     */
    T next();
}

/**
 * The PlaylistIterator class is a concrete implementation of the Iterator interface
 * specifically designed for iterating over a playlist of songs.
 */
public class PlaylistIterator implements Iterator<Song> {
    private List<Song> playlist;
    private int position;

    /**
     * Constructs a PlaylistIterator object with the given playlist.
     */
}
```

```
* @param playlist the playlist to iterate over
*/
public PlaylistIterator(List<Song> playlist) {
    this.playlist = playlist;
    this.position = 0;
}

/**
 * Checks if there are more songs in the playlist.
 *
 * @return true if there are more songs, false otherwise
 */
public boolean hasNext() {
    return position < playlist.size();
}

/**
 * Retrieves the next song in the playlist.
 *
 * @return the next song
 * @throws NoSuchElementException if there are no more songs
 */
public Song next() {
    if (!hasNext()) {
        throw new NoSuchElementException();
    }
    return playlist.get(position++);
}

/**
 * The Playlist interface represents the contract for creating an iterator over a playlist.
 */
public interface Playlist {
    /**
     * Creates an iterator for iterating over the playlist.
     *
     * @return an iterator for the playlist
     */
    Iterator<Song> createIterator();
}

/**
 * The MusicPlaylist class is a concrete implementation of the Playlist interface
 * that represents a music playlist.
 */
public class MusicPlaylist implements Playlist {
    private List<Song> playlist;
    /**
     * Constructs an empty MusicPlaylist object.
     */
    public MusicPlaylist() {
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
    this.playlist = new ArrayList<>();
}
/***
 * Adds a song to the playlist.
 *
 * @param song the song to add
 */
public void addSong(Song song) {
    playlist.add(song);
}
/***
 * Creates an iterator for iterating over the playlist.
 *
 * @return an iterator for the playlist
 */
public Iterator<Song> createIterator() {
    return new PlaylistIterator(playlist);
}
*/
/***
 * The Song class represents a song with a title and an artist.
 */
public class Song {
    private String title;
    private String artist;
    /**
     * Constructs a Song object with the given title and artist.
     *
     * @param title the title of the song
     * @param artist the artist of the song
     */
    public Song(String title, String artist) {
        this.title = title;
        this.artist = artist;
    }
    /**
     * Retrieves the title of the song.
     *
     * @return the title of the song
     */
    public String getTitle() {
        return title;
    }
    /**
     * Retrieves the artist of the song.
     *
     * @return the artist of the song
     */
}
```

```

public String getArtist() {
    return artist;
}

}

/**
 * The Client class is responsible for demonstrating the usage of the MusicPlaylist and Iterator classes.
 */

public class Client {
    public static void main(String[] args) {
        MusicPlaylist musicPlaylist = new MusicPlaylist();
        musicPlaylist.addSong(new Song("Bohemian Rhapsody", "Queen"));
        musicPlaylist.addSong(new Song("Hotel California", "Eagles"));
        musicPlaylist.addSong(new Song("Imagine", "John Lennon"));
        Iterator<Song> iterator = musicPlaylist.createIterator();
        while (iterator.hasNext()) {
            Song song = iterator.next();
            System.out.println("Title: " + song.getTitle() + ", Artist: " + song.getArtist());
        }
    }
}

```



In this example, we have a *MusicPlaylist* class that implements the *Playlist* interface. The *Playlist* interface declares the *createIterator()* method, which creates an iterator for the playlist. The *PlaylistIterator* class implements the Iterator interface and provides the implementation for iterating over the list of songs in the playlist. The client code creates an instance of *MusicPlaylist*, adds songs to the playlist, and obtains an iterator using the *createIterator()* method. The client then uses the iterator to iterate over the playlist, accessing each song's title and artist.

We now have a standardized way to access and iterate over the playlist, making the client code more flexible and reusable. If we decide to change the internal representation of playlist, it won't affect the client code.



The **Iterator pattern** can be beneficial in real-life scenarios such as music or video streaming services, where playlists or collections of media items need to be traversed and accessed in a consistent manner. It enables clients to iterate over the playlist and perform operations on each item without having to know the internal structure or implementation details of the playlist.

By separating the traversal logic from the collection, it allows for easy extension and customization of traversal algorithms. New iterator implementations can be added without modifying the collection or the client code, providing flexibility in choosing different iteration strategies.

The Iterator pattern simplifies client code by providing a high-level interface for iterating over elements. When iterating over a collection requires complex logic or external factors, such as synchronization, filtering, or sorting, the Iterator pattern can help encapsulate and manage the iteration process more efficiently.

6.4. Mediator Pattern



Mediator Pattern encapsulates the interaction between set of objects within a mediator object. This mediator object acts as a facilitator of communication between multiple objects without them having direct reference to each other.

Mediator? Oh! I know what it means. When facing disagreements with my best friend, my girlfriend often stepped in as a mediator. Rather than directly communicating with each other, we relied on my girlfriend as the intermediary. She facilitated our conversations, enabling us to address our differences, resolve conflicts, and even assist in accomplishing tasks by relaying messages between us. Her role as a mediator helped maintain a harmonious relationship and foster effective communication between us, promoting understanding and cooperation.

Design Patterns are not something new. They have not been invented. They have been inspired by our behavior in real world, about how efficiently we try to solve problems, given difficult situations.

Imagine having a debate organized without a facilitator (mediator). What a chaos it would be, although entertaining to the viewers!

Now, let us take an example in context of Software development.

Say, we are developing an air traffic control system, which is responsible for ensuring safe and efficient air travel by managing the communication and coordination between flights. The system needs to facilitate the exchange of information between flights without tightly coupling them together. It should handle the registration of flights, their take-off, and provide a centralized hub for broadcasting location updates to all other flights.



Why it is mentioned to avoid direct coupling of flights?

Allowing them to communicate without a central coordination mechanism can lead to issues. It can result in high dependencies between flights, making the system difficult to extend, modify, and maintain. Additionally, without a centralized communication mechanism, there is a lack of control and consistency in the information shared between flights. And in air traffic control system, lapse in information/communication can turn into a disaster any given second.



To address these challenges, the Mediator pattern can be applied. The Mediator pattern allows for the decoupling of flights by introducing an Air Traffic Control Tower as the central mediator. The tower serves as a hub for communication and coordination, ensuring that flights interact through the mediator rather than directly with each other.

Following are the main components of Mediator Pattern:

1. **Mediator:** Defines an interface for communication between objects.
2. **Concrete Mediator:** Implements the Mediator interface and manages the communication and interaction between objects.
3. **Colleague:** Defines an interface for objects that communicate with each other through the Mediator.
4. **Concrete Colleague:** Implements the Colleague interface and communicates with other objects through the Mediator.



Let us meet our sample Mediator now!

```
import java.util.ArrayList;
import java.util.List;

/**
 * The AirTrafficControl interface defines the methods that a mediator (ATC) must implement.
 */
public interface AirTrafficControl {
    /**
     * Registers a flight with the air traffic control.
     *
     * @param flight The flight to be registered.
     */
    void registerFlight(Flight flight);

    /**
     * Sends the location of a flight to other registered flights.
     *
     * @param location The location of the flight.
     * @param origin The flight sending the location.
     */
    void sendLocation(String location, Flight origin);
}

/**
 * The AirTrafficControlTower class is a concrete implementation of the AirTrafficControl interface.
 * It acts as a mediator between flights, allowing them to communicate with each other.
 */
public class AirTrafficControlTower implements AirTrafficControl {
    private List<Flight> flights;

    /**
     * Constructs a new AirTrafficControlTower object.
     * Initializes the list of flights.
     */
    public AirTrafficControlTower() {
        this.flights = new ArrayList<>();
    }

    /**
     * Registers a flight with the air traffic control tower.
     *
     * @param flight The flight to be registered.
     */
    public void registerFlight(Flight flight) {
        flights.add(flight);
    }

    /**
     * Sends the location of a flight to other registered flights.
     *
     *
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
* @param location The location of the flight.
* @param origin The flight sending the location.
*/
public void sendLocation(String location, Flight origin) {
    for (Flight flight : flights) {
        if (flight != origin) {
            flight.receiveLocation(location);
        }
    }
}
/** 
 * The Flight interface defines the methods that a flight must implement.
 */
public interface Flight {
    /**
     * Performs the take-off operation for the flight.
     */
    void takeOff();
    /**
     * Receives the location information from the air traffic control.
     *
     * @param location The location of the flight.
     */
    void receiveLocation(String location);
}
/** 
 * The FlightImpl class is a concrete implementation of the Flight interface.
 * It represents a flight and interacts with the air traffic control.
 */
public class FlightImpl implements Flight {
    private String flightNumber;
    private AirTrafficControl airTrafficControl;
    /**
     * Constructs a new FlightImpl object with the given flight number and air traffic control.
     *
     * @param flightNumber The flight number.
     * @param airTrafficControl The air traffic control mediator.
     */
    public FlightImpl(String flightNumber, AirTrafficControl airTrafficControl) {
        this.flightNumber = flightNumber;
        this.airTrafficControl = airTrafficControl;
    }
    /**
     * Performs the take-off operation for the flight.
     */
    public void takeOff() {
        System.out.println("Flight " + flightNumber + " is taking off.");
    }
}
```

```
airTrafficControl.sendLocation("In Air", this);
}

/**
 * Receives the location information from the air traffic control.
 *
 * @param location The location of the flight.
 */
public void receiveLocation(String location) {
    System.out.println("Flight " + flightNumber + " received location: " + location);
}

/**
 * The Client class demonstrates the usage of the mediator pattern.
 * It creates flights and an air traffic control tower and registers the flights with the tower.
 * It then initiates the take-off operation for one of the flights.
 */
public class Client {
    public static void main(String[] args) {
        AirTrafficControl airTrafficControl = new AirTrafficControlTower();
        Flight flight1 = new FlightImpl("ABC123", airTrafficControl);
        Flight flight2 = new FlightImpl("XYZ789", airTrafficControl);
        airTrafficControl.registerFlight(flight1);
        airTrafficControl.registerFlight(flight2);
        flight1.takeOff();
    }
}
*****OUTPUT*****
// Flight ABC123 is taking off.

//Flight XYZ789 received location: In Air
```

 In this example, we have an *AirTrafficControl* interface that defines the methods for communication between flights and the air traffic control tower. *AirTrafficControlTower* class implements *AirTrafficControl* interface and manages the communication and coordination between flights.

Flight interface represents the flights that interact with the air traffic control. *FlightImpl* class implements *Flight* interface and communicates with the air traffic control through *AirTrafficControl* mediator.

The client code creates an instance of *AirTrafficControlTower* and *Flight* objects. The flights are registered with the air traffic control tower, and when a flight takes off, it sends its location to the air traffic control tower, which then broadcasts the location to other flights.

 By using **Mediator pattern**, the flights do not need to know about each other or the air traffic control tower directly. They communicate through the mediator, which manages the interaction and broadcasting of locations. It facilitates loose coupling and simplifies the coordination and communication between flights and the air traffic control tower.

The Mediator pattern is particularly useful in complex systems where multiple objects need to communicate and coordinate with each other. Examples include chat applications, multi-player games, or distributed systems, where the objects interact with each other in various ways.

6.5. Memento Pattern



Memento Pattern helps us to capture and restore internal state of an object without violating encapsulation.

Memento. Yes! Yes! A Christopher Nolan movie. A guy, whose wife is murdered, is seeking out revenge on the villains. But he suffers from the short-term memory loss and inability to form new memories. He goes on his quest using an elaborate system of Polaroid photographs, handwritten notes, and tattoos to track information he won't remember.

Oh! What an outstanding movie! Serves the template right to explain and implement Memento Pattern. Only, you don't have to get tattoos and polaroid.

Say, we are developing a text editor application that provides the capability to undo and redo text modifications. The text editor should allow users to perform various operations such as appending text, deleting text, and modifying existing text. Users should be able to revert changes and restore the previous state of the text through the undo/redo functionality.



If we try to develop this application in a naïve manner, then the text editor would need to handle the management of different states itself. This would involve manually saving the state before each modification and implementing a mechanism to track and restore previous states. This could quickly become complex and error-prone, especially when dealing with multiple modifications and the need to handle the correct sequence of undo/redo operations.



Memento Pattern leverages an external object which encapsulates state management process.

Following are the components of Memento Pattern:

1. **Originator:** The object whose state needs to be saved or restored. It creates a memento object containing the current state and can restore its state from a memento.
2. **Memento:** Represents the saved state of the originator. It provides methods to access the state but does not expose any internal details.
3. **Caretaker:** Manages and keeps track of the mementos. It requests mementos from the originator and can restore the originator's state using a memento.



Time to get those tattoos, alright! Aah! A simple data structure would suffice. Just checkout the code snippet and have fun!

```
import java.util.ArrayList;
import java.util.List;

/**
 * The TextEditor class represents a simple text editor that allows users to
```

```
* append text, undo and redo changes.
* It uses the Memento design pattern to save and restore the state of the text.
*/
public class TextEditor {
    private StringBuilder text;
    private History history;
    private int currentState;

    /**
     * Constructs a new TextEditor object with an empty text and a new History
     * object.
     * The current state is set to -1.
     */
    public TextEditor() {
        this.text = new StringBuilder();
        this.history = new History();
        this.currentState = -1;
    }

    /**
     * Appends the given content to the text.
     * Clears the redo history and saves the state.
     *
     * @param content the text to append
     */
    public void appendText(String content) {
        clearRedoHistory();
        text.append(content);
        saveState();
    }

    /**
     * Undoes the last change by restoring the previous state of the text.
     * If there is no previous state, nothing happens.
     */
    public void undo() {
        if (currentState > 0) {
            currentState--;
            restoreState();
        }
    }

    /**
     * Redoes the last change by restoring the next state of the text.
     * If there is no next state, nothing happens.
     */
    public void redo() {
        if (currentState < history.getSize() - 1) {
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
        currentState++;
        restoreState();
    }

}

/***
 * Returns the current text as a string.
 *
 * @return the current text
 */
public String getText() {
    return text.toString();
}

/***
 * Saves the current state of the text by creating a new TextEditorMemento
 * object
 * and adding it to the history.
 * Updates the current state.
 */
private void saveState() {
    history.push(new TextEditorMemento(text.toString()));
    currentState++;
}

/***
 * Restores the state of the text to a previous state using the
 * TextEditorMemento object
 * at the current state index in the history.
 */
private void restoreState() {
    TextEditorMemento memento = history.get(currentState);
    text = new StringBuilder(memento.getState());
}

/***
 * Clears the redo history by removing all TextEditorMemento objects after the
 * current state index.
 */
private void clearRedoHistory() {
    history.clearRedoHistory(currentState + 1);
}

}

/***
 * The TextEditorMemento class represents a memento object that stores the state
 * of the text.
 */

```

```
public class TextEditorMemento {  
    private final String state;  
  
    /**  
     * Constructs a new TextEditorMemento object with the given state.  
     *  
     * @param state the state of the text  
     */  
    public TextEditorMemento(String state) {  
        this.state = state;  
    }  
  
    /**  
     * Returns the state of the text.  
     *  
     * @return the state of the text  
     */  
    public String getState() {  
        return state;  
    }  
}  
  
/**  
 * The History class represents a caretaker that manages the mementos of the  
 * text.  
 */  
public class History {  
    private List<TextEditorMemento> mementos;  
  
    /**  
     * Constructs a new History object with an empty list of mementos.  
     */  
    public History() {  
        this.mementos = new ArrayList<>();  
    }  
  
    /**  
     * Adds the given memento to the list of mementos.  
     *  
     * @param memento the memento to add  
     */  
    public void push(TextEditorMemento memento) {  
        mementos.add(memento);  
    }  
  
    /**  
     * Returns the memento at the specified index in the list.  
     */
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
* @param index the index of the memento
* @return the memento at the specified index
*/
public TextEditorMemento get(int index) {
    return mementos.get(index);
}

/**
 * Returns the number of mementos in the list.
 *
 * @return the number of mementos
*/
public int getSize() {
    return mementos.size();
}

/**
 * Clears the redo history by removing all mementos after the specified index in
 * the list.
 *
 * @param index the index after which to clear the redo history
*/
public void clearRedoHistory(int index) {
    mementos.subList(index, mementos.size()).clear();
}

/**
 * The Client class is a test class that demonstrates the usage of the
 * TextEditor class.
*/
public class Client {
    public static void main(String[] args) {
        TextEditor textEditor = new TextEditor();
        // Append some text
        textEditor.appendText("Hello, ");
        textEditor.appendText("World!");
        // Print current text
        System.out.println("Current Text: " + textEditor.getText());
        // Undo last change
        textEditor.undo();
        // Print text after undo
        System.out.println("After Undo: " + textEditor.getText());
        // Redo last change
        textEditor.redo();
        // Print text after redo
        System.out.println("After Redo: " + textEditor.getText());
    }
}
```

```
}

*****OUTPUT*****  
Current Text: Hello, World!  
After Undo: Hello,  
After Redo: Hello, World!
```

 In this code example, *TextEditor* class includes *redo()* method along with *undo()* method. The *redo()* method restores the next state from the history if available. *TextEditor* class is the **originator**, responsible for managing and manipulating the text content. It has methods to append text, retrieve the current text, save the state by creating a *TextEditorMemento* object, and restore the state from a *TextEditorMemento* object.

The *TextEditorMemento* class represents the **memento**, encapsulating the state of the *TextEditor*. It has a *getState()* method to access the saved state.

History class, acts as the **caretaker**, includes list of *TextEditorMemento* and *clearRedoHistory()* method, which clears the redo history from the specified index onwards. This ensures that redo operations are properly handled.

The client code demonstrates the use of undo and redo functionality. After appending some text, the text editor performs an undo operation, restores the previous state, and prints the text. Then, a redo operation is performed, restoring the subsequent state, and printing the text after redo.

 The **Memento pattern** allows the text editor to save and restore multiple states, enabling users to undo and redo changes in their text content. It encapsulates the object's state within a memento object, preserving encapsulation and information hiding. The **originator** maintains control over its state, and the **memento** provides a way to access and restore that state without exposing internal details. It enables the tracking and management of object state history. **Caretakers** can store multiple mementos, representing different snapshots of the originator's state, and restore any of them when needed.

By moving the responsibility of state management to mementos and caretakers, originator becomes simpler and focused on its core functionality. It doesn't need to handle state saving and restoration, resulting in more maintainable and cohesive code.

6.6. Observer Pattern

 The **Observer pattern**, also known as *Publish-Subscribe pattern* or *Event-Listener pattern*, establishes a one-to-many relationship between objects, where changes in one object (subject) trigger updates in multiple other objects (observers).

Imagine you have been hired by BreakFirst news agency, to develop a system which delivers news stories to its customers on their emails, as soon as event happens or is orchestrated. Who knows! Your mission, should you choose to accept it, is to provide real-time updates to customers and with the ability to add or remove customers.

Let us break down this scenario in terms of Software programming.

The news agency is a central hub that generates and distributes news articles to a diverse set of subscribers. The agency needs a system to efficiently manage and notify subscribers about new articles while ensuring loose coupling between the news agency and the subscribers. The goal is to provide real-

time updates to subscribers without direct dependencies and with the ability to dynamically add or remove subscribers.



System cannot afford directly coupling the news agency with individual subscribers, as it can lead to issues. It would require the news agency to have direct knowledge of each subscriber's implementation details, resulting in tight coupling and reduced flexibility. Additionally, managing subscriptions and notifications for a growing number of subscribers becomes complex and error-prone.



The **Observer Pattern** can be used to ensure real-time communication and manage a dynamic set of subscribers.

The Observer Pattern has following components:

1. **Subject:** Represents the object that is being observed. It maintains a list of observers and provides methods to register, remove, and notify observers of any changes.
2. **Observer:** Defines an interface or an abstract class that observers implement. It specifies the update method that is called by the subject to notify changes.
3. **Concrete Subject:** Extends the subject and implements its methods. It sends notifications to observers when its state changes.
4. **Concrete Observer:** Implements the observer interface and defines the specific actions to be taken when notified by the subject.



Let us look at the code snippet which samples implementation of Observer Pattern for news agency use case.

```
// Subject

import java.util.ArrayList;
import java.util.List;

/**
 * Represents a subject that can register, remove, and notify subscribers about
 * news updates.
 */
public interface NewsSubject {
    /**
     * Registers a new subscriber to receive news updates.
     *
     * @param subscriber the subscriber to be registered
     */
    void registerSubscriber(NewsSubscriber subscriber);

    /**
     * Removes a subscriber from receiving news updates.
     *
     * @param subscriber the subscriber to be removed
     */
    void removeSubscriber(NewsSubscriber subscriber);
```

```
/*
 * Notifies all subscribers about a news update.
 *
 * @param news the news update to be sent to subscribers
 */
void notifySubscribers(String news);

}

// Concrete Subject
/**
 * Represents a news agency that implements the NewsSubject interface.
 */
public class NewsAgency implements NewsSubject {
    private List<NewsSubscriber> subscribers;

    /**
     * Constructs a new NewsAgency object with an empty list of subscribers.
     */
    public NewsAgency() {
        this.subscribers = new ArrayList<>();
    }

    /**
     * Registers a new subscriber to receive news updates.
     *
     * @param subscriber the subscriber to be registered
     */
    public void registerSubscriber(NewsSubscriber subscriber) {
        subscribers.add(subscriber);
    }

    /**
     * Removes a subscriber from receiving news updates.
     *
     * @param subscriber the subscriber to be removed
     */
    public void removeSubscriber(NewsSubscriber subscriber) {
        subscribers.remove(subscriber);
    }

    /**
     * Notifies all subscribers about a news update.
     *
     * @param news the news update to be sent to subscribers
     */
    public void notifySubscribers(String news) {
        for (NewsSubscriber subscriber : subscribers) {
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
        subscriber.update(news);
    }
}
}

// Observer
/**
 * Represents an observer that receives news updates.
 */
public interface NewsSubscriber {
    /**
     * Updates the subscriber with a news update.
     *
     * @param news the news update received by the subscriber
     */
    void update(String news);
}

// Concrete Observer
/**
 * Represents an email subscriber that implements the NewsSubscriber interface.
 */
public class EmailSubscriber implements NewsSubscriber {
    private String email;

    /**
     * Constructs a new EmailSubscriber object with the specified email address.
     *
     * @param email the email address of the subscriber
     */
    public EmailSubscriber(String email) {
        this.email = email;
    }

    /**
     * Updates the email subscriber with a news update.
     *
     * @param news the news update received by the subscriber
     */
    public void update(String news) {
        System.out.println("Email sent to " + email + ": " + news);
    }
}

// Client code
/**
 * Represents a client that demonstrates the usage of the NewsAgency,
 * NewsSubscriber, and EmailSubscriber classes.

```

```
/*
public class Client {
    public static void main(String[] args) {
        NewsAgency newsAgency = new NewsAgency();
        // Create subscribers
        NewsSubscriber subscriber1 = new EmailSubscriber("subscriber1@example.com");
        NewsSubscriber subscriber2 = new EmailSubscriber("subscriber2@example.com");
        // Register subscribers
        newsAgency.registerSubscriber(subscriber1);
        newsAgency.registerSubscriber(subscriber2);
        // Send news updates
        newsAgency.notifySubscribers("Breaking News: New article published!");
        // Unregister subscriber
        newsAgency.removeSubscriber(subscriber2);
        // Send news update after removal
        newsAgency.notifySubscribers("Latest News: Another article published!");
    }
}
*****OUTPUT*****
```

Email sent to subscriber1@example.com: Breaking News: New article published!

Email sent to subscriber2@example.com: Breaking News: New article published!

Email sent to subscriber1@example.com: Latest News: Another article published!

 In this code example, *NewsAgency* acts as the **subject** and maintains a list of subscribers (*NewsSubscriber*). It provides methods to register, remove, and notify subscribers about new news updates.

EmailSubscriber class is a **concrete observer** that implements *NewsSubscriber* interface. It defines *update()* method, which is called by the subject to send news updates to the subscriber via email.

The client code creates a *NewsAgency* instance and registers *EmailSubscriber* instances as subscribers. When the news agency sends news updates by calling *notifySubscribers()*, the subscribers are notified and receive the updates.

 **Observer pattern** promotes loose coupling between the news agency and subscribers, allowing for flexible and scalable news delivery. It enables real-time communication and synchronization between the news agency and its subscribers without requiring direct dependencies between them. It allows to manage a dynamic set of subscribers and notify them about new news updates.

This pattern is widely used in event-driven systems where objects need to react to events or changes in the system. For example, graphical user interfaces (GUIs) often use the Observer pattern to handle user interactions. An application for stock market updates can leverage Observer Pattern, where multiple entities need to be notified about changes in a stock market or financial data.

With the Observer pattern, the subjects and observers are reusable and extensible. New observers can be easily added without modifying the subject or other observers. Similarly, new subjects can be introduced without affecting existing observers.



Observers are dependent on the subject, and not vice versa. This simplifies the management of dependencies, as the subject does not need to be aware of specific observer implementations.

6.7. State Pattern



State pattern allows an object to alter its behavior when its internal state changes. The object appears to be changing its class based on its current state.

Imagine a person who experiences different emotional states such as happiness, sadness, anger, and calmness. Each emotional state represents a specific behavior and response to external stimuli. The person's behavior and reactions vary depending on their current emotional state. When the person is in a state of happiness, they exhibit behaviors such as smiling, laughing, and being more sociable. They respond positively to events and perceive the world in an optimistic way. In a state of sadness, the same person may exhibit behaviors like crying, withdrawing from social interactions, and feeling low. They respond differently to events, expressing sadness and feeling less motivated. When in a state of anger, same person might exhibit behaviors like raised voice, aggressive body language, and confrontational attitude. They respond to events with frustration and may find it challenging to control their emotions.

The person at the core remains same, however, when internal state changes, different behavior is exhibited. Same concept is applied via State Pattern whilst dealing with behavior of objects in certain scenarios.

Consider designing a Vending Machine - Product Dispensing System. A vending machine is a self-service system that allows customers to purchase products by inserting coins and selecting desired items. The vending machine needs to be designed to manage the dispensing of products while handling various scenarios, such as insufficient funds, sold-out products, and successful transactions. Our goal is to provide an efficient and user-friendly system that accurately tracks inventory, accepts payments, and dispenses products based on user requests.



However, designing a vending machine without proper organization and state management can lead to various issues. For instance, it may be challenging to handle different states of the machine, such as no coins inserted, product selection, or dispensing. Additionally, managing inventory, tracking transactions, and handling edge cases can become complex without a structured approach. From the knowledge we have gathered so far, we understand that maintaining or managing state within Vending Machine object is a very bad idea. So, we will come up with a better approach leveraging abstraction.



State pattern encapsulates different behaviors in separate state classes, which can be dynamically switched at runtime.

Following are the participants of **State Pattern**:

1. **Context:** Represents the object whose behavior changes based on its internal state. It maintains a reference to the current state object and delegates behavior-related requests to that state object.
2. **State:** Defines an interface or an abstract class that encapsulates the behavior associated with a particular state. It declares methods that handle the requests and may also define methods to transition to other states.
3. **Concrete State:** Implements the state interface or extends the state abstract class. It provides the actual behavior associated with a specific state.



Okay, its time to get the vending machine working as expected.

```
/**  
 * The VendingMachine class represents a vending machine system.  
 * It maintains the current state of the vending machine and provides methods to  
 * interact with it.  
 */  
public class VendingMachine {  
    private VendingMachineState currentState;  
  
    /**  
     * Constructs a new VendingMachine object with the initial state set to  
     * NoCoinState.  
     */  
    public VendingMachine() {  
        currentState = new NoCoinState();  
    }  
  
    /**  
     * Inserts a coin into the vending machine.  
     */  
    public void insertCoin() {  
        currentState.insertCoin(this);  
    }  
  
    /**  
     * Selects a product from the vending machine.  
     */  
    public void selectProduct() {  
        currentState.selectProduct(this);  
    }  
  
    /**  
     * Dispenses the selected product from the vending machine.  
     */  
    public void dispenseProduct() {  
        currentState.dispenseProduct(this);  
    }  
}
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
/**  
 * Changes the state of the vending machine to the given newState.  
 *  
 * @param newState The new state to set for the vending machine.  
 */  
public void changeState(VendingMachineState newState) {  
    currentState = newState;  
}  
}  
  
/**  
 * The VendingMachineState interface represents the states of the vending  
 * machine.  
 * It defines methods that each concrete state class must implement.  
 */  
public interface VendingMachineState {  
    /**  
     * Inserts a coin into the vending machine.  
     *  
     * @param vendingMachine The vending machine object.  
     */  
    void insertCoin(VendingMachine vendingMachine);  
  
    /**  
     * Selects a product from the vending machine.  
     *  
     * @param vendingMachine The vending machine object.  
     */  
    void selectProduct(VendingMachine vendingMachine);  
  
    /**  
     * Dispenses the selected product from the vending machine.  
     *  
     * @param vendingMachine The vending machine object.  
     */  
    void dispenseProduct(VendingMachine vendingMachine);  
}  
  
/**  
 * The NoCoinState class represents the state of the vending machine when no  
 * coin has been inserted.  
 * It implements the VendingMachineState interface and provides the necessary  
 * behavior for this state.  
 */  
public class NoCoinState implements VendingMachineState {  
    /**  
     * Inserts a coin into the vending machine.  
     */
```

```
 * Changes the state to HasCoinState.
 *
 * @param vendingMachine The vending machine object.
 */
public void insertCoin(VendingMachine vendingMachine) {
    System.out.println("Coin inserted.");
    vendingMachine.changeState(new HasCoinState());
}

/**
 * Selects a product from the vending machine.
 * Prints a message to insert a coin first.
 *
 * @param vendingMachine The vending machine object.
 */
public void selectProduct(VendingMachine vendingMachine) {
    System.out.println("Please insert a coin first.");
}

/**
 * Dispenses the selected product from the vending machine.
 * Prints a message to insert a coin first.
 *
 * @param vendingMachine The vending machine object.
 */
public void dispenseProduct(VendingMachine vendingMachine) {
    System.out.println("Please insert a coin first.");
}

/**
 * The HasCoinState class represents the state of the vending machine when a
 * coin has been inserted.
 * It implements the VendingMachineState interface and provides the necessary
 * behavior for this state.
 */
public class HasCoinState implements VendingMachineState {
    /**
     * Inserts a coin into the vending machine.
     * Prints a message that a coin is already inserted.
     *
     * @param vendingMachine The vending machine object.
     */
    public void insertCoin(VendingMachine vendingMachine) {
        System.out.println("Coin already inserted.");
    }

    /**

```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
* Selects a product from the vending machine.  
* Prints a message that a product is selected and dispenses it.  
* Changes the state back to NoCoinState.  
*  
* @param vendingMachine The vending machine object.  
*/  
public void selectProduct(VendingMachine vendingMachine) {  
    System.out.println("Product selected. Dispensing product...");  
    vendingMachine.changeState(new NoCoinState());  
}  
  
/**  
 * Dispenses the selected product from the vending machine.  
 * Prints a message to select a product first.  
 *  
 * @param vendingMachine The vending machine object.  
 */  
public void dispenseProduct(VendingMachine vendingMachine) {  
    System.out.println("Please select a product first.");  
}  
}  
  
/**  
 * The Client class represents the client code that uses the VendingMachine  
 * system.  
 * It demonstrates the usage of the VendingMachine and its states.  
 */  
public class Client {  
    /**  
     * The main method creates a VendingMachine object and performs a sequence of  
     * actions on it.  
     *  
     * @param args The command-line arguments.  
     */  
    public static void main(String[] args) {  
        VendingMachine vendingMachine = new VendingMachine();  
        vendingMachine.insertCoin();  
        vendingMachine.selectProduct();  
        vendingMachine.dispenseProduct();  
    }  
}  
*****OUTPUT*****  
  
Coin inserted.  
  
Product selected. Dispensing product...  
  
Please insert a coin first.
```



In this example, *VendingMachine* class represents the **context** and maintains a reference to the **current state** object. It provides methods for inserting a coin, selecting a product, and dispensing the product. *VendingMachineState* interface represents the state and defines methods for inserting a coin, selecting a product, and dispensing the product. The **concrete state** classes, *NoCoinState* and *HasCoinState*, implement the state interface and handle the behavior associated with the respective states.

The client code creates an instance of *VendingMachine* and simulates a series of actions. Initially, the vending machine is in *NoCoinState*, and when a coin is inserted, the state changes to *HasCoinState*. The product can then be selected, and if valid, the product is dispensed, and the state changes back to *NoCoinState*.



State pattern helps vending machine to exhibit different behaviors based on its internal state. The pattern allows for easy extension of behaviors by adding new state classes, and it simplifies the management of state transitions and associated behaviors.

The State pattern replaces conditional statements with polymorphic behavior. Instead of having numerous conditional statements based on the object's state, the pattern delegates behavior to the current state object. This simplifies the code and improves readability. It provides a clean approach to manage state transitions. State objects can handle transitions by changing the current state of the context object, allowing for seamless and controlled changes in behavior.

6.8. Strategy Pattern



Strategy Pattern enables an object to alter its behavior dynamically by encapsulating a family of interchangeable algorithms (put each of them into a separate class) and making them (objects) interchangeable at runtime.

Imagine a group of individuals faced with a challenging task. Each person has their own unique approach or strategy to tackle the problem based on their skills, experiences, and preferences. They apply different strategies to solve the problem and achieve their goals. Person A might have a strategy of analysing the problem logically, breaking it down into smaller components, and applying a step-by-step approach to find a solution. Person B might have a strategy of relying on creativity and innovation, thinking outside the box, and exploring unconventional approaches to solve the problem. Person C might have a strategy of seeking guidance from experts, conducting thorough research, and basing their solutions on existing knowledge and best practices.

We leverage same human behavioral pattern in implemented strategy pattern, by letting the client to choose the strategy and varying the outcome based on selected strategy.

Okay, let us understand this via an example application.

Say, you are developing a shopping cart system, which allows customers to add items for purchase and calculates the total amount. However, the system needs to handle different discount strategies based on the customer's membership status. The goal is to design a flexible and extensible shopping cart system that can dynamically apply different discount strategies based on customer types.



If we design this system without proper code management then we might face severe challenges. Implementing fixed discount logic directly in the shopping cart code can make it inflexible and difficult to modify or extend. It will be challenging to handle different discount rates based on customer types or future changes in discount policies. Directly coupling the discount logic with the shopping cart leads to a lack of separation of concerns and also breaks OCP principal. Inconsistent application of discount strategies can occur if the system relies on conditional statements or flags to determine the appropriate discount logic. This approach may lead to code duplication, increased complexity, and potential errors in managing different discount scenarios.



Strategy pattern can be applied to the shopping cart system to address these challenges. It provisions for the encapsulation of different discount strategies in separate strategy classes and dynamically selecting the appropriate strategy at runtime.

Strategy Pattern has following components:

1. **Context:** Represents the object that uses the strategy. It maintains a reference to the strategy object and delegates the algorithm execution to it. The context provides an interface for clients to interact with the strategy.
2. **Strategy:** Defines an interface or an abstract class that encapsulates the algorithm or behavior to be performed. It declares a method or multiple methods that represent the strategy.
3. **Concrete Strategy:** Implements the strategy interface or extends the strategy abstract class. It provides the actual algorithm or behavior implementation.



Let us look at the following code snippet to understand better.

```
/**  
 * Represents a shopping cart that applies a discount strategy to calculate the  
 * total amount.  
 */  
  
public class ShoppingCart {  
    private DiscountStrategy discountStrategy;  
  
    /**  
     * Sets the discount strategy for the shopping cart.  
     *  
     * @param discountStrategy The discount strategy to be set.  
     */  
    public void setDiscountStrategy(DiscountStrategy discountStrategy) {  
        this.discountStrategy = discountStrategy;  
    }  
  
    /**  
     * Calculates the total amount after applying the discount strategy.  
     *  
     * @param amount The original amount.  
     * @return The total amount after applying the discount.  
     */
```

```
public double calculateTotal(double amount) {
    return discountStrategy.applyDiscount(amount);
}

/**
 * Represents a discount strategy.
 */
public interface DiscountStrategy {
    /**
     * Applies the discount to the given amount.
     *
     * @param amount The original amount.
     * @return The discounted amount.
     */
    double applyDiscount(double amount);
}

/**
 * Represents a discount strategy for regular customers.
 */
public class RegularCustomerDiscountStrategy implements DiscountStrategy {
    /**
     * Applies the regular customer discount to the given amount.
     *
     * @param amount The original amount.
     * @return The discounted amount for regular customers.
     */
    public double applyDiscount(double amount) {
        // Apply regular customer discount logic
        return amount * 0.95;
    }
}

/**
 * Represents a discount strategy for premium customers.
 */
public class PremiumCustomerDiscountStrategy implements DiscountStrategy {
    /**
     * Applies the premium customer discount to the given amount.
     *
     * @param amount The original amount.
     * @return The discounted amount for premium customers.
     */
    public double applyDiscount(double amount) {
        // Apply premium customer discount logic
        return amount * 0.85;
    }
}
```

```

}

/**
 * Represents a client that uses the ShoppingCart class to calculate total
 * amounts.
 */
public class Client {
    public static void main(String[] args) {
        ShoppingCart cart = new ShoppingCart();
        // Set regular customer discount strategy
        cart.setDiscountStrategy(new RegularCustomerDiscountStrategy());
        double amount = 100.0;
        double total = cart.calculateTotal(amount);
        System.out.println("Total amount for regular customer: " + total);
        // Set premium customer discount strategy
        cart.setDiscountStrategy(new PremiumCustomerDiscountStrategy());
        total = cart.calculateTotal(amount);
        System.out.println("Total amount for premium customer: " + total);
    }
}

*****OUTPUT*****
Total amount for regular customer: 95.0

Total amount for premium customer: 85.0

```

 In above code example, *ShoppingCart* class represents the **context** object. It allows the discount strategy to be set dynamically and calculates the total amount based on the selected strategy. Each item in the cart can hold a reference to its own discount strategy object. By utilizing **object composition** alongside Strategy pattern, shopping cart system achieves greater flexibility and extensibility.

DiscountStrategy interface represents the strategy and defines *applyDiscount()* method, which encapsulates the discount algorithm. The concrete strategy classes, *RegularCustomerDiscountStrategy* and *PremiumCustomerDiscountStrategy*, implement discount strategy interface and provide respective discount logic.

The client code demonstrates how shopping cart uses different discount strategies based on the customer's membership status. First, regular customer discount strategy is set, and the total amount is calculated. Then, premium customer discount strategy is set, and the total amount is recalculated.

 By applying **Strategy pattern**, the shopping cart system allows for easy switching between different discount strategies without modifying the cart's code. The Strategy pattern provides a modular and extensible approach to handle varying behaviors and algorithms in an elegant and maintainable manner. It promotes flexibility in adapting to different customer types and simplifies the addition of new discount strategies in the future. It follows the **Single Responsibility Principle** by assigning each behavior or algorithm to a separate strategy class. More importantly, it eliminates the need for conditional statements

that select different behaviors based on conditions. Instead, it provides a clear separation of concerns by encapsulating each behavior in a separate strategy class, improving code readability and maintainability.

6.9. Template Method Pattern

 **Template Method Pattern** allows to define the skeleton of an algorithm in a base class but allows subclasses to override specific steps of the algorithm without changing its overall structure.

One fine day, Mr John Wick decided to retire from the bounty hunting business and open a coffee corner. He had the challenge of growing his business in quick time, as he had no access to funds from the syndicate. Hence, he relied to making personal connections with his customers. He would observe his frequent customers, enquire about their taste and like and would make coffee to please their taste buds. Being a peculiarly discipline man, Mr Wick finalized upon a template for making a coffee and noted it down on piece of paper and stick it near his cooking window. As coffee making was not his natural instinct, he would often refer to this template while preparing coffee and would do necessary customizations for the customers in the specific steps defined in coffee preparation. Say, a customer asked for 2 teaspoon of coffee powder and other preferred just one teaspoon of sugar and no milk. He would fit it all within the template of coffee preparation and it helped him maintain a consistent process for making coffee. This consistency ensured that end result, a cup of coffee, meets the expectations of customers. In case Mr Wick decides to hire a few staffs, regardless of who is making the coffee, as long as they follow the template, the outcome will be the same.

Every cup of coffee starts with a boiling desire and a grinding determination, following the template to perfection. Just like John Wick followed his precise combat techniques, he adhered to the template method, grinding coffee beans with deadly precision.

Okay, let us all calm down, have a cup of coffee and look at a problem statement which will help understand **Template Method Pattern** better.

Say, we are tasked to develop a job application system, and there is a need to streamline the process of applying for different job positions while maintaining a consistent structure. However, each job position may have specific steps and requirements during the application process. The challenge is to design a flexible system that allows for customization of specific steps while ensuring a standardized overall process for job applications.

 Without a structured approach, the job application system may encounter several issues related to code duplication and maintenance challenges, due to implementing similar steps for each job position. Changes or updates to the application process may require modifications in multiple places, making the system error-prone and difficult to maintain. Each application may have different steps, leading to confusion for applicants and difficulties in tracking and managing applications consistently.

 **Template Method Pattern** helps us define a blue-print or a template which is to be adhered to complete processes or procedures share a common structure but have specific steps or behaviors that need customization.

It has three main participants:

1. **Abstract Class (Template):** Represents the base class that defines the template method, which outlines the algorithm's structure and invokes the steps in a specific order. It may also provide default implementations for some steps that can be overridden by subclasses.
2. **Concrete Classes:** Inherit from the abstract class and provide concrete implementations for the steps defined in the template method. These classes customize or override specific steps of the algorithm while maintaining the overall algorithm structure.
3. **Template Method:** The abstract class contains a template method that calls the abstract methods representing the common steps and allows the concrete classes to provide their own implementations for the category-specific steps.



Let us look at a code implementation that will help us visualize better.

```
/**  
 * This abstract class represents a job application.  
 */  
  
public abstract class JobApplication {  
    /**  
     * This method should be implemented by subclasses to fill in personal  
     * information for the job application.  
     */  
    protected abstract void fillPersonalInformation();  
  
    /**  
     * This method should be implemented by subclasses to upload the resume for the  
     * job application.  
     */  
    protected abstract void uploadResume();  
  
    /**  
     * This method should be implemented by subclasses to complete any additional  
     * forms required for the job application.  
     */  
    protected abstract void completeAdditionalForms();  
  
    /**  
     * This is the template method that orchestrates the job application submission  
     * process.  
     * It calls the required steps in a specific order and then submits the  
     * application.  
     */  
    public final void submitApplication() {  
        fillPersonalInformation();  
        uploadResume();  
        completeAdditionalForms();  
        submit();  
    }  
  
    /**
```

```
* This method is called by the template method to submit the job application.
* Subclasses can override this method to provide custom submission behavior.
*/
protected void submit() {
    System.out.println("Submitting job application...");
}

/**
* This class represents a job application for a software developer position.
*/
public class SoftwareDeveloperApplication extends JobApplication {
    /**
     * {@inheritDoc}
     */
    @Override
    protected void fillPersonallInformation() {
        System.out.println("Filling personal information for software developer application...");
    }

    /**
     * {@inheritDoc}
     */
    @Override
    protected void uploadResume() {
        System.out.println("Uploading resume for software developer application...");
    }

    /**
     * {@inheritDoc}
     */
    @Override
    protected void completeAdditionalForms() {
        System.out.println("Completing additional forms for software developer application...");
    }
}

/**
* This class represents a job application for a marketing manager position.
*/
public class MarketingManagerApplication extends JobApplication {
    /**
     * {@inheritDoc}
     */
    @Override
    protected void fillPersonallInformation() {
        System.out.println("Filling personal information for marketing manager application...");
    }
}
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
/*
 * { @inheritDoc}
 */
@Override
protected void uploadResume() {
    System.out.println("Uploading resume for marketing manager application...");
}

/*
 * { @inheritDoc}
 */
@Override
protected void completeAdditionalForms() {
    System.out.println("Completing additional forms for marketing manager application...");
}

/*
 * { @inheritDoc}
 */
@Override
protected void submit() {
    System.out.println("Submitting marketing manager application with references...");
}
}

/**
 * This class demonstrates the usage of the job application classes.
 */
public class Client {
    public static void main(String[] args) {
        // Creating a software developer job application
        JobApplication softwareDeveloperApplication = new SoftwareDeveloperApplication();
        softwareDeveloperApplication.submitApplication();
        System.out.println("-----");
        // Creating a marketing manager job application
        JobApplication marketingManagerApplication = new MarketingManagerApplication();
        marketingManagerApplication.submitApplication();
    }
}
*****OUTPUT*****
```

Filling personal information for software developer application...

Uploading resume for software developer application...

Completing additional forms for software developer application...

Submitting job application...

Filling personal information for marketing manager application...

Uploading resume for marketing manager application...

Completing additional forms for marketing manager application...

Submitting marketing manager application with references...



In this example, *JobApplication* class is the **abstract** class that defines the template method *submitApplication()*. It represents a generic job application and provides the overall algorithm structure for applying to a job. The template method includes steps for filling personal information, uploading a resume, completing additional forms, and finally submitting the application.

SoftwareDeveloperApplication and *MarketingManagerApplication* classes are **concrete** subclasses that inherit from *JobApplication*. They provide specific implementations for the abstract methods *fillPersonalInformation()*, *uploadResume()*, and *completeAdditionalForms()*. The *MarketingManagerApplication* class also overrides the *submit()* method to include references while submitting the application.



By using **Template Method pattern**, the overall structure of a job application is defined in the abstract class, while specific steps and behaviors are customized by the concrete subclasses. The template method ensures a consistent sequence of steps while allowing each subclass to implement its own variations. It promotes code reuse, eliminates code duplication, and provides a flexible approach to tailor the algorithm's behavior based on specific requirements.



Template Method pattern and Strategy pattern both aim to vary behavior, but they differ in their approach.

In **Template Method pattern**, inheritance is used to define the overall structure of an algorithm in a base class, while allowing subclasses to override specific steps of the algorithm. The base class contains a template method that calls various other methods, including abstract methods that are implemented by subclasses. This way, the base class controls the overall flow of the algorithm, while subclasses provide their own implementations for specific steps. Inheritance enables code reuse and promotes a consistent structure while allowing customization of certain parts.

On the other hand, **Strategy pattern** uses delegation to vary the entire algorithm. It encapsulates different algorithms or strategies in separate classes, each implementing a common interface or following a common contract. The context class contains a reference to the strategy interface and delegates the execution of the algorithm to the strategy object. The context can switch between different strategies at runtime, allowing the entire algorithm to be varied dynamically. The Strategy pattern promotes flexibility and encapsulation by allowing the behavior of the context object to be changed by swapping different strategies.

6.10. Visitor Pattern

 **Visitor Pattern** enables the separation of algorithms or operations from the objects they operate on. It allows adding new operations to existing object structures without modifying the objects themselves.

Say we are developing a document processing system to perform various operations on different types of documents, such as Word documents, PDFs, and spreadsheets. However, the system should allow adding new operations without modifying the document classes themselves and provide a flexible way to visit and operate on the elements of the document structure.

 If we do not follow a standardized approach to build the system and rely on plain inheritance, operations and algorithms might get tightly coupled with the document classes. Adding or modifying operations would require modifying the document classes, violating the **Open-Closed Principle**, and leading to a less maintainable and flexible system. Implementing similar operations for each document type individually may lead to code duplication.

 Visitor Pattern helps us avoid such situations via providing a way to separate the operations or algorithms from the document classes, allowing the addition of new operations without modifying the existing classes. It achieves this by introducing a visitor hierarchy that encapsulates the operations and can visit or operate on different elements of the document structure.

Visitor Pattern has following key participants:

1. **Visitor:** Declares a visit method for each element type within the object structure. This method is responsible for defining the operation to be performed on the visited element.
2. **Concrete Visitor:** Implements the visitor interface and provides the actual implementation of the visit methods for each element type.
3. **Element:** Defines the accept method that accepts a visitor and allows the visitor to visit and perform operations on the element.
4. **Concrete Element:** Implements the element interface and provides the implementation of the accept method, allowing the visitor to visit and operate on the element.
5. **Object Structure:** Represents a collection or structure of elements that can be visited by the visitor. It provides methods for adding or iterating over the elements.

 Let us look at a sample code implementation to see these participants in action.

```
/**  
 * This interface represents a visitor for different types of documents.  
 */  
public interface DocumentVisitor {  
    /**  
     * Visit a Word document.  
     *  
     * @param document The Word document to visit.  
     */  
    void visit(WordDocument document);  
  
    /**
```

```
 * Visit a PDF document.
 *
 * @param document The PDF document to visit.
 */
void visit(PDFDocument document);

/**
 * Visit a Spreadsheet document.
 *
 * @param document The Spreadsheet document to visit.
 */
void visit(SpreadsheetDocument document);
}

/**
 * This class implements the DocumentVisitor interface to calculate statistics
 * of visited documents.
 */
public class DocumentStatisticsVisitor implements DocumentVisitor {
    private int wordCount;
    private int pageCount;
    private int sheetCount;

    /**
     * Visit a Word document and update the word count.
     *
     * @param document The Word document to visit.
     */
    public void visit(WordDocument document) {
        wordCount += document.getWordCount();
    }

    /**
     * Visit a PDF document and update the page count.
     *
     * @param document The PDF document to visit.
     */
    public void visit(PDFDocument document) {
        pageCount += document.getPageCount();
    }

    /**
     * Visit a Spreadsheet document and update the sheet count.
     *
     * @param document The Spreadsheet document to visit.
     */
    public void visit(SpreadsheetDocument document) {
        sheetCount += document.getSheetCount();
    }
}
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
}

/**
 * Print the calculated statistics.
 */
public void printStatistics() {
    System.out.println("Word Count: " + wordCount);
    System.out.println("Page Count: " + pageCount);
    System.out.println("Sheet Count: " + sheetCount);
}

/** 
 * This interface represents a document.
 */
public interface Document {
    /**
     * Accept a DocumentVisitor to perform operations on the document.
     *
     * @param visitor The DocumentVisitor to accept.
     */
    void accept(DocumentVisitor visitor);
}

/** 
 * This class represents a Word document.
 */
public class WordDocument implements Document {
    private int wordCount;

    /**
     * Create a Word document with the given word count.
     *
     * @param wordCount The word count of the document.
     */
    public WordDocument(int wordCount) {
        this.wordCount = wordCount;
    }

    /**
     * Get the word count of the document.
     *
     * @return The word count.
     */
    public int getWordCount() {
        return wordCount;
    }
}
```

```
/**  
 * Accept a DocumentVisitor to perform operations on the Word document.  
 *  
 * @param visitor The DocumentVisitor to accept.  
 */  
public void accept(DocumentVisitor visitor) {  
    visitor.visit(this);  
}  
}  
  
/**  
 * This class represents a PDF document.  
 */  
public class PDFDocument implements Document {  
    private int pageCount;  
  
    /**  
     * Create a PDF document with the given page count.  
     *  
     * @param pageCount The page count of the document.  
     */  
    public PDFDocument(int pageCount) {  
        this.pageCount = pageCount;  
    }  
  
    /**  
     * Get the page count of the document.  
     *  
     * @return The page count.  
     */  
    public int getPageCount() {  
        return pageCount;  
    }  
  
    /**  
     * Accept a DocumentVisitor to perform operations on the PDF document.  
     *  
     * @param visitor The DocumentVisitor to accept.  
     */  
    public void accept(DocumentVisitor visitor) {  
        visitor.visit(this);  
    }  
}  
  
/**  
 * This class represents a Spreadsheet document.  
 */  
public class SpreadsheetDocument implements Document {
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
private int sheetCount;

/**
 * Create a Spreadsheet document with the given sheet count.
 *
 * @param sheetCount The sheet count of the document.
 */
public SpreadsheetDocument(int sheetCount) {
    this.sheetCount = sheetCount;
}

/**
 * Get the sheet count of the document.
 *
 * @return The sheet count.
 */
public int getSheetCount() {
    return sheetCount;
}

/**
 * Accept a DocumentVisitor to perform operations on the Spreadsheet document.
 *
 * @param visitor The DocumentVisitor to accept.
 */
public void accept(DocumentVisitor visitor) {
    visitor.visit(this);
}

/**
 * This class demonstrates the usage of the Visitor pattern.
 */
public class Client {
    public static void main(String[] args) {
        DocumentVisitor statisticsVisitor = new DocumentStatisticsVisitor();
        Document wordDocument = new WordDocument(500);
        Document pdfDocument = new PDFDocument(10);
        Document spreadsheetDocument = new SpreadsheetDocument(3);
        wordDocument.accept(statisticsVisitor);
        pdfDocument.accept(statisticsVisitor);
        spreadsheetDocument.accept(statisticsVisitor);
        ((DocumentStatisticsVisitor) statisticsVisitor).printStatistics();
    }
}
*****OUTPUT*****
```

Word Count: 500

Page Count: 10

Sheet Count: 3



In code example above, *DocumentVisitor* interface defines the **visitor** contract with *visit()* methods for each type of document. The **concrete visitor** class *DocumentStatisticsVisitor* implements the visitor interface and keeps track of statistics such as word count, page count, and sheet count.

Document interface represents the **element** interface that declares the *accept()* method, allowing visitors to visit and perform operations on different types of documents. The concrete document classes (*WordDocument*, *PDFDocument*, *SpreadsheetDocument*) implement *Document* interface and provide their own implementations of *accept()* method by invoking the appropriate visitor's *visit()* method.

In the client code, we create instances of different document types and a concrete visitor (*DocumentStatisticsVisitor*). We then invoke *accept()* method on each document, passing the visitor as an argument. The appropriate *visit()* method is called based on the document type, allowing the visitor to perform the specific operation on each document.



By using **Visitor Pattern**, the document processing system achieves separation of concerns by encapsulating different operations in visitor classes. New operations can be added by implementing additional visitor classes without modifying the document classes. The Visitor pattern promotes extensibility, code modularity, and flexibility in performing operations on elements within a complex object structure.



Double Dispatch: The Visitor pattern implements double dispatch, where the type of the visitor and the type of the element determine the actual method to be called. This enables dynamic method resolution based on the runtime types of both the visitor and the visited element, allowing for flexible and extensible operations.

Visitor pattern is suitable when we have a complex object structure and want to perform operations that are specific to different elements within the structure. Some examples include calculating statistics, performing transformations, or generating reports on various parts of the structure.

6.11. Summary

Behavioral design patterns, taking the cues from human behavioral patterns to solve different problems, focus on defining the interaction and communication between objects to achieve flexible and reusable designs. These patterns address the behavior and communication patterns among objects to provide effective solutions for complex systems and help in managing algorithms, encapsulating behaviors, and organizing the responsibilities and interactions among objects.

Following are the patterns covered in this section:

1. **Chain of Responsibility Pattern:** Establishes a chain of objects where each object has the capability to handle a request or pass it on to the next object in the chain until the request is handled or reaches the end of the chain.

2. **Command Pattern:** Encapsulates a request as an object, thereby allowing parameterization of clients with different requests, queue, or log requests, and support undoable operations.
3. **Iterator Pattern:** Provides a way to access elements of an aggregate object sequentially without exposing the underlying representation. It decouples the iteration algorithm from the aggregate object.
4. **Mediator Pattern:** Provides a central mediator object that encapsulates the communication between a set of objects. It promotes loose coupling by eliminating direct dependencies between communicating objects and allows them to communicate with each other through the mediator.
5. **Memento Pattern:** Allows capturing and restoring the internal state of an object without violating encapsulation. It enables an object to be restored to a previous state, undoing changes, or providing rollback functionality.
6. **Observer Pattern:** Allows objects to establish a one-to-many dependency, where changes in one object are notified to and automatically reflected in other dependent objects.
7. **State Pattern:** Allows an object to alter its behavior when its internal state changes. It encapsulates each state as a separate class and enables the object to switch between different states dynamically.
8. **Strategy Pattern:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable. It allows algorithms to be selected at runtime based on the context.
9. **Template Method Pattern:** Defines the skeleton of an algorithm in a base class, allowing subclasses to provide specific implementations of certain steps of the algorithm while maintaining the overall structure.
10. **Visitor Pattern:** Defines a new operation to be performed on the elements of an object structure without modifying the classes of the elements themselves. It separates the operations from the objects by encapsulating them in visitor classes.

SECTION 2

DATA

Chapter 7

Database Systems

“

“Data is a precious thing and will last longer than the systems themselves.” - Tim Berners-Lee

So far we have studied about Objects, their composition and behavioral patterns. We look at the entities in the world around us and model it in terms of objects or data structures and we create APIs to manipulate them. But we want to preserve this structure and we do so in form of data models being stored on physical devices in form of disks, memory, cache, etc. As Martin Kleppmann mentions in his highly appreciated book, *Designing Data Intensive Application*, “*On the most fundamental level, a database needs to do two things: when you give it some data, it should store the data, and when you ask it again later, it should give the data back to you.*”

We have already covered history of database and its evolution in the first chapter. In this section, we will get an overview of **database systems**, which encompasses data storage and management. Database systems play a pivotal role in organizing and handling vast amounts of data efficiently. We will delve into an overview of these systems, examine key data models **SQL**, **NoSQL**, **BigData**, etc. prevalent in modern times, and gain a concise understanding of how data is stored within them.

Understanding **database systems** entails grasping the fundamental concepts and principles that govern the organization, retrieval, and manipulation of data. We will examine the various components of a database system, such as the **database management system (DBMS)**, which serves as the software infrastructure for interacting with databases.

Within the realm of data models, we will explore several prevalent approaches to structuring and representing data, including **relational and non relational (No SQL) models**. Each model brings its own strengths and weaknesses, and understanding these models aids in selecting the most appropriate one for a given application's requirements.

As we progress, we will connect dots between object-oriented programming and data modeling, recognizing how the principles and concepts underlying object-oriented design can be aligned with database systems. This alignment facilitates the development of enterprise architectures where objects in applications can seamlessly interact with data stored in databases.



By studying the interplay between objects and data, we gain a deeper understanding of how enterprise architecture takes shape, with application developers leveraging the power of database systems to model and manipulate data effectively. This knowledge equips us to make informed decisions and design robust systems that bridge the gap between the real world and the digital realm, enabling efficient data storage, management, and retrieval within the context of modern software applications.

7.1. Database Management System

A **Database Management System**, commonly known as **DBMS**, is a software system that facilitates defining, constructing, manipulating, and sharing of data and a set of programs to access those data.

Collection of data, known as **database**, contains data types, structures, and constraints of the data relevant to an enterprise. A database management system enables manipulation of database via query, updates, etc. and sharing of database among users and program simultaneously.

The role of a **Database Management System (DBMS)** is to facilitate the storage, organization, and retrieval of this database information in a manner that is both convenient and efficient. It provides a user-friendly interface and tools that allow users to define the structure of the database, perform data manipulation operations, and retrieve information based on specific criteria. It aims to provide efficient storage and retrieval mechanisms to minimize the time and resources required to access and manipulate data. It employs various techniques, such as **indexing, caching, query optimization, and transaction management**, to enhance performance and ensure that database operations are executed in a timely manner.

7.1.1. Queries and Transaction

An application program interacts with the database by sending queries or requests to the DBMS. These queries can be used to retrieve specific data or perform operations that involve reading or writing data within the database.

A **query** is a request made by the application program to the DBMS to retrieve specific data from the database. It typically consists of a **structured query language (SQL)** statement that specifies the desired data and any conditions or criteria for the retrieval. A **query language** is a specialized language used to interact with databases and retrieve information from them. The DBMS processes the query and returns the requested data in a format that is suitable for consumption by the application program.

On the other hand, a **transaction** is a logical unit of work performed by the application program that may involve both reading and writing data within the database. A transaction typically consists of multiple database operations that need to be executed as a single, indivisible unit. For example, a transaction may involve reading customer details, checking product availability, and updating inventory levels.

During a transaction, the application program can read data from the database to gather information or perform calculations. This read operation retrieves the requested data without modifying it. Additionally, the transaction may involve writing data into the database, which involves updating existing records or creating new ones. These write operations modify the data stored in the database to reflect the changes made by the application program.

The DBMS ensures the consistency and integrity of the database during these operations. For example, when a transaction reads data, the DBMS ensures that the data remains unchanged and is not affected by other concurrent transactions. When a transaction writes data, the DBMS ensures that the changes made by the transaction are durable and persist even in the event of failures or system crashes.

The interaction between the application program and the DBMS, through queries and transactions, allows the application to retrieve and manipulate the necessary data from the database. This enables the application program to perform its intended functionality, whether it involves retrieving specific data subsets, generating reports, or updating information within the database.

7.1.2. Data Abstraction

To ensure efficient data retrieval, database system developers employ complex data structures to represent data in the database. However, as many users of database systems may not have a technical background, developers utilize multiple levels of data abstraction to hide this complexity and simplify user interactions with the system.

- **Physical Level:** This is the lowest level of abstraction, which describes how the data is physically stored on the storage media. It encompasses details such as disk blocks, file organization, indexing techniques, and other low-level data structures. The physical level deals with the technical aspects of data storage and retrieval.
- **Logical Level:** The logical level is the next-higher level of abstraction. It describes what data is stored in the database and the relationships that exist among the data. At this level, the entire database is represented in terms of a relatively small number of simple structures, such as **tables, views, and relationships**. The logical level provides a conceptual view of the database that is independent of the physical storage details. This means that changes in the physical level (e.g., storage reorganization) do not affect the way users perceive or interact with the data. Database administrators work at this level to determine the organization and structure of the database.
- **View Level:** The view level is the highest level of abstraction. It represents a specific subset of the database that is relevant to a particular user or group of users. The view level simplifies user interactions by presenting a customized and tailored view of the data, containing only the information necessary for a specific user's needs. It is not always desirable for all users to see the entire set of relations in the database. Multiple views can be created from the same underlying logical data structure, allowing different users to access and manipulate specific subsets of the database without exposing the entire complexity of the logical level.

Consider a University database as an example, which contains details about courses taught in university, instructors, departments, which instructors are mapped to which departments and courses, students who have enrolled for these courses and so on. Let us look at a section of database which contains relevant information about instructors.

Following table ([Figure 7-1](#)) contains information related to all the Instructors in University database. It represents **Logical Level** of data abstraction.

id	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000

Figure 7-1 Instructor Table

Say, there is an accountant in the university who does not have permission to view salary of instructors but needs to know other details. Therefore, a **view level** of abstraction can be prepared for accountant ([Figure 7-2](#)). In the context of a database system, a view relation represents a subset or derived data set from the underlying tables of the database. Unlike base relations (tables) that store data physically, a view relation is not precomputed and stored as tuples. Instead, the database system stores the query expression associated with the view relation.

id	name	dept_name
10101	Srinivasa	Comp. Sci.
12121	Wu	Finance
15151	Mozart	Music
22222	Einstein	Physics
32343	El Said	History
33456	Gold	Physics
45565	Katz	Comp. Sci.
58583	Califieri	History
76543	Singh	Finance
76766	Crick	Biology
83821	Brandt	Comp. Sci.

Figure 7-0-1 Faculty View

When a user accesses the view relation, its tuples are dynamically created by computing the query result based on the stored query expression. This means that the view relation is generated on-demand, whenever it is accessed or queried.

On a **physical level**, data records can be stored as a block of consecutive bytes. However, its implementation detail is abstracted from the users of Database systems.

7.1.3. Instances and Schema

The specific collection of data stored in the database at any given point in time is referred to as an **instance** of the database. It represents the snapshot or the current state of the data within the database.

Databases undergo changes as information is inserted, updated, or deleted over time. An instance of the database comprises the actual data stored in the tables, including the records, values, and relationships between entities. It reflects the cumulative effect of all the past and ongoing data modifications and operations that have occurred within the database.

The **database schema** refers to the overall design or blueprint of the database. It defines the structure, organization, and logical representation of the database objects, such as tables, views, indexes, constraints, and relationships between entities. It outlines the logical framework that determines how the data is structured and how different entities relate to each other ([Figure 8-1](#)).

In the next chapter, we will be looking at some schema examples of different data models.

7.2. Database Languages

Database systems provide two categories of commands to play with database schema. **Data Definition Language (DDL)** is used to define and manage the structure and organization of a database (schema) and **Data Manipulation language (DML)** is used to express database queries and updates.

7.2.1. Data Definition Language (DDL)

The primary purpose of **Data Definition Language (DDL)** is to define the schema or the logical structure of the database. It allows users to specify the data types, relationships, constraints, and other properties associated with the database objects. DDL statements are typically executed by database administrators or authorized users who have the necessary privileges to modify the database structure.

DDL statements are also used to allow Database Admins to define and enforce different levels of access privileges for users or user groups. These access privileges determine what actions users can perform on specific data values or database objects. By assigning appropriate access permissions, the DBMS ensures that users can only interact with the data according to their authorized privileges, enhancing data security and confidentiality.

7.2.2. Data Manipulation Language (DML)

Data Manipulation Language (DML) is a set of commands in Database systems which allows users to manipulate or interact with the data stored in a database. DML provides a set of syntax and operations to perform tasks such as querying, inserting, updating, and deleting data within the database.

Procedural DMLs: Procedural DMLs require users to specify both the data they need (what) and the specific steps or procedures (how) to obtain that data. Users need to provide detailed instructions on how to navigate the database and perform the necessary operations. Procedural DMLs are often associated with programming languages and involve writing code or scripts to manipulate data. Examples of procedural DMLs include SQL/PSM (Persistent Stored Modules) and PL/SQL (Procedural Language/Structured Query Language).

In a procedural DML, users define the sequence of operations, loops, conditions, and control flow statements to retrieve, insert, update, or delete data. These DMLs offer a high degree of control and allow for complex data manipulations. However, they require users to have a deeper understanding of the database structure and programming concepts.

Declarative DMLs: Declarative DMLs, also known as nonprocedural DMLs, focus on specifying what data is needed without explicitly stating how to obtain it. Users express their data requirements through queries or statements, and the database management system (DBMS) takes care of determining the most efficient way to retrieve or manipulate the data. The most widely used declarative DML is **SQL (Structured Query Language)**.

In a declarative DML, users define the desired data using SELECT statements for data retrieval or INSERT, UPDATE, DELETE statements for data modification. The DBMS processes these statements and generates an execution plan, optimizing the query execution based on internal algorithms and indexing strategies. Declarative DMLs provide a more intuitive and user-friendly approach to interact with databases, as users focus on specifying what they need rather than the procedural steps to achieve it.

Declarative DMLs offer several advantages over Procedural DMLs.

Simplicity: Users can express their data requirements using SQL statements, which are concise and easier to write and understand compared to procedural code.

Efficiency: DBMSs can optimize the execution of declarative queries, leveraging internal mechanisms like query optimization, indexing, and caching to improve performance.

Portability: Declarative DMLs are more portable across different DBMSs because they adhere to standardized SQL syntax. This allows users to write SQL statements that can be executed in various database environments.

7.3. Database Internals

So far, we have had an overview of different interfaces of Database Systems to perform necessary tasks related to data. Functionality of these interfaces are encapsulated under functional components of a database system.

The functional components of a database system can be broadly categorized into three main components: the **storage manager**, the **query processor**, and the **transaction management** component (see [Figure 7-2](#)).

7.3.1. Storage Manager

The storage manager is responsible for managing the physical storage of data in the database. It is important because databases typically require a large amount of storage space. Corporate databases commonly range in size from hundreds of gigabytes to terabytes of data. Since, computer memory is not sufficient to store this much of information and information might also get lost in case of system failures, information is stored on physical media such as disks.

Data movement between disk storage and main memory is a crucial aspect of database systems. However, this process is relatively slow compared to the speed of the **central processing unit (CPU)**. To optimize performance, a well-designed database system aims to minimize the need for data movement between disk and main memory.

Buffer Pool/Caching: Database systems utilize a buffer pool, which is a portion of main memory reserved for caching frequently accessed data pages. When data is read from disk, it is stored in the buffer pool, allowing subsequent reads to be serviced from memory, which is significantly faster than disk access. This helps reduce the number of disk I/O operations required.

Indexing: Indexes are data structures that provide quick access to specific data values. By creating appropriate indexes on frequently accessed columns or attributes, the database system can locate the desired data more efficiently. This minimizes the need for scanning entire data files and reduces disk I/O.

Data Placement and Partitioning: Careful data placement and partitioning strategies can help improve data access performance. For example, storing related data together on disk (data clustering) can reduce disk I/O by minimizing the need to access multiple disk locations for a single query. Partitioning data across multiple disks can also distribute the I/O load, improving overall performance.

Query Optimization: The query optimizer within the database system analyses queries and generates an optimized execution plan. This plan considers factors such as available indexes, join strategies, and access paths to minimize disk I/O. By selecting efficient query execution plans, the optimizer reduces the need for excessive data movement.

Systems have transitioned to **solid-state disks (SSDs)** for data storage as they offer faster access time and lower latency compared to traditional magnetic disks. SSDs are particularly beneficial for random access patterns, which are common in database workloads. With faster read and write speeds, SSDs can significantly reduce the latency associated with disk I/O operations.

7.3.1.1. Storage Manager Components

The storage manager component of a database system includes following important subcomponents:

- **Authorization and Integrity Manager:** This component is responsible for enforcing data security and integrity constraints. It verifies the authority of users to access specific data and ensures that data modifications and operations adhere to defined integrity rules and constraints. The authorization and integrity manager plays a crucial role in maintaining data confidentiality, privacy, and consistency.

- **Transaction Manager:** The transaction manager is responsible for ensuring the consistency and durability of transactions within the database system. It manages the execution of multiple concurrent transactions, ensuring that they proceed without conflicts and that the database remains in a consistent state despite system failures. The transaction manager oversees transaction initiation, coordination, concurrency control, and crash recovery to maintain data integrity.
- **File Manager:** The file manager handles the allocation of space on disk storage and manages the data structures used to represent information stored on disk. It provides functionalities for creating, opening, and closing database files, as well as reading from and writing to these files. The file manager organizes and tracks the physical storage of database objects, such as tables, indexes, and views, on the disk.
- **Buffer Manager:** The buffer manager is a critical component of the storage manager as it facilitates efficient data retrieval and caching. It manages the movement of data between disk storage and main memory (buffer pool), deciding what data to fetch from disk into memory and what data to cache. The buffer manager aims to minimize disk I/O operations by keeping frequently accessed data pages in memory, improving overall data access performance. It enables the database system to handle datasets that exceed the available main memory size.

7.3.2. Query Processor

The query processor is responsible for transforming user queries written in a query language (such as SQL, Mongo queries, etc.) into an execution plan that the database system can understand and execute.

Some key operations of the query processor are listed below:

- **Query Parsing and Optimization:** The query processor parses user queries, validates their syntax, and performs semantic analysis. It then generates an optimized execution plan based on various factors like available indexes, query selectivity, and join strategies. This optimization phase aims to minimize the query execution time.
- **Query Execution:** The query processor executes the generated execution plan to retrieve the requested data. It interacts with the storage manager to access the required data and performs operations such as joins, aggregations, and sorting based on the query requirements.
- **Query Result Presentation:** Once the query execution is complete, the query processor presents the query results to the user in a desired format, such as tabular data or reports.

7.3.2.1. Query Processor Components

The query processor can perform above operations with the capabilities extended by following subcomponents:

- **DDL Interpreter:** The DDL (Data Definition Language) interpreter is responsible for interpreting DDL statements, such as CREATE, ALTER, and DROP statements. It records the definitions and metadata associated with database objects, such as tables, views, indexes, and constraints, in the data dictionary or system catalogue. The data dictionary stores the database schema and other information about the structure and organization of the database.
- **DML Compiler:** The DML (Data Manipulation Language) compiler translates DML statements, typically written in a query language like SQL, into an evaluation plan. The evaluation plan comprises a series of low-level instructions that the query evaluation engine can execute. The DML compiler also performs query optimization, analysing the query and generating the most efficient evaluation plan based on factors like available indexes, query selectivity, join strategies, and access paths. The goal is to select the evaluation plan with the lowest cost in terms of resource utilization and query execution time.

- **Query Evaluation Engine:** The query evaluation engine executes the low-level instructions generated by the DML compiler. It interprets and processes the evaluation plan to retrieve the desired data from the database. The engine interacts with other components, such as the storage manager, to access the necessary data pages and perform operations like joins, aggregations, sorting, and filtering based on the evaluation plan. The query evaluation engine is responsible for efficiently executing the instructions to produce the desired query result.

7.3.3. Transaction Manager

Transaction management is a critical component of a database management system (DBMS) that ensures the reliable and consistent execution of **transactions** within the database.

A **transaction** represents a logical unit of work that consists of a sequence of database operations, such as data modifications (insertions, updates, deletions) or read operations. A database transaction shall happen in all its entirety or it shall not be executed at all. There is no middle ground. All participating resources are in a consistent state both when the transaction begins and when the transaction ends.

Transactions are often related with banking domain and rightly so. Say, we are working on a banking system. A customer transfers certain amount from his/her bank account to another account of his wife. For the customer entity, it is a debit however, for wife's account it is a credit. And both events shall be replicated in the system correctly. In case, there is a failure in network or system crash on any end, the entire transaction shall be rolled back, and the initial state should be maintained.

In modern times, situation is lot more complex. With online banking, it is possible that more than one person is trying to perform transactions on one bank account (concurrency), and in every scenario, system should be able to reflect the absolute sum value of money transferred among entities.

7.3.3.1. ACID PROPERTIES

Handling concurrency is the main goal for leveraging transactions in software systems. Following are the main properties of a Software Transaction:

- **Atomicity:** Atomicity ensures that a transaction is treated as an indivisible unit of work. It guarantees that either all operations within a transaction are successfully completed and committed, or none of them are. If any operation fails or encounters an error, the entire transaction is rolled back, and the database remains unchanged.
- **Consistency:** Consistency ensures that a transaction brings the database from one consistent state to another. It enforces integrity constraints, rules, and relationships defined in the database schema, ensuring that they are satisfied before and after the transaction execution. Consistency prevents data inconsistencies and ensures data integrity.
- **Isolation:** Isolation ensures that concurrent transactions are executed in isolation from one another. Each transaction appears to execute sequentially, even though multiple transactions may be executed concurrently. Isolation prevents interference and conflicts between transactions and maintains data consistency.
- **Durability:** Durability ensures that once a transaction is committed, its changes are permanently saved and will survive future system failures or crashes. Committed data is written to non-volatile storage, such as disk, to guarantee its durability. This allows the database system to recover the committed state of the database, even after a failure, ensuring data persistence.

7.3.3.2. Transaction Manager Components

- **Recovery Manager:** It is responsible for maintaining atomicity and durability within a transaction. It makes sure that a failed transaction has no effect on the state of database and any changes made are restored to original state. It detects system failures across transaction.
- **Concurrency Control Manager:** In case where multiple transactions are attempting to update database at one instance only, concurrency control manager is responsible to maintain consistency of database state.

The transaction manager in a local database system just needs to inform the recovery manager of their choice to commit a transaction. However, in a distributed system, the transaction manager should consistently enforce the decision to commit and communicate it to all the servers in the various sites where the transaction is being conducted.

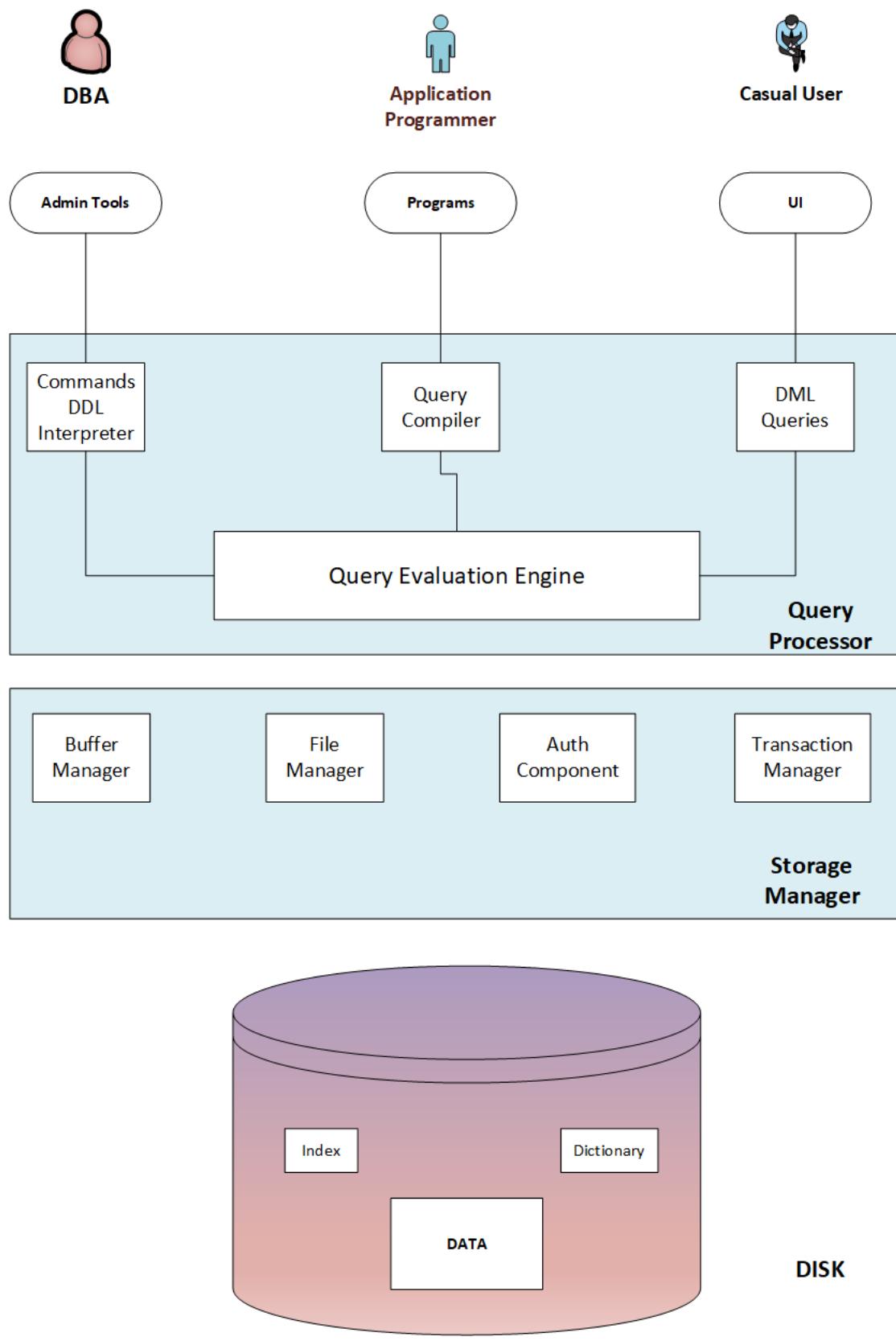


Figure 7-0-2

The [diagram](#) above shows different layers of architecture beneath a DBMS application.

7.4. Distributed Databases

Once upon a time, there was a small e-commerce company 'G-Mart' that relied on a single powerful mainframe computer to process all its data and run its applications. They had a state-of-the-art mainframe system, but it struggled to keep up with the skyrocketing number of users accessing their online store. As the traffic increased, the mainframe became overwhelmed, causing slowdowns and even occasional crashes. One fateful day when disaster struck, the mainframe crashed, causing a complete halt to the company's operations. The company suffered significant losses, and its reputation took a hit.

Learning from this incident, the company's technology team started thinking about ways to prevent such catastrophes in the future. They realized that putting all their eggs in one basket—relying on a single mainframe—was a risky strategy. They needed a more robust and fault-tolerant solution.

After researching and brainstorming, the team came up with an idea: What if they could break down the workload and data across multiple smaller computers, interconnected over a network? This way, even if one computer failed, others could pick up the slack and ensure the company's operations continued smoothly. They envisioned creating a network of interconnected machines, each taking care of a portion of the load. This way, they could achieve better performance, scalability, and fault tolerance.

*The team's vision materialized into what they called a "distributed system." Their ambitious plan led them to build several small servers, each specialized in handling a specific task. They would call them "**nodes**." These nodes communicated with each other over a network, sharing data and coordinating their efforts to fulfil user requests. Each node had its own processing power, memory, and storage. These nodes could communicate and coordinate with each other to handle the company's data and applications collectively.*

*The transition to a distributed system was not without challenges. The engineers at G-Mart faced the task of designing a system that could maintain data consistency across all nodes, ensuring that users would always get accurate and up-to-date information. They introduced a concept of "**replication**," where data was duplicated and stored on multiple nodes to ensure redundancy and availability. This way, even if one node failed, the data remained accessible from other nodes. With database replication in place, the read requests from customers were distributed across the replica servers, alleviating the read load from the primary server. This load balancing improved the system's response times and enhanced the overall performance of the platform.*

As they developed the distributed system, the team discovered several benefits. Scalability was a significant advantage as they could now easily add more nodes as their business grew, ensuring that the system could handle increasing data volumes and user demands. This scalability allowed them to cater to millions of users concurrently, providing a seamless shopping experience for everyone.

The newfound fault tolerance was a game-changer. If any node failed, the distributed system could redistribute the workload to other healthy nodes, preventing a complete shutdown of operations. This not only prevented catastrophic failures like the previous mainframe crash but also allowed them to perform maintenance and updates without disrupting the entire system.

The performance of the distributed system was remarkable too. With the workload distributed among several nodes, they noticed reduced response times and better computational efficiency, which greatly improved their applications' performance and user experience.

As time went on, the company expanded its operations to multiple locations. With a distributed system, they could easily span their infrastructure across different data centres, reducing network latency and ensuring data redundancy for disaster recovery. With added benefit of geographical distribution, they opened data centres in various locations, spreading their infrastructure across the globe.

Different teams could work simultaneously on various projects, accessing shared resources and data from the distributed network. This flexibility and decentralization brought efficiency and agility to the organization.

The flexibility and cost-effectiveness of the distributed system were also commendable. They no longer needed to invest in expensive mainframes; instead, they could use commodity hardware for their nodes, making it more economical.

The story of the small company's journey from a single mainframe to a robust and fault-tolerant distributed system became an inspiration for many others. The idea of breaking down complexity, distributing workloads, and ensuring resilience caught on, becoming the foundation of modern distributed systems that power the internet, cloud services, social media platforms, and countless other applications we rely on today.

7.4.1. Distributed Database Management System (DDBMS)

A **distributed database (DDB)** is a collection of multiple logically interrelated databases that are spread across different locations and connected through a computer network. Each of these databases, known as nodes, can be hosted on separate servers, data centres, or even in different geographical regions.

A **distributed database management system (DDBMS)** is a software system designed to manage a distributed database. It provides an integrated and transparent view of the distributed data to the users and applications. Users interact with the DDBMS as if it were a single, centralized database, even though the data is physically distributed across multiple nodes.

The information stored in the various database nodes must be logically related. The data distributed across different nodes should have some form of commonality or relationship, enabling users and applications to access and query the data as if it were a single coherent database. Users and applications are unaware of the distribution and location of data. This is called transparency in DDBMS.

In some cases, data might not be distributed but replicated. Replicas of data are stored on multiple nodes, ensuring that if one node fails, the data can still be accessed from another replica. DDBMS supports this feature to improve fault tolerance and data availability.

However, the nodes in a distributed database do not need to be identical in terms of data, hardware, and software. They can have different configurations, data sets, and database management systems. Each node in a DDBMS operates autonomously, allowing it to handle its local data independently. This autonomy enhances system efficiency and allows for better load distribution.

Distributed concurrency control mechanisms are employed across DDBMS cluster, to manage concurrent data access from multiple users across different nodes. This ensures data consistency and prevents conflicting updates. Further, distributed transactions are also supported adhering to ACID properties.

Let us now understand different concepts, which come to play while creating a distributed system of databases.

7.4.2. Database Replication

Replication is the concept of maintaining multiple copies of your data.

Replication is a vital aspect of distributed systems, forming its core over time. Its significance lies in ensuring continuous data access, even when some nodes or servers might be unavailable due to downtime, disasters, or maintenance work. By replicating data across multiple servers, the risk of losing access to critical information during server failures is greatly reduced. This redundancy ensures that data remains available, even if one server becomes inaccessible.

The point of replication is to make sure that in the event your node goes down, you can still access your data using a standby database.

7.4.2.1. How does Replication work?

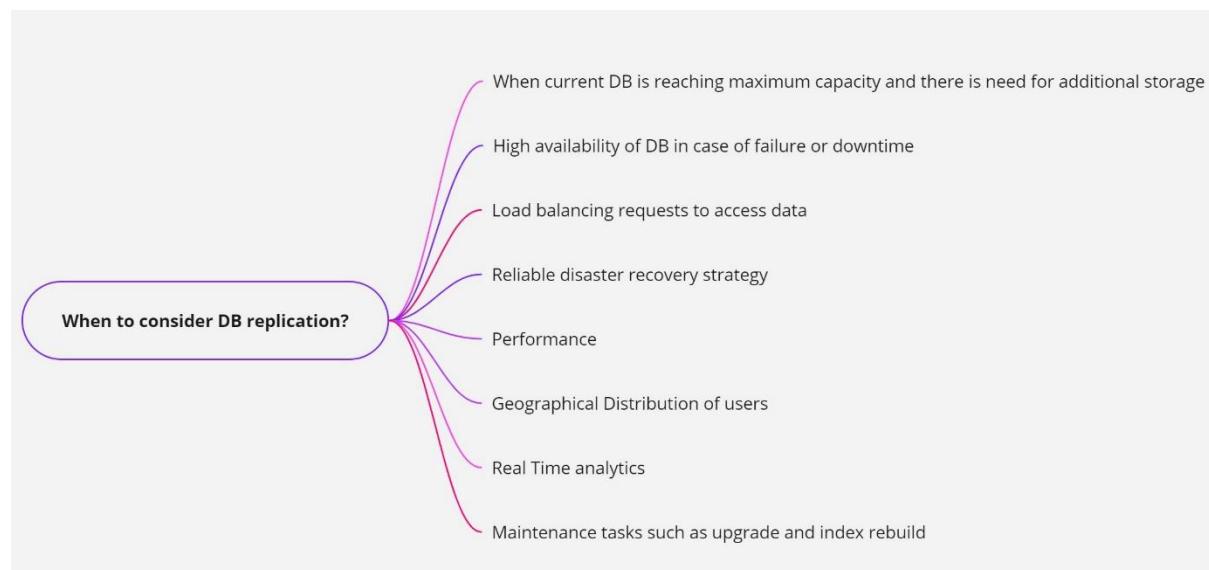
In a replicated system, there are a couple extra nodes (read: database server) on hand, and they hold copies of our data. A group of nodes that each have copies of the same data is called a **replica set**.

In a replica set, entire data is handled by default through one of the nodes(master), and it's up to the remaining nodes(slaves) in the set to sync up with it and replicate any new data that's been written through an asynchronous mechanism. This mechanism is also known as **Master-Slave system**, which serves the purpose that all nodes stay consistent to each other.

In times of disaster when the primary node goes down, one of the secondary nodes can take its place as primary in a process known as **failover**. The process of selecting the new master or primary node is literally called as **election**, where, nodes vote for one another.

To maintain a durable system, the process of failover happens in a very quick time and the end application will not even sense as if something has gone wrong at all. After, the original primary node comes back to life, it catches up on the missed data via process of syncing and re-joins the replica set.

Availability and redundancy of data are typical properties of a durable database solution.



In coming chapters, we shall be discussing about data models in detail. Primarily there are two classification, one is relational model also known as SQL based and other non-relational, also known as NoSQL. However, in the context of data replication, let us briefly discuss how various replication strategies

can be considered in case of different data models. Replication plays a vital role in ensuring data consistency and availability across distributed systems. Each data model presents unique characteristics and requirements, necessitating different replication approaches.

7.4.2.2. Replication strategy for Relational Database

In the relational model, characterized by SQL-based databases, there are three main strategies: **transactional, snapshot, and merge replication**, each serving distinct use cases and scenarios.

7.4.2.2.1. Transactional Replication

Transactional replication is characterized by real-time replication of transactions or changes from the original database to the replicated database in a sequential manner. This means that users experience these changes on the replicated database almost instantly, ensuring transactional consistency. This type of replication is commonly used in server-to-server environments where data needs to be synchronized in real-time across multiple servers. The advantage of transactional replication lies in its ability to provide up-to-date and consistent data across all replicas, ensuring that the distributed system operates seamlessly.

When a transaction commits in an **ACID-compliant database**, the system immediately writes a transaction record to the **transaction log**. The transaction log serves as a chronological record of all committed transactions, including their before and after states. This log acts as a safeguard against potential failures, ensuring that changes made during transactions are preserved.

Database replication utilizes this transaction log to create and maintain replicas of the original database. A replication process continuously monitors the transaction log for any committed changes. As new transactions commit, the replication process captures those changes from the transaction log and applies them to a backup database (slave).

By doing so, the backup database is kept up-to-date with the changes made in the primary database (master). This process allows the backup database to mirror the original database's state, creating a replica that is synchronized with the primary database's data.

In the event of a failure or downtime of the primary database, the replication process enables a seamless switch to the replica. As the replica has been continuously updated with the changes from the transaction log, it reflects the latest committed data, ensuring data availability and minimizing downtime.

This replication approach is particularly valuable for disaster recovery and high availability scenarios. By maintaining real-time or near-real-time replicas, organizations can ensure business continuity even in the face of hardware failures, software glitches, or other unexpected disruptions.

Asynchronous Replication Strategy: In many cases, replication is asynchronous, meaning there may be some delay between the time a transaction is committed on the master database and when it is applied to the slave databases. This approach helps reduce potential performance impact on the primary database. However, it also means that there might be a brief period when the slave databases are not fully up-to-date with the primary database.

Deferred Commit Strategy: In some databases, a commit can be deferred until the transaction has been successfully replicated to the slave database(s). This is known as synchronous or semi-synchronous replication. By deferring the commit until replication is complete, it ensures that the slave databases are fully synchronized with the primary database before acknowledging the transaction as committed. This provides stricter data consistency guarantees at the cost of potential performance overhead.

Practically the combination of asynchronous writing to data files and immediate writing to the transaction log, along with various replication strategies, allows for efficient and reliable data management in distributed systems, ensuring data integrity and availability even in the face of failures or replication delays.

7.4.2.2.2. Snapshot replication

Snapshot replication captures and overwrites a snapshot of data from the original database onto the receiving database. Unlike transactional replication, snapshot replication does not continuously monitor for data updates. Instead, it distributes data exactly as it appears at a specific moment in time. Snapshot replication is typically used when data changes happen infrequently, as it is less efficient in updating changes in real-time. While this type of replication may be slower than transactional replication, it is well-suited for scenarios where occasional data updates are acceptable, and real-time synchronization is not critical.

7.4.2.2.3. Merge Replication

Merge replication is the most complex type of SQL database replication. It involves the merging of data from multiple databases into a single receiving database. Unlike transactional and snapshot replication, merge replication allows for independent changes to be made by both the publisher (original database) and the subscribers (replicated databases). This model enables one publisher to send changes to multiple subscribers, making it suitable for server-to-client environments where bidirectional data synchronization is necessary. Merge replication provides the flexibility to accommodate multiple sources of data and ensures that all changes are merged seamlessly into the receiving database, maintaining data integrity across the distributed system.

7.4.2.3. Replication Strategy for NoSQL Database

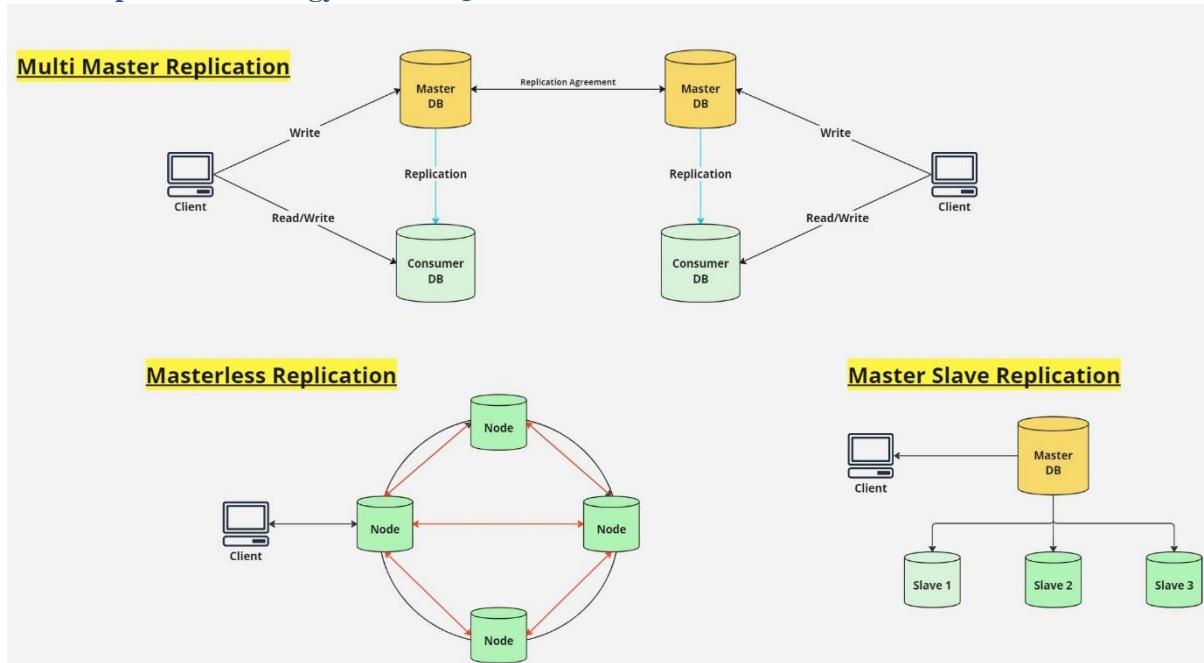


Figure 7-0-3 NoSQL Replication Strategies

Non-relational databases in the NoSQL category encompass a diverse range of data models, such as document-based, key-value, column-family, and graph databases. Each NoSQL data model requires distinct replication strategies to address its unique characteristics and to ensure data availability and fault tolerance.

Three primary replication models are commonly used: **multi-master, master-slave, and masterless** architectures.

In **multi-master** databases like DynamoDB, multiple nodes act as masters, allowing read and write operations on any node. This model offers high availability and distributed data updates. However, it introduces a point of failure, as when a master node goes down, electing a new master causes a brief downtime, potentially leading to SLA violations.

Master-slave architectures, exemplified by MongoDB, consist of a master node that handles all write operations and replicates data to multiple read-only slave nodes. While providing read scaling, this model still faces the risk of a single point of failure, where the master node going down disrupts write operations and necessitates the promotion of a new master.

To address this limitation, **masterless architectures**, like Scylla DB, have emerged. In these databases, data is replicated across multiple nodes, and all nodes are considered equal. This approach ensures that no single node can bring down the entire cluster, enhancing resilience and fault tolerance. In a typical masterless setup, each dataset is replicated across three or more replicas, distributing data efficiently and eliminating single points of failure.

By adopting a NoSQL database that employs a masterless architecture, applications gain an additional layer of resilience, particularly for high-volume and low-latency use cases. With multiple equal replicas, masterless databases offer uninterrupted operations even in the face of node failures, ensuring consistent data availability and minimizing downtime for critical applications.

7.4.2.4. Different forms of Data Replication

Data replication can take one of two forms, i.e. binary replication, and statement-based replication.

Binary Replication

Suppose application is inserting a document into database system, and after process completes it has a few bytes on disk that were written to contain some new data.

The way binary replication works is by examining the exact bytes that changed in the data files and recording those changes in a binary log. The secondary nodes then receive a copy of the binary log and write the specified data that changed to the exact byte locations that are specified on the binary log.

Pros:

- Replicating data is smooth on the secondary nodes since they get really specific instructions on what bytes to change and what to change them to.
- Secondary nodes aren't even aware of the statements that they're replicating.

Cons:

- Assumption is made that OS will be consistent across replica set. However, if one set is on Windows and other is on Linux, the same binary logs cannot function.
- In case of same OS, all the machines should have same instruction set.
- In case, data set is not updated on of the servers, it will result in corrupted data.

Statement Based Replication

After a write operation is completed on the primary node, the write statement itself is stored in a specific Log and the secondary nodes sync their logs with the primary node's log and replay any new statements on their own data.

Pros:

- This approach works regardless of the operating system or instruction set of the nodes in the replica set.
- Data consistency.
- No OS level or machine level dependency, hence valuable for any cross-platform solution that requires multiple OSs in same replica set.

Cons:

- Process of replication is slow in comparison with Binary replication.
- Statement based replication uses actual database commands to write in logs, hence, operation is bit heavier than Binary replication and workload is more.

7.4.3. Database Sharding

In literal terms, dividing larger parts into smaller parts is called '**sharding**'. In the world of distributed systems and databases, Sharding means splitting a single logical dataset and storing in multiple datasets.

Sharding is a booster when it comes to horizontal scaling of data. In a replica set, where we are having a master-slave architecture for fault tolerance (yes, we are referring to database replication), each server/node needs to contain the entire dataset. With time, as the dataset grows to a point, where it becomes tedious to serve clients, architects start thinking in terms of **scalability**.

One option could be to increase the capacity of individual machines so that they acquire more RAM or disk space or maybe even a powerful CPU. In short, we are talking about **Vertical Scaling**. However, this growth can be potentially costly operation over time and at some point, of time, it will meet a dead end. Especially, when it comes to cloud-based services, Vertical Scaling forever is not an option at all since there is limit to hardware configuration as well as storage facility.

Horizontal scaling comes to the rescue. Instead of scaling up our machines, more machines are added, and dataset is distributed among them. Here, entire dataset is not stored on one server. We can create as many shards of our dataset, which will eventually make up the formation of Sharded Cluster.

However, we still have one problem to solve. What if, one of the shards fail to serve data? Well, we can always create a replica set in one shard.

Basically, **each shard can be deployed as a replica set to ensure a level of fault tolerance**.



Figure 7-0-4 Sharding Example

7.4.3.1. Sharding Architecture Overview

Till now, we are clear about the part that we are distributing our data into multiple shards and maintaining a replica set in each shard. The main challenge would be to serve data to client application from multiple shards, as querying becomes trickier.

If client application is looking for a specific document, how to figure out where to look for the document in the sharded cluster?

Well, necessity is mother of all solutions, so architects introduced a '**router process**' between **Sharded Cluster and respective clients**, whose job is to accept queries from clients and figure out which shard should be processing the query. This router process maintains a **metadata** related to which shards are maintaining what data.

Now, this metadata itself can be big, hence, metadata is stored separately in **Config Servers**. To maintain, high availability of metadata for the router process, Config servers are also deployed as **Config replica set**.

To distribute data across shards, architects might use the strategy of using **a unique identifier (Shard key)** which consists of a field or fields that exist in every chunk of data across shards. There can be different sharding strategies to shard data across sharded cluster. Some common strategies are **Hashed Sharding, Ranged Sharding, etc.**

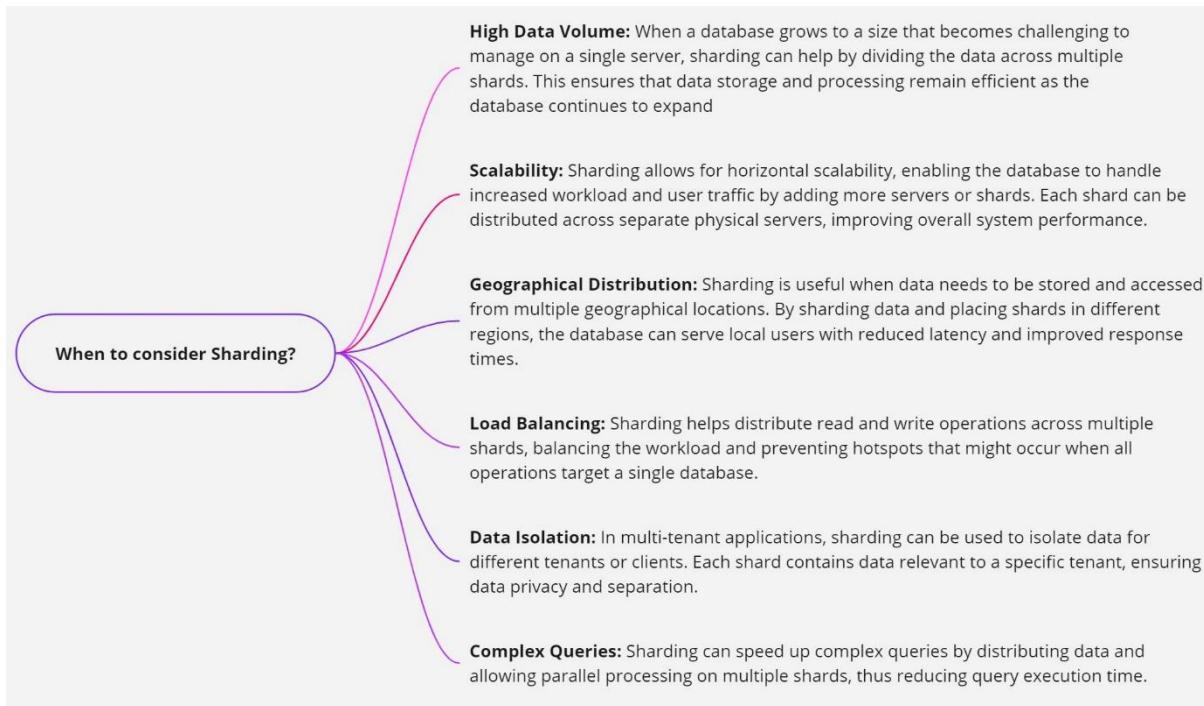
It is the job of Config Servers to maintain uniform distribution of data across shards. It is not an ideal scenario to have disproportionate data across shards in a cluster.

In cases where, we must serve data from multiple shards across a cluster, the router process will send the query to respective shards via Config Server and gather the results and merge them in a single result and serve it back to the client application.

7.4.3.2. When to Shard a DB?

It is not always a good idea to just scale your architecture vertically right from the word Go. A good architect must compute the requirements and look for indicators before deciding to split up the system.

Objects, Data & AI: Build thought process to develop Enterprise Applications



Let us go through that flow of decision making:

First check to do would be to estimate whether it is still economically viable to vertically scale our system. Vertical scaling would require scaling up one or more of the vertical resources like RAM, CPU, or disk space. And if vertical scaling of any of the resource would be less costly than adding up a new server altogether, then why not go for it.

However, in course of time, vertical scaling will reach to a deadline, after which, it would not be able to scale up any further.

Consider a scenario, where the current architecture is using 3 servers costing 200\$ each, working together in a master-slave pattern, with a total cost of 600\$. Now, to meet up the performance, we want to scale up to the next level of servers, which cost 800\$ each, however, performance increase would be just 2 times the previous set up, while total cost would be 2400\$.

Now, consider, setting up another replica set with the original server type costing 200\$ each. So, two replica set operating together will involve 6 servers, with each costing 200\$, which leads to the final cost estimation of 1200\$.

So, by splitting the system horizontally, we are getting the same performance at half the cost i.e. 1200\$ instead of scaling up, which is going to cost 2400\$.

Also, we need to consider, some scenarios like, while vertically scaling up, if we are increasing size of the disk, we might also need to increase the RAM to return back the same level of performance, which is a classic case of one expense leading to other.

There are use cases like Netflix, which stores data in different Geo-locations and have clients of respected Geo-locations attached to them. It uses the concept of **zone-sharding** to distribute data sets geographically.

7.4.4. Concurrency Control

In a distributed environment, multiple transactions may attempt to access and modify the same data concurrently. Without concurrency control, this could lead to data inconsistencies and violations of data integrity. Concurrency control mechanisms ensure that transactions are executed in a manner that maintains the consistency of the database, preventing conflicts and preserving the correctness of data.

Let us revisit [example](#) of 'G-Mart', an e-commerce application with respect to scenarios in DDB.

Imagine 'G-Mart', a large e-commerce platform operating as a Distributed Database System (DDB) to handle its vast customer base and product catalogue and global reach. The platform's architecture includes multiple database nodes spread across different data centres to serve customers worldwide and ensure high availability. The e-commerce platform decides to run a "Flash Sale" offering significant discounts on select products for a limited time. During the flash sale, the platform experiences a massive surge in user traffic as thousands of customers simultaneously browse the website, add items to their carts, and proceed to checkout.

With many customers adding the same discounted products to their carts, concurrency control ensures that the available inventory is accurately updated and that products are not overbooked or oversold. Concurrent updates to the inventory data must be synchronized to prevent multiple customers from purchasing the same item that is no longer available.

High traffic and an increased number of concurrent transactions can create a situation where multiple transactions contend for the same resources, leading to potential transactional deadlocks.

Suppose two customers simultaneously attempt to purchase the last item of a discounted product. Without proper concurrency control, both transactions may read the same inventory count, leading to lost updates. Concurrency control ensures that only one transaction can update the inventory count at a time, avoiding conflicts and preserving the correct inventory count.

Customers may simultaneously access their shopping carts, add, or remove items, and proceed to checkout. Concurrency control ensures that each customer's cart is isolated from others, providing a consistent view of their cart's content, and avoiding data inconsistencies.

Upon purchase of same product by concurrent customers, inventory updates need to be carried out in a coordinated manner, preventing any conflicts that might result in partial order fulfilment or incorrect shipping.

The distributed nature of the database introduces unique challenges in managing concurrency. The absence of a central authority controlling all transactions necessitates the adoption of distributed concurrency control mechanisms. Ensuring global consistency across distributed nodes becomes a paramount concern, as each node may hold partial or temporary updates during transaction execution.

There are various strategies for ensuring concurrency control in systems and they depend on factors such as the application requirements, data access patterns, and desired trade-offs between performance and data consistency.

Following section covers a short overview of some of the techniques used in concurrency control in distributed database systems. A detailed discussion on this topic seems to be out of scope, as focus is more on building the intuition of designing a robust system. Reader is encouraged to take up detailed research in interested areas of concurrency control.

7.4.4.1. Concurrency Control Techniques

- **Two-Phase Locking (2PL):** Two-Phase Locking is a widely used concurrency control technique that enforces strict serialization of transactions. It consists of two phases: the "growing" phase, where locks are acquired on data items before accessing them, and the "shrinking" phase, where locks are released after the transaction completes. This disciplined approach ensures serializability and prevents conflicts between concurrent transactions.
- **Timestamp Ordering:** Each transaction is assigned a unique timestamp that denotes its start time. Transactions are then executed in a strict order based on their timestamps. If two transactions have conflicting operations on the same data item, the transaction with the older timestamp can proceed, while the other is delayed. Timestamp ordering ensures conflict-free execution of transactions and guarantees serializability.
- **Optimistic Concurrency Control (OCC):** Transactions can proceed optimistically without acquiring locks initially. When a transaction attempts to commit, a conflict check is performed to ensure that it has not conflicted with other transactions. If conflicts are detected, the transaction is rolled back and re-executed. OCC provides higher concurrency but may require more transaction retries.
- **Serializable Snapshot Isolation (SSI):** SSI extends the concept of snapshot isolation to guarantee serializability in distributed databases. Transactions read from a consistent snapshot of the database, ensuring that their read values do not conflict with concurrent updates. SSI prevents various anomalies, including "write skew" and "phantom reads."
- **Distributed Deadlock Detection:** Distributed deadlock detection algorithms are employed to detect and resolve deadlocks that may occur in a distributed environment. These algorithms analyse the transaction dependency graph and proactively identify deadlocks, allowing the system to take appropriate actions to break the deadlock.
- **Multiple Granularity Locking:** This technique allows transactions to acquire locks at different levels of granularity, such as row-level, page-level, or table-level. This flexibility enables more fine-grained locking and reduces contention, improving concurrency.
- **Conservative Two-Phase Locking (CTPL):** CTPL is an enhancement to traditional Two-Phase Locking that allows transactions to acquire all the locks they may need at the beginning of the transaction. This approach reduces the risk of deadlock occurrence.
- **Concurrency Control in Replicated Databases:** In the context of replicated databases, additional considerations are necessary for concurrency control to ensure consistency across replicas. Techniques such as Quorum-based Replication and Conflict-free Replicated Data Types (CRDTs) are used to manage concurrent updates and maintain data consistency.

7.4.5. Transaction Management

In distributed databases, data is spread across multiple nodes, and transactions may need to access data located on different sites.

Continuing with [example](#) of 'G-Mart', an e-commerce application in section above, let us consider a scenario.

A customer located in one geographical region accesses the e-commerce platform and places an order for multiple products. The order details, such as product IDs, quantities, and customer information, are sent to the distributed database for processing.

Upon receiving the order, the e-commerce platform initiates a distributed transaction to handle the various components of order processing, which include:

Inventory Update: The transaction updates the inventory at multiple warehouse locations to reflect the reduction in stock for each purchased product. This update ensures that customers can only purchase products that are available in stock. The transaction identifies the warehouse locations where the selected products are stored and acquires locks on the corresponding inventory records to prevent conflicts with other transactions attempting to update the same inventory.

Payment Processing: The transaction involves payment processing to charge the customer's payment method for the total order amount. This step is critical, and the transaction must ensure that the payment is authorized and processed securely. Sensitive financial data, such as credit card information, is isolated and protected from unauthorized access.

Order Confirmation: Once the payment is successfully processed, the transaction confirms the order and generates an order confirmation for the customer. This step ensures that customers receive accurate and timely confirmation of their purchases. An order confirmation is generated for the customer, providing details of the purchase, estimated delivery time, and other relevant information.

Given the distributed nature of the e-commerce platform, there could be other concurrent transactions processing orders from different customers or performing inventory updates simultaneously. Concurrency control mechanisms, such as Two-Phase Locking (2PL) or Timestamp Ordering, are employed to ensure that transactions execute in a manner that maintains data consistency and isolation.

The distributed transaction ensures that all components of order processing are executed atomically. If any part of the transaction fails or encounters an error (e.g., inventory update failure or payment processing error), the entire transaction is rolled back. This ensures that the database is restored to its original state before the transaction began and prevents any partial or inconsistent updates.

Furthermore, once the transaction is committed successfully, the changes made to the database (e.g., inventory updates, payment records, order confirmation) are durably stored to withstand system failures. The distributed recovery mechanism ensures that committed changes survive crashes or errors.

Keeping Distributed Database systems ACID compliant is a very challenging and critical task as it ensures data consistency, integrity, and isolation of concurrent transactions executed across multiple nodes or sites of the distributed database system. Various distributed transaction models are employed to address these challenges and ensure the reliability and integrity of concurrent transactions. Let's discuss some of the key distributed transaction models briefly.

7.4.5.1. Flat Distributed Transactions:

In this model, a transaction spans multiple nodes or sites in the distributed database. The entire transaction, including all its sub-operations, is managed as a single, indivisible unit. The **distributed transaction manager** ensures that all nodes involved in the transaction follow a common protocol for coordination, concurrency control, and conflict resolution. All the participating nodes collaborate to guarantee the transaction's atomicity, consistency, isolation, and durability properties.

This model is simple to implement and well-suited for scenarios where the transaction's operations are distributed across different nodes. Centralized control over the entire transaction ensures that all parts of the transaction adhere to the same rules and constraints.

However, high coordination overhead, especially when involving many sites or nodes, can impact performance and scalability. Due to the involvement of multiple sites in a single transaction, there can be increased network communication and latency.

7.4.5.2. Nested Distributed Transactions:

The nested distributed transaction model involves a hierarchical structure of transactions. Each transaction consists of multiple sub-transactions executed at different sites. The sub-transactions are also referred to as "nested" transactions because they are logically nested within the main transaction. The main transaction coordinates the execution of its sub-transactions and manages their commit or rollback decisions. Coordination and communication occur between the sub-transactions and their parent transaction, ensuring that all operations are performed in a coordinated manner.

This model is suitable when the overall transaction can be divided into smaller, manageable units, and each sub-transaction may need to access different data sets at various sites. Improved modularity and separation of concerns, as different aspects of the overall transaction are handled by individual sub-transactions. Simplified failure handling and recovery at the level of sub-transactions, ensuring better fault tolerance.

However, this model faces challenges in managing the complexity of hierarchy of transactions and their interactions.

7.4.5.3. Atomic Commit Protocol:

The Atomic Commit Protocol is used to ensure that either all participating nodes in a distributed transaction commit the changes successfully or none of them do. The protocol follows a **two-phase commit (2PC) process, involving a coordinator and participating nodes**. It ensures that the coordinator communicates with all participating nodes to coordinate the commit or rollback decision.

- In **Phase 1 (Prepare Phase)**, the coordinator sends a "prepare to commit" request to all participating nodes, and each node responds with its vote (either "Yes" to commit or "No" to abort).
- In **Phase 2 (Commit Phase)**, based on the votes received, the coordinator decides whether to commit or abort the transaction. The decision is propagated to all nodes, and they perform the respective action accordingly.

The Atomic Commit Protocol ensures that all participating nodes reach a consensus on the commit decision, achieving global consistency in distributed transactions. However, the blocking nature of 2PC may lead to potential performance issues and contention, especially in scenarios with long transaction durations or high network latency.

7.4.6. Distributed Recovery

In the event of a system failure or partial failure (e.g., a node or site crash), the distributed recovery mechanism comes into play. The system uses undo and redo logs to recover the database to a consistent state. The transaction logs are crucial for undoing incomplete transactions and reapplying committed changes, ensuring that data integrity is maintained even after a failure.

There are two main types of failures that can occur in a distributed database:

- **Local failures** occur at a single site. This could be a failure of a database server, a network link, or a storage device.
- **Global failures** affect multiple sites. This could be a failure of the network, a power outage, or a natural disaster.

Distributed recovery must address both local and global failures. In the event of a local failure, the database at the affected site must be restored to a consistent state. In the event of a global failure, all the databases in the system must be restored to a consistent state.

7.4.6.1. Strategies for Distributed Recovery

Logging and Log-Based Recovery: Logging is a fundamental technique used in distributed recovery. Each transaction's operations are logged to a transaction log before they are executed. This log can be used to replay the changes that were made to the database after a failure. Log-based recovery involves two phases:

- **Undo Phase:** In case of a failure during the execution of a transaction, the system uses the transaction log to identify incomplete or uncommitted transactions. The incomplete transactions are "undone," and the changes made by these transactions are rolled back to maintain data consistency.
- **Redo Phase:** After the undo phase, the system uses the transaction log again to identify committed transactions that may not have been persisted to disk due to a crash. The changes made by these transactions are "redone" to restore the database to its most recent consistent state.

In distributed database systems, the transaction log may also be distributed across multiple nodes. Distributed logging ensures that the log records are stored redundantly in different locations, reducing the risk of losing critical information in the event of a node failure.

Checkpointing: It involves periodically saving the state of the database to reduce the time taken for recovery. Checkpoints are snapshots of the database state taken at specific intervals. If a failure occurs, the system can use the most recent checkpoint and the subsequent transaction log records for recovery, rather than going back to the beginning of the log.

3-Phase Commit Protocol: An extension of 2-Phase Commit protocol discussed above, adds a third phase, called the **dissemination phase**. The dissemination phase is used to ensure that all participants receive the commit message before they commit the transaction. This prevents the situation where a participant commits the transaction, but other participants do not, which could lead to inconsistencies in the database.

The main difference between 2-Phase commit and 3-Phase commit is that 3-Phase commit is more robust in the face of failures. If the coordinator fails in the prepare phase of 2-Phase commit, the participants will be left in a limbo state, where they do not know whether to commit or abort the transaction. However, if the coordinator fails in the prepare phase of 3-Phase commit, the participants will simply wait for the coordinator to recover. Once the coordinator recovers, it will send the commit message to all participants, and they will all commit the transaction.

In general, 3-Phase commit is more complex than 2-Phase commit, but it is also more robust. The choice of which protocol to use will depend on the specific requirements of the distributed system.

7.4.6.2. Distributed Deadlock Recovery

Distributed deadlock recovery is the process of detecting and resolving deadlocks in a distributed system. A **deadlock** occurs when two or more processes are blocked, waiting for each other to release the resources they need. This can lead to a system-wide stall, where no process can make progress.

Each participating node maintains information about the transactions and the resources they hold or are waiting for. Using this information, a **transaction dependency graph** is constructed, representing the relationships between transactions and the resources they need. The graph shows the circular dependencies that indicate potential deadlocks.

There are two main approaches to distributed deadlock recovery:

- **Detection-based recovery:** This approach involves detecting deadlocks and then resolving them. Deadlocks can be detected using a variety of algorithms, such as the Wait-for Graph (WFG) or the

Distributed Wait-for Graph (DWFG), which traverse the transaction dependency graph and identify cycles or loops that indicate deadlock situations. Once a deadlock is detected, it can be resolved by aborting one or more of the deadlocked processes.

- **Avoidance-based recovery:** This approach involves preventing deadlocks from occurring in the first place. There are a variety of deadlock avoidance algorithms, such as the banker's algorithm. Deadlock avoidance algorithms typically require that each process declare its resource requirements in advance.

Distributed deadlock recovery must be able to handle both local and global deadlocks. It can be difficult to detect deadlocks in distributed systems because the state of the system is distributed across multiple sites. Once a deadlock is detected, the distributed system needs to resolve it to break the circular dependency and allow transactions to proceed.

Deadlock resolution typically involves selecting one or more transactions to be aborted to free up the necessary resources and resolve the deadlock. Since distributed deadlock recovery involves multiple nodes, coordination is essential to ensure that deadlock resolution decisions are made consistently across all nodes. A **Distributed Deadlock Manager (DDM)** or Coordinator oversees the deadlock resolution process. It communicates with the participating nodes to identify and resolve deadlocks. To ensure a consistent resolution, a quorum-based decision-making approach is often used. A quorum is most participating nodes that must agree on the deadlock resolution decision. Once a quorum is reached, the decision is enforced across all nodes, ensuring consistent recovery.

After deadlock resolution, the distributed recovery mechanism ensures that the system restores data consistency. If a transaction was aborted as part of the deadlock resolution, the distributed recovery process will undo the changes made by the aborted transaction to bring the database back to a consistent state.

7.4.7. Distributed Query Management (DQP)

We have looked at [use case of an e-commerce application/platform](#) with respect to Distributed Database systems. To adapt with rapid growth and production, application leverages distributed environments to manage data and cater to growing demands of consumers spread across geo-locations. Let us have short recap of what happens behind the scene:

Data is partitioned and distributed across multiple nodes or sites to achieve better scalability and manageability. The platform divides its product data into fragments and stores them across different server clusters. This distribution improves data access and allows the system to handle the increasing volume of data efficiently. As it gains popularity, it experiences a surge in online traffic, especially during peak hours or special events like seasonal sales. Concurrent user queries to search for products, place orders, and retrieve customer information put a strain on the database infrastructure. To meet the increasing demands of a growing user base, the platform needs a database system that can scale horizontally to distribute the processing load across multiple nodes. Scalability ensures that the system can handle the increasing number of users and queries without sacrificing performance. As platform expands globally, it serves customers from different regions. The database system must cater to users from various geographic locations. This introduces the challenge of reducing query response times for users accessing the system from distant locations. Maintaining data consistency and integrity becomes critical in a distributed environment, where data is distributed and replicated across multiple nodes. Ensuring that all nodes hold consistent and up-to-date information is vital to avoid data inconsistencies and conflicts.

To present a unified and consistent view of data to its users, irrespective of the data's physical distribution across multiple nodes, system faces several challenges related to data distribution, scalability, performance, data consistency, and user experience.

Distributed Query Processing (DQP) addresses these challenges and ensures efficient and transparent query execution.

Distributed query processing (DQP) is the process of answering queries in a distributed database environment. This involves several steps for transforming a high-level query into an efficient query execution plan and opens various alternative ways for executing query operations of this plan.

7.4.7.1. Stages of DQP

In a distributed database system, query processing involves several stages to ensure efficient and optimized execution of queries across multiple sites.

Query Mapping: The first stage in distributed query processing is Query Mapping. In this stage, the user's input query, written in a query language, is formally specified. The query is then translated into an algebraic query on global relations, referring to the global conceptual schema. This translation process is similar to what is performed in a centralized DBMS and does not consider the actual distribution and replication of data.

During Query Mapping, the input query is normalized, checked for semantic errors, simplified, and restructured into an algebraic form. The goal is to convert the user's high-level query into a standardized form that can be efficiently processed by the distributed database system.

Localization: Once the query has been mapped to a global algebraic form, the next stage is Localization. In a distributed database, data is fragmented and stored in separate sites, with some fragments possibly being replicated across multiple sites. The Localization stage maps the global algebraic query to separate queries on individual data fragments using data distribution and replication information.

The Localization process ensures that the query is distributed to the relevant sites where the required data resides. This stage considers the actual distribution and replication scheme implemented in the distributed database, ensuring that the query is directed to the appropriate data fragments.

Global Query Optimization: In the Global Query Optimization stage, the distributed query is optimized to select the most efficient execution strategy from a list of candidate queries. The list of candidate queries is obtained by permuting the ordering of operations within the fragment queries generated during the Localization stage.

The optimization process aims to minimize the overall query processing cost, which includes factors like CPU cost, I/O costs, and communication costs over the network. Communication costs are particularly significant in distributed databases, especially when sites are connected through a wide area network (WAN).

Local Query Optimization: The Local Query Optimization stage is common to all sites in the distributed database. It involves applying optimization techniques similar to those used in centralized database systems. Each site optimizes its local query based on its local data and resources.

Coordination and Execution: The first three stages discussed above are typically performed at a central control site, often referred to as the query coordinator or optimizer. This central site is responsible for mapping, localizing, and globally optimizing the query. Once the query has been optimized, it is sent to the relevant sites for local optimization and execution.

Each site executes its part of the query using its local data, and the results are sent back to the query coordinator. The coordinator combines the results from all sites and presents the final query result to the user.

7.4.7.2. Challenges with DQP

There are several challenges that need to be addressed in distributed query processing, such as:

- **Data fragmentation:** The data in a distributed database is typically fragmented across multiple sites. This can make it difficult to identify the relevant data for a query.
- **Communication cost:** The communication cost between sites can be significant. This can impact the performance of the query execution plan.
- **Resource availability:** The resources at each site may not be available. This can make it difficult to execute the query execution plan.
- **Heterogeneity:** The different sites in a distributed system may have different hardware, software, and data formats. This can make it difficult to optimize the query plan and execute the query efficiently.
- **Failures:** The different sites in a distributed system can fail. This can lead to the failure of the query.

7.5. In-Memory Databases

Over the years, main-memory technology has significantly advanced, leading to larger and more cost-effective memory sizes. As main memory became cheaper, organizations started considering the possibility of storing entire databases in memory, leading to the concept of in-memory databases, where all data resided in the memory (RAM) of the host. The evolution of in-memory databases has been driven by the increasing affordability and availability of large main-memory sizes, as well as the need for faster data processing and reduced latency in modern data-driven applications.

7.5.1. Benefits

Storing a database entirely in memory eliminates the need for disk I/O operations for reading data, resulting in significantly faster data retrieval and processing. Since accessing data from memory is orders of magnitude faster than reading from disk, in-memory databases provide a substantial performance boost. It dramatically reduces data access latency, which is crucial for applications that require real-time or near-real-time responses. Queries and transactions executed against an in-memory database experience significantly lower response times, improving user experience and enabling more responsive applications. The ability to process data entirely in memory enables complex analytical queries and data manipulations to be performed at a much higher speed than traditional disk-based databases. This is particularly advantageous for data-intensive applications, big data analytics, and real-time processing.

In-memory databases are designed to leverage efficient in-memory data structures, such as hash tables and indexes, to optimize data access and manipulation. These data structures further improve performance and enable faster execution of various database operations.

However, there are certain challenges associated with in-memory databases.

7.5.2. Challenges

In-memory databases come with inherent risks due to their volatile nature and higher cost of memory compared to traditional disk storage. In-memory databases store all data in RAM, which is volatile memory. In the event of a server failure or unexpected shutdown, the data residing in memory is lost since it is not

persisted anywhere. This makes in-memory databases susceptible to data loss and requires careful consideration of backup and recovery strategies to mitigate this risk. To ensure data durability, in-memory databases need to implement mechanisms such as periodic snapshots, replication, or synchronous writes to disk, which can add complexity and performance overhead. Many in-memory databases offerings nowadays offer in-memory performance with persistence.

A very important point to be considered is that memory is more expensive than hard disks, making in-memory databases more costly to deploy and maintain. Since memory is more expensive and finite compared to disk storage, in-memory databases may face challenges in accommodating extremely large datasets. The cost of equipping servers with sufficient memory to accommodate large datasets can be a significant investment for organizations. Organizations may need to carefully manage data sizes and consider data partitioning or caching strategies to ensure optimal performance and avoid running out of memory.

7.5.3. Use Cases

In-memory databases offer unparalleled speed and responsiveness, making them well-suited for scenarios that demand real-time data processing, quick access, and high-performance computing. Let us look at some use cases, where in memory databases are leveraged by enterprise applications.

- **IoT Data:** Internet of Things (IoT) devices generate vast amounts of data that require rapid processing and analysis. In-memory databases can efficiently handle the high data ingestion rates and perform real-time computations on the data before storing it in a traditional database for long-term analysis.
- **E-commerce:** In-memory databases are well-suited for e-commerce applications, where certain data, such as user shopping carts or session information, needs to be quickly retrieved and updated during each user interaction. By storing this data in memory, e-commerce applications can offer a seamless and responsive shopping experience to users.
- **Gaming:** In-memory databases are highly beneficial for gaming applications, especially in scenarios like leader boards where rapid updates and real-time sorting of player scores are essential. In-memory databases ensure that leader boards are continuously up to date and readily accessible to all players, providing a smooth and interactive gaming experience.
- **Session Management:** In stateful web applications, session management requires quick access and updates to user-specific data, such as user preferences or recent actions. Storing session data in an in-memory database reduces the need for frequent round trips to the central database, improving overall application performance and responsiveness.
- **Caching and Performance Optimization:** In-memory databases are also used as caching layers to store frequently accessed data, reducing the need to query the primary database repeatedly. This caching mechanism accelerates data retrieval and alleviates the load on the primary database, enhancing overall system performance.

As the demand for high-performance and real-time data processing is increasing, several in memory database implementations have surfaced in the scene of enterprise development. Some are open source, and some are backed by enterprises based on specific use cases. Following are some of the well-known in memory databases being used today.

- **Redis:** Redis is an open-source, in-memory data structure store that supports various data structures such as strings, lists, sets, and hashes. It is known for its blazing-fast performance and versatility, making it popular for caching, session management, real-time analytics, and message queuing.

- **Memcached:** Memcached is a distributed, in-memory caching system that is often used to accelerate dynamic web applications. It is commonly used to cache frequently accessed data and reduce the load on the primary database, improving application responsiveness.
- **Apache Ignite:** Apache Ignite is an in-memory computing platform that provides data caching, compute grid, and data processing capabilities. It offers distributed in-memory data storage and supports SQL queries, key-value operations, and streaming data processing.
- **Microsoft SQL Server (In-Memory OLTP):** Microsoft SQL Server offers an in-memory OLTP (Online Transaction Processing) feature that allows users to store certain tables and procedures entirely in memory. This feature provides significant performance improvements for transaction-intensive workloads.
- **Aerospike:** Aerospike is a high-performance, distributed, and scalable in-memory NoSQL database. It is designed for real-time applications, such as real-time bidding, ad targeting, and recommendation engines.

7.5.4. In-Memory Databases in Distributed Architecture

In-memory databases have become an indispensable component of modern distributed architectures, transforming the landscape of data management and processing. These databases store data directly in memory, offering lightning-fast data access and retrieval times. In distributed systems, where nodes are dispersed across servers or data centres, such speed is critical for reducing latency and ensuring rapid response to user requests. In-memory databases also play a vital role as caching layers, storing frequently accessed data closer to the application or microservices. By doing so, they minimize the need for repetitive requests to the primary database, enhancing overall system performance and responsiveness, especially in read-heavy scenarios.

Additionally, many in memory databases support data replication and distribution, ensuring high availability and fault tolerance. In the event of node failures, data remains accessible from other replicated nodes, safeguarding data integrity and minimizing downtime. The scalability of in-memory databases is a remarkable feature, allowing easy horizontal expansion by adding more nodes to the cluster. This scalability ensures that the system can efficiently handle growing data volumes and user demands while balancing the load across nodes to optimize resource utilization.

In memory databases are ideal for real-time analytics, stream processing, and event-driven architectures. They enable rapid data ingestion, analysis, and immediate insights, which are essential for data-intensive applications in distributed environments. Furthermore, in-memory databases can serve as shared data stores for microservices, facilitating seamless communication between services and reducing the need for frequent API calls. This improves inter-service communication and simplifies the complexities of data exchange in distributed systems.

7.6. SUMMARY

In this chapter, we had an overview of what Database systems are and how they function. Let us look at the benefits we can reap via using a Database Management System:

- **Data Integrity and Consistency:** A DBMS enforces data integrity constraints, such as unique key constraints and referential integrity, to ensure the accuracy and consistency of data. It prevents data duplication, maintains data relationships, and enforces rules defined in the database schema, reducing the risk of data inconsistencies.
- **Data Security and Access Control:** DBMS provides mechanisms for user authentication, authorization, and access control. It enables administrators to define user roles, permissions, and

restrictions, ensuring that only authorized users can access and manipulate data. This helps protect sensitive information and maintain data security.

- **Data Sharing and Collaboration:** DBMS facilitates data sharing and collaboration among multiple users and applications. It supports concurrent access to the database, allowing multiple users to work simultaneously while maintaining data consistency. DBMS provides mechanisms like locks and transactions to manage concurrent access and prevent conflicts.
- **Data Centralization and Data Independence:** DBMS allows for centralizing data storage, making it easier to manage and maintain data. It provides a unified view of data, eliminating data redundancy and reducing data duplication. Additionally, DBMS offers data independence, enabling changes in the database schema without impacting the application programs using the data, promoting flexibility and scalability.
- **Data Query and Retrieval:** DBMS provides powerful query languages, such as SQL (Structured Query Language), allowing users to retrieve data based on specific criteria and perform complex queries. This simplifies data retrieval, analysis, and reporting tasks, enabling users to extract meaningful information from large datasets efficiently.
- **Data Consistency and Transaction Management:** DBMS ensures data consistency by providing transaction management capabilities. It ensures that groups of related operations (transactions) either complete successfully or fail together, maintaining data integrity. ACID properties (Atomicity, Consistency, Isolation, Durability) of transactions ensure reliability and recoverability of data.
- **Data Scalability and Performance Optimization:** DBMS supports scalability, allowing databases to handle growing amounts of data and increasing user demands. It provides optimization techniques, such as query optimization, indexing, and caching, to enhance query performance and improve overall system efficiency.
- **Data Backup and Recovery:** DBMS includes features for data backup and recovery. It enables regular backups of the database, ensuring that data can be restored in the event of system failures, errors, or disasters. This helps protect data and provides a means for recovering data to a consistent state.

We also studied about **Distributed Database Systems** and concepts related to it. Distributed database systems offer a few advantages over traditional centralized database systems:

- **Scalability:** Distributed database systems can be scaled to handle large amounts of data and users.
- **Availability:** Distributed database systems can be made highly available by replicating the data across multiple sites.
- **Performance:** Distributed database systems can be designed to improve performance by distributing the load across multiple sites.

However, distributed database systems also have some disadvantages, as follows:

- **Complexity:** Distributed database systems are more complex to design and manage than traditional centralized database systems.
- **Cost:** Distributed database systems can be more expensive to implement and maintain than traditional centralized database systems.
- **Security:** Distributed database systems can be more difficult to secure than traditional centralized database systems.

Distributed database systems are used in a variety of applications such as:

Objects, Data & AI: Build thought process to develop Enterprise Applications

- E-commerce: to store and manage product information, customer orders, and financial transactions in e-commerce applications.
- Telecom: to store and manage customer records, billing information, and network traffic data in telecom applications.
- Banking: to store and manage customer accounts, transactions, and financial data in banking applications.

In the coming chapter, we shall be discussed about storage structures of databases and Index in database systems in detail.

Data Storage

“Proper storage is about creating a home for something so that minimal effort is required to find it and put it away.” - Geralin Thomas

In the last chapter, we discussed Database Management System and its components on a higher level. In this chapter, let us discuss briefly about various mediums via which data is stored in databases internally and how efficiently we can access it. We will be first discussing about physical implementation of database systems.

8.1. Physical Storage Mediums

Computerized Databases are stores as files of records on a storage medium, which can further facilitate the retrieval, update, and processing of data as and when required. Data storage in computer systems comes in various forms, and each type of storage medium is classified based on several key factors, including the speed of data access, the cost per unit of data, and the reliability of the medium. These factors are crucial for determining which type of storage is best suited for a particular use case or application.

Faster access times are essential for applications that require quick retrieval and processing of data. Whereas the cost per unit of data is an important consideration, especially for organizations that need to manage large volumes of data. The reliability of a storage medium is crucial to ensure data integrity and availability. Data volatility is important consideration as we need to safeguard data in times of system failures. Based upon these factors, computerized storage mediums are divided into three main categories as show below:

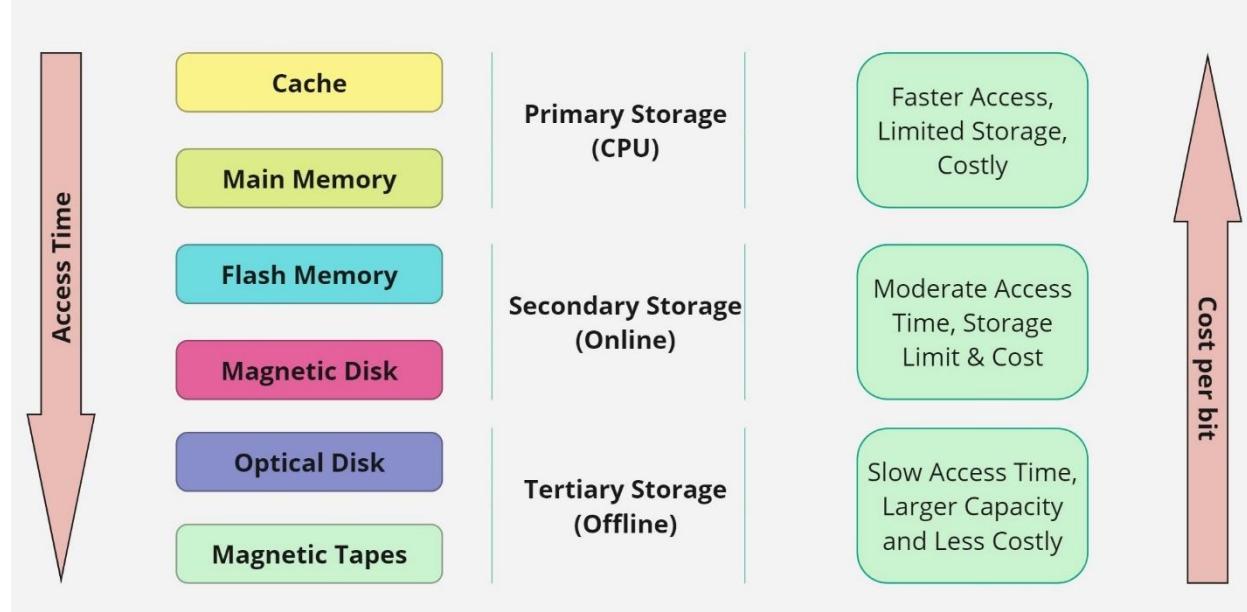


Figure 8-0-1 Physical Storage Mediums

8.1.1. Primary Storage

Imagine primary storage as the fast and small desk right in front of us while we work. This desk (RAM/Cache) allows us to quickly access the things we are actively using, like the current page of a

book. **Cache** memory is relatively small and is managed by CPU to speed up execution of program instructions using techniques such as prefetching and pipelining.

The data to be operated on is stored in **main memory or DRAM (dynamic RAM)**, which provides the main work area for the CPU for keeping program instructions and data. It may contain tens of gigabytes of data, even enterprise databases can be entirely stored here, however, it is volatile, meaning in case of power failure or system crash, data will be gone!

8.1.2. Secondary Storage

We can think of secondary storage as a bookshelf or filing cabinet. It is not as fast as primary storage, yet it can store a lot more items, like all the books or files. This is where our important documents, pictures, and software are kept. Even if the computer turns off, the data on secondary storage stays safe. Magnetic disks (hard drives) are like bookshelves, and flash memory (like USB drives) is like a digital filing cabinet. Magnetic disks provide the bulk of secondary storage for modern computer systems.

Flash memory have been widely used a medium of data storage in phones, camera, etc. Even in personal computers, flash memory is replacing magnetic disks and is referred as solid-state disk (SSD). It uses flash memory internally to store data but provides an interface similar to a magnetic disk, allowing data to be stored or retrieved in units of a block.

8.1.3. Tertiary Storage

Tertiary storage is like an attic or basement. This is where we keep things we don't need frequently yet need to store for a really long time. Imagine it using as a store to keep old family photo albums, holiday decorations, and other stuff we don't access regularly. In the computer world, this is analogues to things like CDs, DVDs, and magnetic tapes. They can store a lot of data and don't cost too much, however they come with the trade-off of access time.

To access data stored on magnetic disk, the system must first move the data from disk to main memory (personal computer), from where they can be accessed. After the system has performed the designated operations, the data that have been modified must be written to disk.

Optical storage options like CD/DVDs are capable of storing any type of digital data, including backups of database contents, however, they are not suitable for storing active database data since the time required to access a given piece of data can be quite long compared to the time taken by a magnetic disk.

So, in a computer, primary storage is like one's working desk, secondary storage is like a bookshelf or filing cabinet, and tertiary storage is like your attic or basement for things you don't use often but still want to keep. Each has its own role in managing and accessing data with their own trade-offs. In general, Magnetic tapes are used as a storage medium for backing up databases because storage on tape costs much less than storage on disk. Their capacities have been growing steadily in recent years; however, the storage requirements of large enterprise applications have outgrown rate of disk capacities.

Alternatively, in recent years, SSD storage sizes have been growing rapidly, and their cost has come down significantly, which has made them a main competitor to magnetic disks due to their superior performance. It has become the preferred choice for enterprise data and are being used as an intermediate layer between main memory and secondary rotating storage in the form of magnetic disks.

8.1.4. RAID

In fact, the data-storage requirements of some applications such as multimedia applications have grown so fast that large number of disks are required to store the data. And this is where a relatively newer technology called **RAID** is being leveraged by organizations for improving the rate at which data can be read or written.

RAID originally stood for **redundant arrays of inexpensive disks**, however, in recent years, the **I** in **RAID** is said to stand for **independent**. The idea is to improve performance and reliability and even out the widely different rates of performance improvement of disks against those in memory and microprocessors. An argument can be made that RAM capacities have quadrupled every two to three years but in contrast disk access time and transfer rate have not made significant improvement.



Figure 8-0-2 Bit Level Stripping

RAID relies heavily of the technique of parallelization where a large array of small independent disks, acting as a single higher performance logical disk, are operated in parallel. Concept of **data-stripping** is used to distribute data transparently over multiple disks to make them appear as a single large, fast disk. Each disk in the array holds a portion of each data block. This allows for parallel read and write operations, improving overall data transfer rates and I/O performance. This also provides a window to improve the reliability parameter as redundant information can be stored on multiple disks and failure of one disk does not account for loss of data or distribute the data uniformly across all disks resulting in better load balancing. When data is read, it can be retrieved from the disk with shorter queuing, seek, and rotational delays. If a disk fails, the other disk is used until the first is repaired.

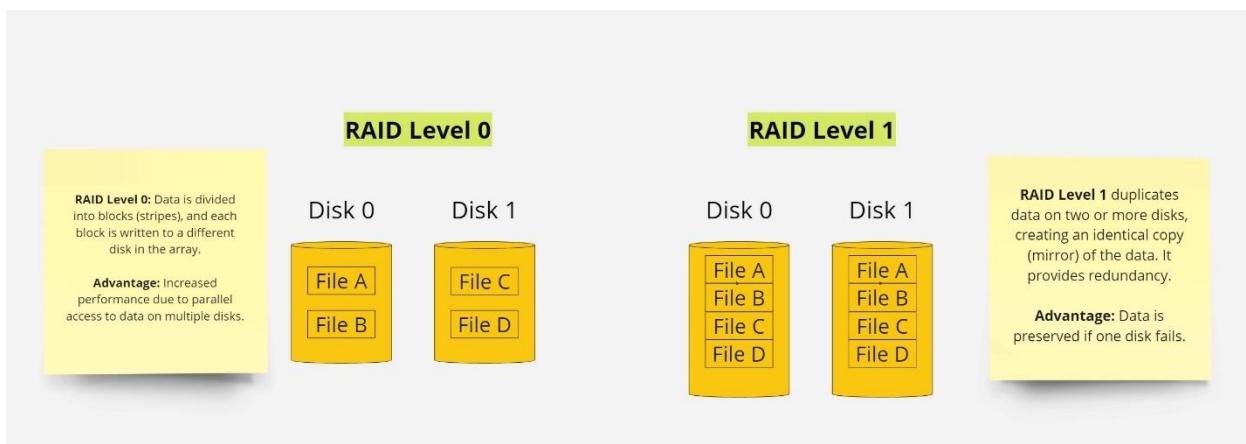


Figure 8-0-3 RAID Level 0 & 1

There are different RAID levels that use data striping, and each has its own approach to striping data across disks. Diagram above shows RAID Level 0 and 1. RAID Level 5 stripes data at the block level across multiple disks and includes distributed parity information for fault tolerance across all the disks. It gives a good balance between performance and redundancy. Each disk in the array stores both data and parity information.

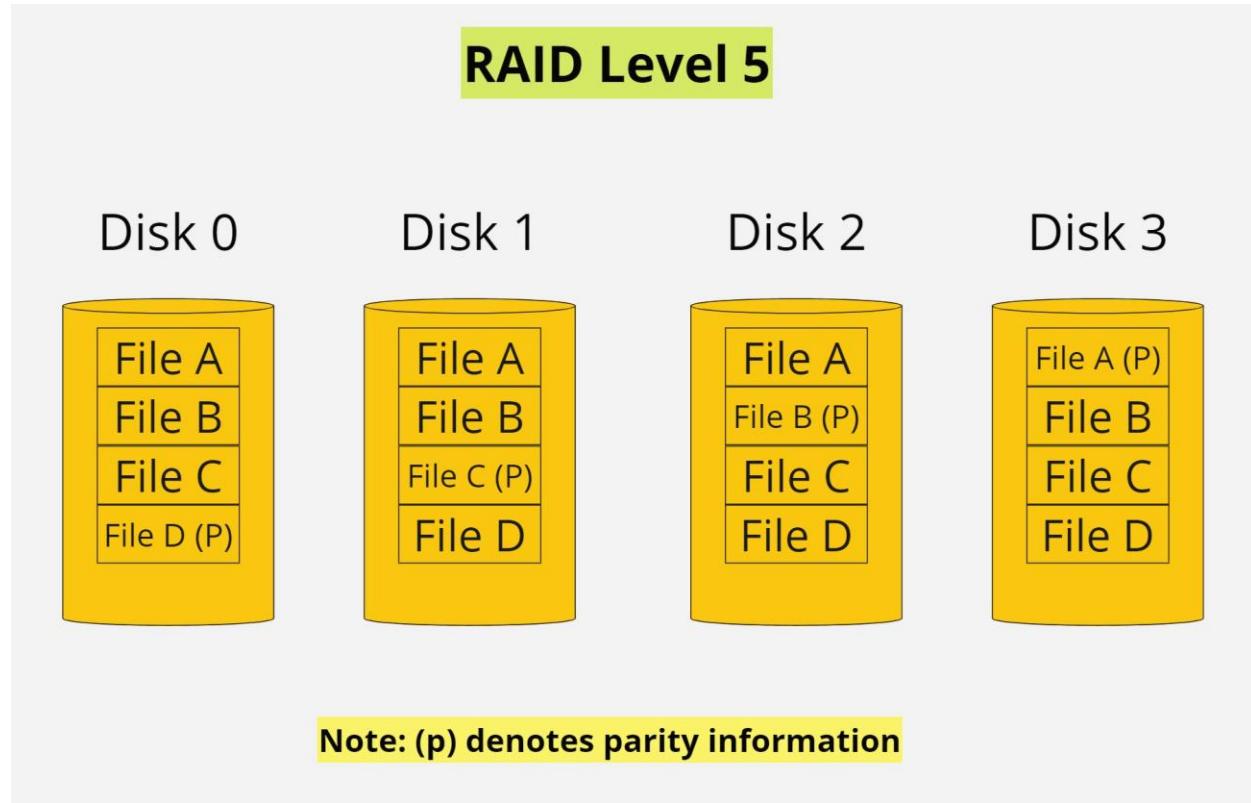


Figure 8-0-4 RAID Level 5

In the context of RAID, parity is a calculated value that represents the XOR (exclusive OR) result of a specific set of data bits. The purpose of parity is to detect errors and, in some RAID levels, to reconstruct data in case of disk failure. The result is stored as the parity information on a designated drive or distributed across all drives, depending on the RAID level.

RAID 5 can tolerate the failure of a single drive without losing data. If one drive fails, the missing data can be reconstructed using the parity information from the remaining drives. It offers good read performance because data can be read in parallel from multiple drives. However, write performance can be slower due to the need to calculate and update parity information for each write operation. RAID 5 is more space-efficient than mirroring-based RAID levels (e.g., RAID 1) because it does not require a complete duplication of data. It is commonly used in scenarios where a balance of performance and data protection is required, such as file servers, application servers, and database servers. However, it is not recommended for high-write environments or with very large capacity drives, as the time to rebuild the array after a drive failure increases with drive size.

Originally, people used RAID because it was cheaper to buy lots of small drives than one big one. But today, technology has changed, and larger drives are cheaper per unit of data storage. So now, we use RAID

for speed and safety, not just because it is cheaper. Managing a bunch of smaller drives working together is often easier than dealing with a single large drive. It is like managing a team of people who can help each other out. In simple terms, RAID is like a team of smaller hard drives working together to store your data faster, keep it safe, and make it easier to manage.

8.1.5. Storage Area Networks

With the rapid growth of online businesses, streaming platforms, and complex software systems that manage all sorts of information, the need for storing data has shot up. It is like needing more and more shelves to store all our stuff because we have got so much data stacked. Storing and managing all this data is getting really expensive. In some cases, it is even more expensive to manage the data storage part than the actual computers that use the data. Many organizations use RAID systems to store data efficiently, but they face a problem: these systems are tied to specific servers and can't easily be shared. It is like having a bookshelf that is permanently attached to one room; we can't move it around to where we need it.

To solve these issues, organizations have adopted a concept called a **Storage Area Network (SAN)**. Think of it like a super-fast network where entire data storage is connected. It is like having books on wheel trolley and connected to a high-speed track so we can easily move them to the room where we need them.

In today's world, where the internet plays a massive role in how organizations operate, there is a need to shift away from rigid and fixed data centres to a more adaptable and flexible infrastructure. This change is necessary because the cost of handling all the data is increasing so quickly that, in many cases, it is actually more expensive to manage the storage of data connected to servers than it is to buy the servers themselves.

Many companies have become providers of Storage Area Networks (SANs) and they have their own unique ways of setting up these networks. In fact, companies offer their own versions of Storage Area Networks (SANs) with each offering having its unique way of setting up these networks. These SANs are designed to put data storage systems far away from the servers. This flexibility means we can set up our systems in more diverse ways. If we already have applications that manage our data storage, we can use these in the SAN setup. They can work with the SAN using a special kind of network called **Fibre Channel**. This network makes the old way of connecting storage devices (using SCSI) compatible with the new SAN setup. So, our SAN-connected devices can act like the familiar SCSI devices.

Alternatively, we can also use a Fibre Channel switch to connect multiple storage systems (like RAID setups and tape libraries) to your servers. It is similar to having a central hub that connects various stores to our house. Another option is using Fibre Channel hubs and switches to connect servers and storage systems in different arrangements. It is like having a more complex road system with multiple intersections and pathways. Idea is to start with simpler setups and gradually make them more complex by adding more servers and storage devices as needed.

The benefits of SAN set up is centred around flexibility. We can connect many servers and storage devices in various ways using hubs and switches. It is like having a versatile road network that connects different places. We can place a server up to 10 kilometres away from a storage system using the right kind of fibre optic cables. New devices and servers can be added without causing disruptions. Data can be quickly copied to multiple storage systems. This is helpful for making sure our data is always available and for keeping backups in case of disasters. It is like making instant copies of our important documents both at home and in a safe deposit box at the bank.

Although, Storage Area Networks (SANs) are becoming more and more popular, they come with their fair share of challenges centered around compatibility. If we have storage equipment from various companies, making them work together in a single SAN can be tricky. The standards for both the software and hardware

used in storage are constantly evolving. So, keeping our SAN up to date and compatible with the latest technology can be a bit of a headache.

Despite these challenges, many big companies are looking at SANs as a viable option for storing their important databases. They see the potential benefits, even though they have to deal with these issues.

8.1.6. Network-Attached Storage

Network Attached Storage (NAS) is a way to share and manage data over a network, but it appears more like a networked file system. The NAS devices are like super-sized hard drives that you can connect to a network. They are helpful because they can provide a lot of storage space to multiple computers without making them stop working for maintenance or upgrades. This means NAS provides a way to access files and folders, much like how we would use our computer's local storage. NAS uses networked file system protocols, such as NFS (Network File System) or CIFS (Common Internet File System), to make files and folders available over the network.

We can place NAS devices anywhere on your local network (like in different rooms of your house) and use them to store all sorts of data. We can even group them together in various ways. Each NAS system has a central box (like a small computer) that connects to the network. This box acts as the go-between for the NAS system and the computers on the network. We don't need a monitor, keyboard, or mouse to use it. We can even add one or more hard drives or tape drives to many NAS systems to increase the total storage space. These protocols allow us to access and manage our data as if it were on a regular computer's hard drive, rather than treating it like a big shared disk.

NASs act as an alternative to SANs, as the latter focus more on block-level data access and often appear as large disks that store data but don't necessarily provide the same kind of file management and sharing features that NAS does. Network Attached Storage (NAS) systems aim to be reliable and easy to manage. They come with features like secure login and the ability to send email alerts if something goes wrong. These devices are designed to be scalable, dependable, flexible, and fast. NAS systems are designed to work with a wide range of operating systems (like Windows, UNIX, or NetWare) without needing specific changes on the client side. SANs may require more specific configurations on the client's end. SANs usually create their own private network, often called a LAN, where all the storage devices and servers connect. In contrast, NAS devices are directly connected to the existing public network. In simple terms, NAS is like a user-friendly and versatile file storage system that is easy to manage and works with different devices and operating systems (more compatibility than SANs, reference to drawback of SANs).

8.1.7. Object-Based Storage

In recent years, there have been significant changes in how data storage works, driven by the rapid growth of cloud computing, distributed database and analytics systems, and data-intensive web applications. These changes have led to a transformation in the way enterprise storage infrastructure is designed.

Traditionally, storage systems used to be hardware-centric and focused on managing files stored in blocks. However, the latest trend in storage architecture is known as **object-based storage**. In object-based storage, data is organized as objects, not files made up of blocks. These objects come with metadata, which contains important information for managing them. Each object is uniquely identified with a global ID, making it easy to find and access.

The idea of object storage started with research projects at institutions like Carnegie Mellon University (CMU) and the Oceanstore system at the University of California, Berkeley. These projects aimed to create a global infrastructure that allows continuous access to data from various trusted and untrusted servers.

With object storage, we don't need to worry about lower-level storage tasks like managing storage capacity or deciding which RAID architecture to use for data protection. Object storage simplifies data management by treating data as self-contained objects, making it easier to organize and access large volumes of information.

Object storage provides a high level of flexibility by allowing applications to directly control objects and making objects easily accessible across a wide network. It supports replication and distribution of these objects. Object storage is particularly well-suited for handling large amounts of unstructured data, such as web pages, images, audio/video files, and other types of data that don't neatly fit into traditional file structures.

Object-based storage was introduced as part of the SCSI protocol but didn't become a commercial product until Seagate adopted it in its Kinetic Open Storage Platform. Today, it is widely used by technology giants like Google, Facebook, Spotify, and Dropbox for storing vast amounts of data. Many cloud services, including Google Cloud, Amazon AWS, and Microsoft Azure, rely on object storage to store files, relations, and messages as objects.

Some popular object storage products include Hitachi's HCP, EMC's Atmos, and Scality's RING. OpenStack Swift is an open-source project that simplifies object storage by allowing users to retrieve and store objects using simple HTTP GET and PUT requests. It is cost-effective, fault-resistant, takes advantage of geographic redundancy, and scales well for managing large numbers of objects.

However, object storage may not be the best choice for transaction-intensive systems that require concurrent processing. So, it might not be considered suitable for mainstream enterprise-level database applications where high-throughput transaction processing is crucial.

8.2. Storage Structures

8.2.1. File Storage

Persistent data, which are data that remain even when the computer is turned off, are typically stored on non-volatile storage devices such as magnetic disks or solid-state drives (SSDs). These storage devices are organized into blocks, meaning data is read from or written to them in fixed-size units called blocks.

Databases, on the other hand, deal with records. Records are usually much smaller than these blocks, although they may contain attributes that are quite large in some cases. To manage this difference in size, most databases use the operating system's files as an intermediate layer for storing records. These files abstract away some of the underlying details related to blocks. Each file is structured in a way that it is like a sequence of records. These records are then mapped onto blocks of data on the disk.

However, even though databases use these files to store records, they still need to be aware of the block structure for efficient data access and to support recovery from system failures. So, it is crucial for databases to understand how individual records are stored in these files while taking the block structure into account. This ensures that data can be efficiently managed, accessed, and safeguarded, even when the underlying storage operates in blocks.

Now, files are a fundamental concept in operating systems. They serve as containers to store data. So, we assume that there is a file system in place that takes care of managing these files on the storage devices. It is important to figure out how to represent the logical data models (the way data is organized conceptually) in terms of these files. Essentially, it is about translating how the database is supposed to work into how the data is actually stored on the disk through files. This translation is a critical aspect of database management to ensure data is organized efficiently and can be accessed as needed.

Each file is further divided into fixed-size storage units called “blocks”. These blocks are the fundamental units for allocating storage space and transferring data. Most databases default to block sizes of 4 to 8 kilobytes, although some databases allow you to specify the block size when creating a database instance. In certain cases, larger block sizes can be advantageous for specific database applications.

Now, each block can contain multiple records, but which records are stored within a block depends on the specific way the data is physically organized in the database. The choice of how data is organized, and which records are placed in a block is a fundamental aspect of database design and directly impacts how efficiently data can be accessed and managed.

Let us look at the structure of a magnetic disk:

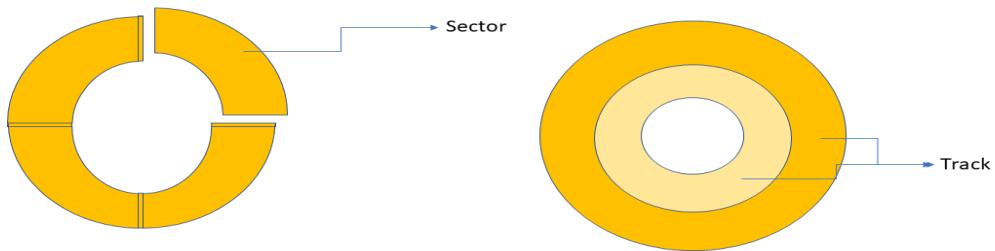


Figure 8-0-5 Magnetic Disk Structure

A disk surface is logically divided into concentric circles called **tracks**, and each track is further subdivided into **sectors**. When data is read from or written to the disk, it's done at the sector level. The division of a track into sectors is hard-coded on the disk surface and cannot be changed. Sectors are typically fixed in size, commonly 512 bytes or 4 KB.

When a disk is formatted or initialized, the operating system sets up the structure of the disk, including the division of the track into equal-sized portions called **disk blocks or pages**. These blocks are predetermined in size and remain fixed once the initialization process is completed. The size of these blocks cannot be changed dynamically. Typically, a disk with hard-coded sectors (the smallest addressable unit on a disk) organizes these sectors into larger units called blocks during initialization. The organization involves either combining multiple sectors into a single block or subdividing larger sectors into smaller blocks based on the predefined block size. Data is transferred between disk and main memory in units of blocks.

Each block is separated by interblock gaps, which are spaces reserved between blocks on the track. These gaps contain specialized control information that is written during the disk initialization process. This control information includes specific codes or markers that identify the beginning and end of each block.

This block size is significant for various file systems and disk management operations, as it determines how data is organized, stored, and retrieved on the disk. Larger block sizes can be more efficient for handling larger files and reducing overhead, while smaller block sizes might be advantageous for storing smaller files or optimizing space usage. However, there's a trade-off between efficiency and wasted space, as

smaller block sizes can lead to increased internal fragmentation (wasted space within blocks). Changing this size later would involve reformatting the disk, which is a process that erases all existing data, making it a critical decision during the initial setup of the disk.

Now, let us consider a sample table to understand how data is organized on the disk in the form of database:

Employee ID	Name	Department	Section	Address
E101	Rahul	D-201	A	<.....>
E102	Shardul	D-301	B	<.....>
.....
E200	Naresh	D-202	D	<.....>

Now, let us consider the column wise size break up:

Employee Id	10 bytes
Name	50 bytes
Department	10 bytes
Section	8 bytes
Address	50 bytes
Total	128 bytes

This means that each row of the table is of 128 bytes. Say, we have a block size of 512 bytes (default), so we can calculate, how many records a block of disk can store:

$$\text{Records per block} = (512/128) = 4;$$

Hence, we can deduce that to store 100 such records, we will utilize 25 blocks of the disk. In other words, we can say, to access entire table, we will have to access 25 blocks.

Now in a relational database, we have tables, and each table consists of rows, which are called tuples, and these tuples often have different sizes because they represent different sets of attributes or fields.

When it comes to storing this relational data on a physical storage device like a disk, there are following two options:

Fixed-Length Records in Separate Files: In fixed-length records, each record (or row) in a database table is allocated a specific, fixed amount of storage space, regardless of the actual amount of data it contains. This ensures that each record takes up the same amount of space in the storage file.

Consider a database of customer information. In a fixed-length record format, each customer's record is, let's say, exactly 200 bytes long. Even if one customer's data only requires 100 bytes (e.g., name and address), their record would still occupy 200 bytes in the file. This approach makes it easy to locate and retrieve records because we always know that, for example, the first customer record starts at byte 0, the second one at byte 200, and so on.

This approach is relatively simple to implement because we always know where a specific record starts and ends in the file. However, it can lead to wasted space if some records are much smaller than the allocated space.

Variable-Length Records in the Same File: An alternative is to structure your files in a way that can accommodate records of different lengths within the same file. This way, you don't allocate a fixed amount

of space for each record. Each record contains its own metadata or markers that indicate its start and end, so the system can distinguish one record from another.

Let's stick with the customer database. In a variable-length record format, one customer's data might occupy 100 bytes, and the next customer's data could take up 150 bytes. The records themselves include markers indicating where each record starts and ends. This approach makes better use of storage space as there's no wasted, empty space between records.

This is more efficient in terms of storage utilization because it doesn't waste space and can accommodate varying record sizes, making it more adaptable, but it can be more challenging to implement because we need to manage variable-length records and keep track of where each record starts and ends in the file. Retrieving data can be slower compared to fixed-length records because the system must locate the start and end of each record before processing it.

Storing Large Objects: Databases often need to store data that can be much larger than the typical storage block on a disk. For example, multimedia data like images, audio recordings, or video files can be quite large, ranging from megabytes to gigabytes in size. In SQL databases, you have data types like BLOB (Binary Large Object) and CLOB (Character Large Object) specifically designed to handle such large data.

To manage these large objects, many databases have an internal restriction where the size of a record is limited to be no larger than the size of a block on the storage device. In other words, a record in the database is usually limited to a certain size, and this size restriction can be smaller than the size of large objects. In such cases, databases logically include large objects within records but store these large objects separately from the other attributes of the record.

To make this work, the database stores a reference or pointer to the large object within the record. This pointer essentially tells the database where to find the actual large object's data. The large objects can be managed in two main ways:

- **File System Storage:** In this approach, large objects are stored as separate files in a file system area that is managed by the database. The database record contains a reference to the location of the large object in the file system.
- **Database-Managed Storage:** In this method, the database itself manages the storage of large objects. Instead of storing them as external files, the database stores large objects as internal structures. The database can use storage mechanisms like B+ tree file organizations to efficiently access different parts of the large object. This means you can efficiently read the entire large object or specific portions of it, as well as insert or delete parts of the object. For example, consider a video file stored as a large object in a database. The record might contain information about the video, while the video data itself is stored separately. The record would include a pointer to the location of the video data. With the use of B+ tree file organization, the database can efficiently access and manage this video data. This approach helps databases handle very large data objects without overwhelming the size of each individual record in the database.

However, accessing large objects directly through database interfaces might not be as efficient as accessing them from a file system. Database systems are optimized for structured data and complex queries, while handling large, unstructured data like multimedia can be less efficient. Databases are often backed up periodically. These backups are known as **database dumps**. When we store large objects in the database, it can significantly increase the size of these database dumps, making them larger and potentially slower to create and restore.

Because of these concerns, many applications opt to store very large objects, such as video data, outside of the database, typically in a file system. In this approach, the application stores a reference to the file, usually in the form of a file path, as an attribute of a record in the database. This way, the database record contains a reference to the location of the file in the file system.

Storing data outside of the database in a file system can lead to potential issues with data integrity. For example, if a file is deleted or moved in the file system, the reference stored in the database might become invalid, resulting in a form of foreign-key constraint violation. This can lead to data inconsistencies. Database systems typically have robust security and authorization controls in place to manage access to data. When data is stored in the file system, these controls are not directly applicable, and access to the files might not be as secure or easily managed.

We need to be aware about the issues when storing data in this fashion as it requires careful handling to ensure data integrity and security.

8.2.1.1. Record Organization in Files

So far, we have seen how records are represented in file structure in databases. Each record represents an entity or an item, and a collection of records forms a relation, which is essentially a table in a database. In this section, we will discuss about how to organize these records within a file in the database system. This involves deciding on the physical structure and layout of the data on the storage device.

The organization of records in a file is a critical decision in database design, and it impacts the efficiency and performance of data access operations. Different file organization methods can be employed based on the specific needs and characteristics of the data, as well as the anticipated access patterns. Understanding how the data will be accessed is crucial. Are we going to retrieve records in a specific order or based on certain criteria, or will access be random? Size and nature of the data needs to be considered as well. Whether record size is fixed, or does it vary? Are we dealing with large objects like multimedia files? We also need to think about query patterns to be run on the dataset. While we do need to ensure that data organization ensures efficient query run and at the same time adheres to data integrity and consistency while maintaining optimal space utilization.

There are certain file organization methods which are used across industry, but they come with their advantages and trade-offs. The choice of file organization method should align with the specific requirements and characteristics of the database and the expected usage patterns. Let us discuss common file organization methods:

8.2.1.1.1. Heap file organization

It is the simplest file management technique where records are placed in the file without any particular order. There is no inherent logic to how they are stored. Imagine tossing items into a drawer randomly. There is no specific order or arrangement. Each item can be anywhere in the drawer. Similarly, in Heap, records are put at the end of the file as they are added. The records are not sorted or ordered in any way. The next record is saved in the new block after the data block is filled. This new block does not have to be the next one.

Heap organization works well for smaller databases or in scenarios where significant amount of data needs to be inserted in the database at once. However, it does not suit a larger database as record access time will take a toll.

8.2.1.1.2. Sequential File organization

In this technique, ordering of records is maintained at the time of insertion. The record will be entered in the same order as it is inserted into the tables. For deletion and update operations, memory blocks are identified for the records in question and are marked for deletion. A new record is added in their place.

It is a simple technique data is stored with comparatively little effort and avoids usage of expensive storage mechanisms. However, accessing a specific record at once is not possible in this organization; instead, we must proceed in a sequential manner, which is a costly operation.

8.2.1.1.3. Multitable Clustering Organization

In multitable clustering, records from different tables are stored together in the same file or block. This is done to make certain operations, like joining data from different tables, more efficient. Imagine a big bookshelf where books are sorted not only by title but also by genre. All science fiction books are in one section, all mysteries in another, and so on. In a typical relational database, data is organized into tables, with each table representing a specific entity or concept (e.g., customers, orders, products). Each table is usually stored in its own file or block. However, in multitable clustering, records from different tables are stored together in the same file or block.

For example, we have a sample employee table and department table below:

EMP-ID	EMP Name	Address	Dep-ID
E-1	John	London	D-2
E-2	Rahul	India	D-3
E-3	Mark	US	D-2

Dep-ID	Dep Name
D1	IT
D2	HR
D3	Management

A resultant clustered table would look something like:

Dep-ID	Dep-Name	EMP-ID	EMP Name	Address
D2	HR	E1	John	London
D3	Management	E2	Rahul	India
D2	HR	E3	Mark	US

In the above table, the department information is clustered with the corresponding employee records. This organization lowers the cost of searching multiple files for various records. This can significantly reduce the number of disk input/output (I/O) operations, which is a performance bottleneck in many database systems, since the relevant data is collocated in the same file or block, potentially reducing the need for extensive disk I/O during join operations. A cluster key is used to link the tables together. The primary goal of multitable clustering is to co-locate data that is often joined together in queries. For example, if we frequently need to retrieve data from both the “Employee” and “Department” tables, storing them together can make the join operation more efficient.

Note that in a real-world scenario, additional considerations such as indexing, and data distribution would be taken into account for optimal performance. However, this approach has its own limitations. It does not

perform well for larger databases, as in scenarios like change in join condition, it will not give appropriate result. In case of update, traversal of files will take time.

8.2.1.1.4. B+ Tree Organization

B+ tree structure is an advanced way of indexed sequential mechanism for storing and managing data in a structured way within the database. We will be discussing about index in detail in next section, and indexes are the place where B+ tree structures are utilized the most. B+-Trees are designed to remain balanced, meaning they have roughly the same number of branches at each level. This balance is maintained during insertions and deletions. Balancing ensures that searches remain efficient, as the system can quickly determine the path to the data you want to access. It is particularly useful when you need efficient and ordered access to records, even when there are frequent insertions, deletions, or updates.

Think of a B+ Tree as a hierarchical structure, somewhat like a family tree but for data records. The tree is organized based on a specific attribute, often called a “search key” or “primary key”. This key is used to determine the position of records in the tree. The index value is generated for each primary key and mapped to the record.

8.2.1.1.4.1. Working of a B+ tree

The order of a B+ tree, represented by the variable ‘d’, determines the maximum number of entries in a node. Specifically, each node (except the root) must have at least ‘d’ entries and at most ‘2d’ entries, making them at least half full. Leaf nodes can end up with fewer than ‘d’ entries after deletions. Entries within each node must be sorted, allowing for efficient search operations. Between each entry in an inner node, there is a pointer to a child node. An inner node can have at most ‘2d + 1’ child pointers, which is also known as the tree’s fanout. After deletions, leaf nodes may end up with fewer than “d” entries, but the overall structure remains consistent with the defined order.

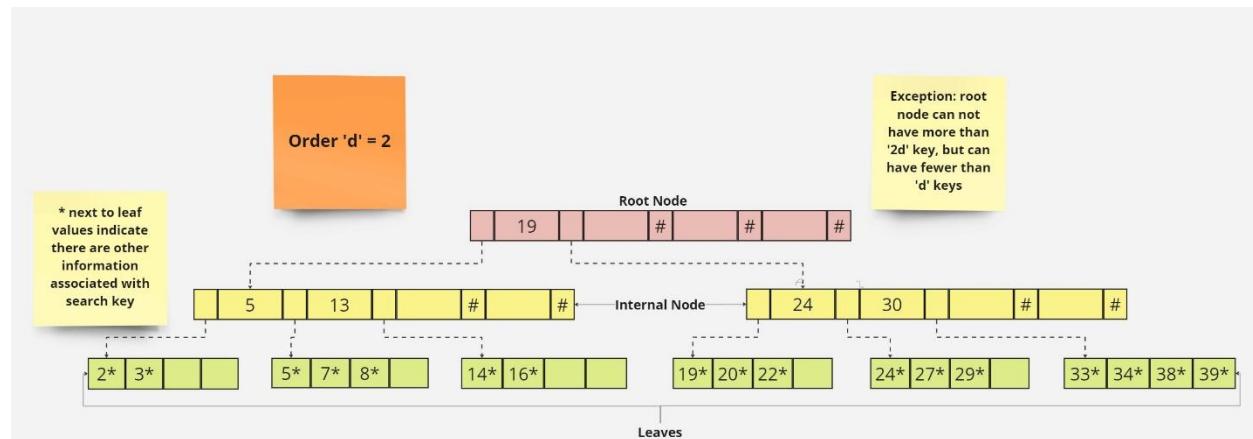


Figure 8-0-6 B+ Tree

The tree in [picture](#) has a root node and two internal nodes serve as a pointer to the leaf nodes. Values lesser than the root node are stored in nodes which are left to the root node and greater ones are stored to the right ones. This ordering facilitates the search process, guiding the traversal down the tree based on the keys. Here is how a lookup in B+ tree would look like:

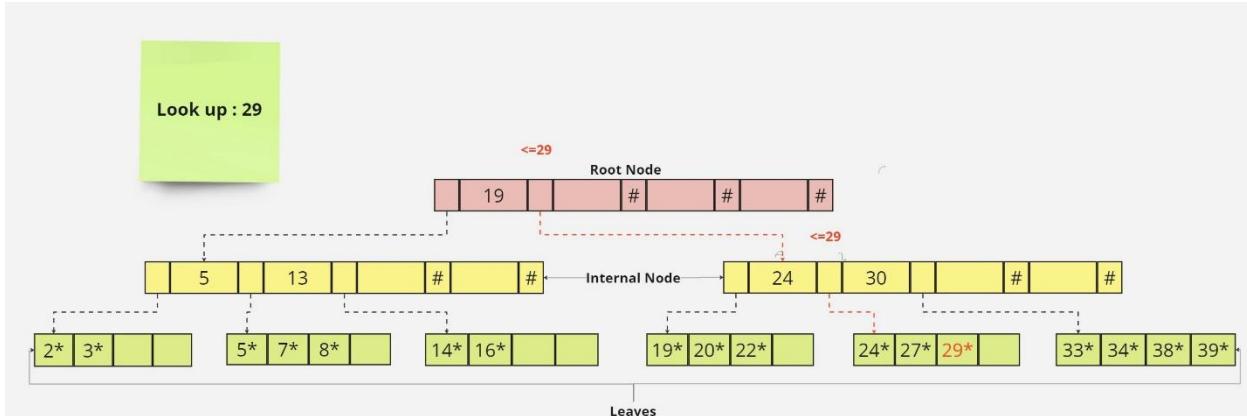


Figure 8-0-7 Look Up in B+ Tree

Now, let us look at a sample insertion in B+ tree and then understand the process and reasoning behind it.

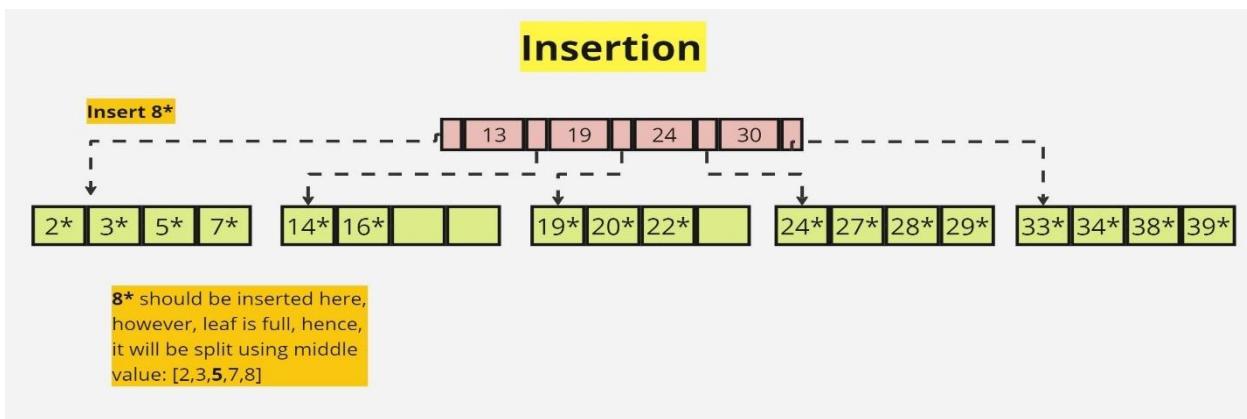


Figure 8-0-8 Insertion in B+ tree: Step 1

When a new record is added to the database, the system determines where it should be placed based on the search key. The system traverses down the tree from the root to the leaf level. At each level, it selects the appropriate branch to follow based on the search key of the record being inserted. The process continues until the system reaches a leaf node. In the leaf node, the system checks if there is enough space for the new record. If there is sufficient space, the new record is added to the leaf node in the appropriate position to maintain the sorted order.

If the leaf node is full, it may need to be split to accommodate the new record. If the leaf node is full, a split occurs. The existing records are divided into two, and the middle record is promoted to the parent node. The new record is inserted into the appropriate half of the split leaf node.

After inserting the record in the leaf node, the system checks if the parent node needs to be updated. If the parent node is full, a split may occur, and the process of promoting a record to the higher-level repeats. The updates propagate up the tree until the root, potentially causing a split at each level. If the root is split during the insertion process, a new root is created, and the height of the tree increases. Throughout the insertion process, the tree is balanced to ensure that each level has roughly the same number of branches. Balancing helps maintain the efficiency of search operations.

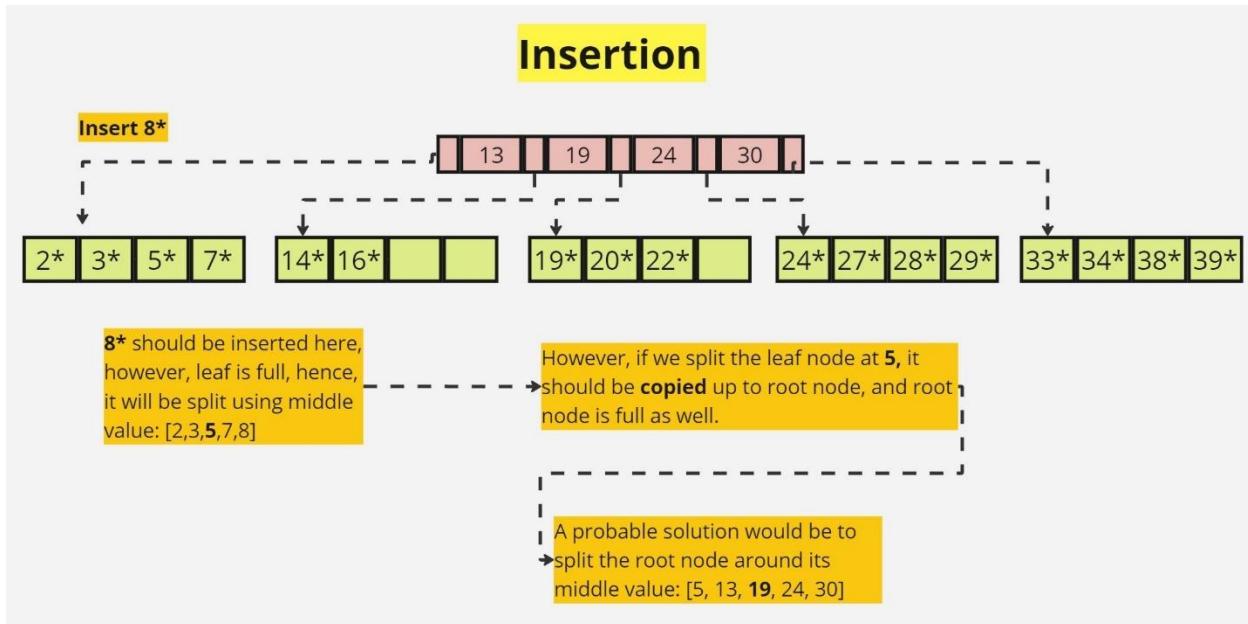


Figure 8-0-9 Insertion in B+ tree: Step 2

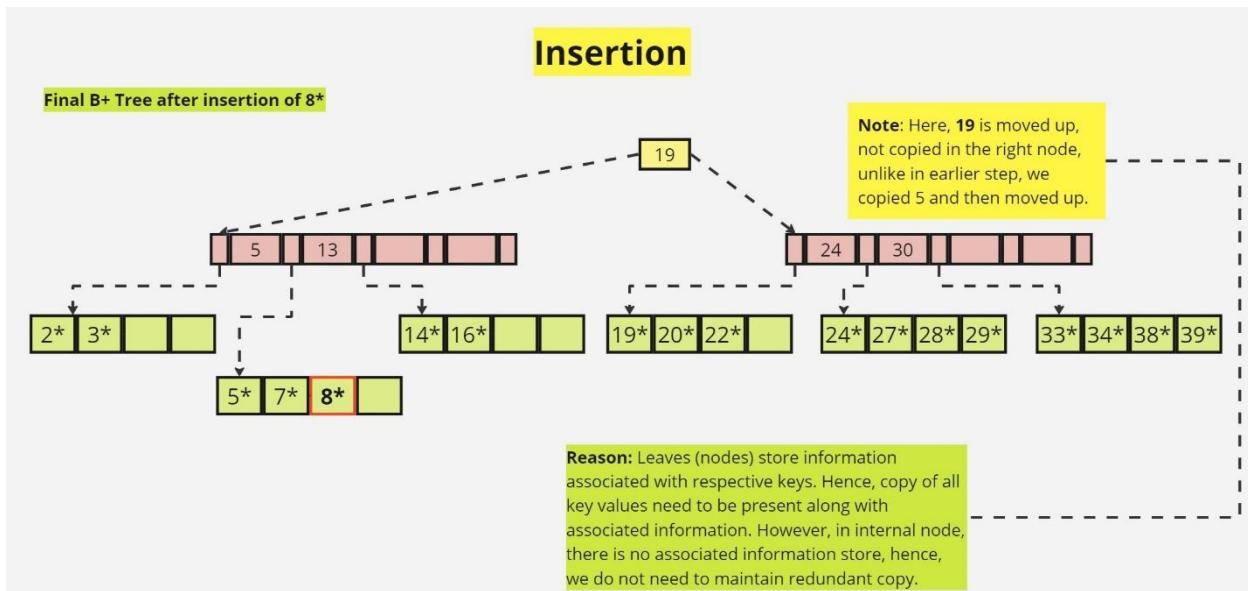


Figure 8-0-10 Insertion in B+ Tree: Step 3

Similarly, to delete a value, just find the appropriate leaf and delete the unwanted value from that leaf.

8.2.1.1.4.2. Storing of records in B+ tree

A record can be stored in B+ tree in three possible ways. We can either contain the records directly on the leaf nodes or maintain a pointer to the records on the leaf nodes. In an advanced alternative, we can store list of pointers to corresponding records in leaf nodes.

B+ Trees are particularly efficient for range queries, where we need to retrieve data within a specific range of key values. It is often used as the data structure behind multilevel index in databases. B+ trees with small nodes fitting within cache line (usually 64 bytes) also seem to provide good performance with in-memory data. It is especially well-suited for scenarios where we need quick and ordered access to data, even in the presence of frequent data changes. The hierarchical structure and balanced design make it a fundamental tool in database systems for optimizing data access.

8.2.1.1.5. Hashing File Organization

Hashing is transformation of a string of characters to a shorted fixed length value that represents original text. A shorter value helps in indexing and faster searches. It is quite popularly used in data structures to verify integrity of data. It is the heart of data structures like Hash Table, and is used to implement search algorithms, bloom filters, fast look up and queries.

In Hashing, a ‘bucket’ is the basic unit of storage that can store one or more records. In memory-based hash indices, a bucket could be a linked list of index entries or records. For disk-based indices, a bucket would be a linked list of disk blocks. In a hash file organization, buckets store either record pointers or actual records.

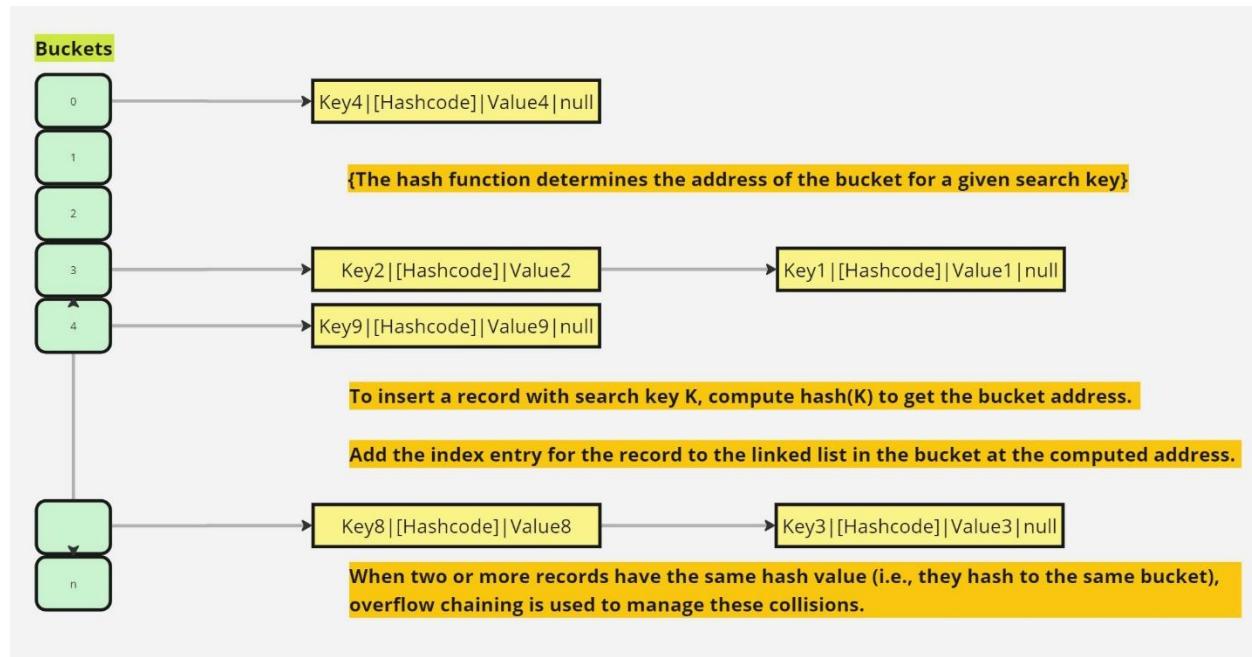


Figure 8-0-11 Data storage representation in HashMap structure

Building on this concept, hashing file organization is a technique used to organize and store records in a file based on the result of a hash function applied to a key. Every time we add or update a new piece of information (like a key-value pair) in a file, we simultaneously update a special map to remember where that information is stored within the file. This process happens whenever we insert new keys or update existing ones. When we need to find a specific value, this map helps us quickly locate where that information is stored in the file. By using this map, we directly know the precise location, so we can go straight to that spot in the file, retrieve the stored data, and read the associated value.

This method is particularly useful for achieving fast retrieval of records when the search key is known. It is often used to build in memory hash indices. However, handling collisions (when two or more records hash to the same bucket or slot in the file) can be a tricky scenario. Hashing file organization is less suitable

for range queries or partial key searches since Hash maps don't store keys in a sequential order; they are scattered across buckets based on hash values. Sequential scanning or iterating through keys within a range isn't straightforward due to non-sequential storage. The hash map doesn't maintain inherent information about key sequences or ranges. To find all keys within a range, each key needs to be individually looked up using its hash value. Performing individual lookups for each key in a range incurs significant lookup overhead and computational cost. For large ranges, the time and resources required for individual lookups can be impractical and inefficient. Alternatives can be use of B+ Trees, which use data structure like sorted trees which are suitable for maintaining sequential order of keys. Another approach can be to maintain secondary indexes or auxiliary structures alongside the hash map to facilitate range queries efficiently. Or else divide large ranges into smaller, manageable chunks to perform more targeted queries. Data structures like Sorted String Table (SSTable) or LSM trees are quite efficient in overcoming limitations poses by Hashing data structures in maintaining range queries.

8.2.1.2. Data Dictionary Storage

Earlier section, we discussed about how we can represent relation; however, database systems need to maintain information about the relations as well, kind of a knowledge hub, housing essential metadata about its structure and organization. This hub contains comprehensive details about various elements within the database, enabling effective management, querying, and optimization.

A **data dictionary** serves as a repository of metadata that describes the structure, organization, and properties of a database. It holds essential information about the database schema, such as the names of tables, columns, their data types, constraints, relationships, indexes, and other pertinent details. The storage of this critical metadata varies depending on the database system and its architecture.

In many database management systems (DBMS), the data dictionary is maintained internally within system catalogue tables. These system catalogue tables are structured tables stored within the database itself, containing detailed information about the database schema. Whereas, some databases opt for separate files exclusively dedicated to storing data dictionary information. These files might be indexed and managed by the DBMS to facilitate quick access and efficient management of metadata.

The efficiency and speed of accessing system metadata are crucial for the smooth operation of a database. To ensure rapid access and responsiveness, most databases employ a strategy of preloading system metadata into in-memory data structures during the database start-up phase before any query processing begins. These in-memory structures, like caches or in-memory tables, are optimized for quick access, allowing the system to efficiently retrieve critical metadata during runtime. By having system metadata readily available in memory, the system minimizes latency associated with disk reads, significantly speeding up access to critical information, which means faster query processing and overall system responsiveness, benefiting users and applications.

8.2.1.3. Database Buffer

In recent years, size of main memory has increased significantly, even to the extent that medium sized databases are easily fit in the main memory, however, for enterprise applications, there has a good amount of load on server's main memory and sustaining databases on main memory is not preferred yet. Larger databases are anyways much larger in size than the memory available on the servers. And for the same reasons, disk is still preferred as the storage medium for databases and data must be transferred from disk to main memory for required purposes and updates shall also be written back to the disk.

Now, this process of transfer of data from disk to main memory and back can be resource intensive as well as time consuming, and different approaches are always leveraged to minimize the cost (number of block transfers between the disk and memory). One such approach is the usage of Database Buffer or buffer pool.

The **database buffer** is a fundamental component of a database management system (DBMS), responsible for efficiently managing data between the disk and memory. It is a segment of the available main memory that's specifically reserved for storing copies of disk blocks. These disk blocks contain portions of the database and having them in-memory optimizes data access and speeds up operations. Similar to operating system concepts of virtual memory manager, a **buffer manager** oversees the allocation of memory space within the buffer for storing these disk block copies. It decides which blocks to retain in memory, evicting older or less frequently used blocks to make room for new ones. Although there is always a copy of each block on disk, the version in the buffer might be more recent due to modifications. The buffer manager determines when to write modified blocks back to disk to ensure data persistence and consistency.

The buffer manager handles I/O requests by managing buffer space, fetching data from disk to the buffer or writing modified data back to disk as needed. When a program needs a specific disk block, it sends a request to the buffer manager. The buffer manager checks if the requested block is already present in the buffer (main memory). If the requested block is present in the buffer, the buffer manager provides the requester with the address of that block in main memory. If the block is not in the buffer, the manager proceeds to allocate space in the buffer for the new block. If needed, the manager makes space in the buffer by evicting another block, employing a specific replacement policy such as **Least Recently Used (LRU)**. It selects a block for eviction, ensuring that modified blocks are written back to disk if necessary, to maintain data consistency. The buffer manager writes the evicted block back to disk only if it has been modified since its last write. Subsequently, it also reads the requested block from the disk into the buffer. And finally, the buffer manager provides the program with the address of the requested block in main memory for data access.

A Database buffer managed by a buffer manager creates a necessary level of abstraction to the programs making I/O requests.

8.2.1.4. Column-Oriented Storage

What we have discussed till now, dealt with storing all attributes of a tuple (relation) in a row-oriented storage. However, with every increasing data accumulation and rise of NoSQL based databases, demands have risen for faster access of data, especially in analytical applications. Column oriented storage is one such approach which has been trending in recent times. Unlike traditional database storage mechanism, column-oriented storage is designed to store each attribute of a relation separately and values of the attributes from successive relations or tuples are stored at successive positions in the file.

Here, a typical employee table has been represented in column-oriented approach in [Fig 8-0-12](#).

Column-oriented storage for databases is designed to reduce the overall disk I/O requirements and reduce the amount of data to be loaded from disk. Such set up can be easily scaled out using distributed clusters of low-cost hardware to increase throughput, which makes them ideal candidate for data warehousing and Big Data processing. Storing data by columns allows for better compression as similar data types are stored together thus reducing storage requirements. And since initial data retrieval is column specific, only columns needed for a particular query are retrieved, which results in reduced I/O operation and faster access time. This arrangement is well-suited for analytical queries, aggregations, data analytics and even parallel processing operations like filtering. In big data environments, columnar databases handle large volumes of structured data efficiently. Parquet and Apache ORC are two widely used column-based storage formats.

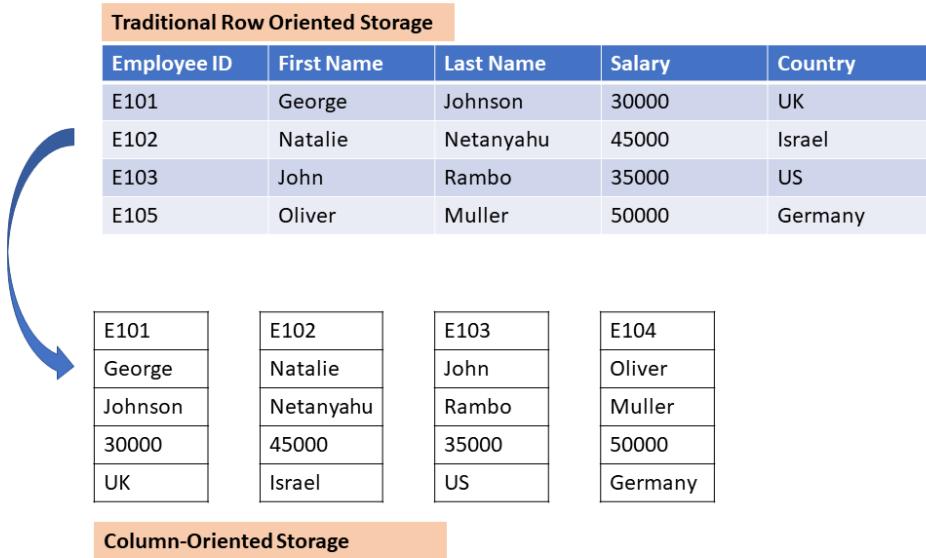


Figure 8-0-0-12 Column-Oriented storage

However, there are certain drawbacks of this approach which we must keep in mind and consider using columnar approach only in selected scenarios like analytics or Big Data processing. One of the main drawbacks of this approach is that it makes reconstruction of the original tuple very costly, hence not suited for transaction processing applications.

8.3. Indexing

While discussing about structure of magnetic disks earlier, we discussed that for the sample example, to read 100 records, 25 blocks of the disk would need to be traversed. Imagine, if the number of the records are doubled or tripled over time, the look up time would also increase twice or thrice respectively. Suppose we are looking for a particular employee's record, then it is not efficient to look up every tuple in the given relational table, to find the specific record. In ideal situation, system should be able to locate these records directly.

To speed up the retrieval of records under certain search conditions, additional auxiliary access structures called **indexes** are used. Indexes are the additional data structures present in the file system, that provide alternate ways to efficiently access a record without affecting the original placement of records in the primary data file.

General idea behind the usage of index is to keep a metadata on the side which act as specialized maps or signposts to help us quickly locate specific information within the bulk of data. Just like an index in a book lists key topics and their page numbers, database indexes list values or keys (like words in an index) along with pointers to where that data is stored in the database (similar to page numbers).

When searching for records, the database uses the index to locate pointers or references that direct it to the specific disk blocks or locations in the data file where the desired records are stored. When we perform a search or query, the database engine refers to these indexes first. It quickly navigates through the index's signposts to find the relevant data rather than scanning the entire dataset, reducing the time it takes to locate

specific information. If we want to search for the same data in different ways or based on different criteria, we might need several indexes targeting various parts of the data. Each index allows us to search the data using a specific field or criterion. For instance, one index might help search by customer names, another by product IDs, and so on.

Several databases extend the option to add or remove indexes (without altering the actual data stored in the database) to adjust for certain search condition, in order to improve query performance. Naturally, maintaining an additional structure is an overhead, especially when records are being added in the database (write operation); as the indexes would need to be updated every time there is a write operation related to the table on which index has been applied. This is considered as a necessary trade-off, as read performance of certain important query is enhanced significantly.

8.3.1.1. Types of Index: Ordered & Hashed

Moreover, not everything is indexed by default. It requires understanding of application's typical query pattern to choose indices that yield maximum benefit while limiting the overhead. For example, if we are trying to maintain index on a large table containing employee records, then implementing an index on employee relation by keeping a sorted list of employees ID will not be efficient, as index would be large in size itself and searching and update-delete operation on such index will be expensive. There are various data structure techniques (commonly B+ trees and HashMap structures discussed above) which are leveraged for implementing indexes. On a higher level, indices are divided into two categories: **Ordered indices**, based on sorted ordering of values and **Hash indices**, based on uniform distribution of values corresponding to their keys across a range of buckets (discussed above in section 8.2.1.1.5).

An **index entry or index record** serves as a reference (pointer) within a database that helps locate specific data based on a defined search key value, which is the value used for searching or querying data within the database. It is the specific attribute or field value that's being indexed for quick retrieval. The pointers indicate where the records with the corresponding search-key value are stored i.e. disk block address and offset within the disk block to locate the record.

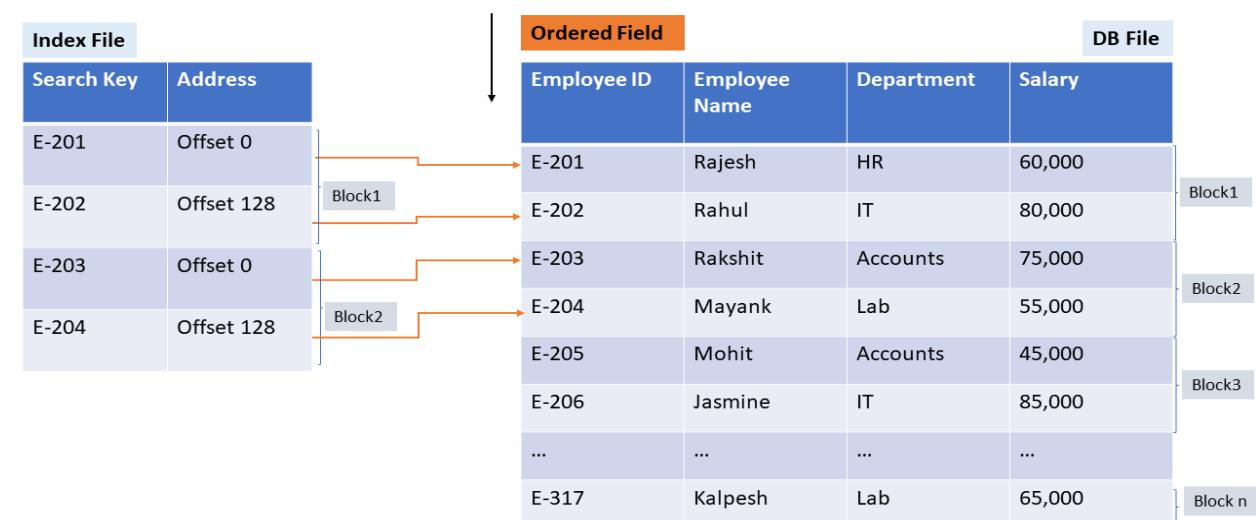


Figure 8-0-12 Dense Index

One approach to implement ordered index is to maintain an index entry for every search-key value in the file (see in Fig 8-0-12). This kind of arrangement is called **dense index**. In case a clustered index is used, a pointer to first data record corresponding to the search-key is maintained in the index file. Other approach can be to maintain an index entry only for selected search-keys (see in Fig 8-0-13). This kind of arrangement is called **sparse index**. To locate a record in this kind of set up, we find index entry with largest search-key value that is less than or equal to the search-key value we are looking for and then starting from the pointed record we start our search sequentially until the desired record is found in the file. Note: block size is considered only 128 bytes in diagrams referred.

Now, this kind of arrangement might not be very efficient if the tuples or records in the table are large in number (say in millions). In this case, a sparse outer index is created on the original index file (now referred as inner index file) and binary search is used first to locate in the inner index the largest search-key value that is less than or equal to the search-key value we are looking for. Then, corresponding offset of the disk block is referred to track the record in database file. This kind of arrangement is called **multi-level indices**.

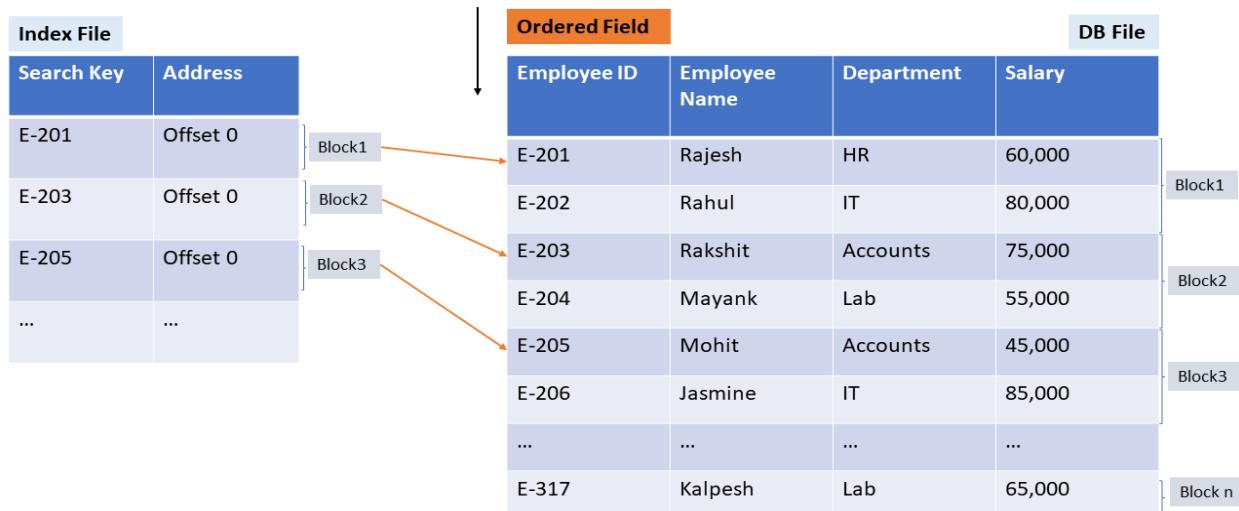


Figure 8-0-13 Sparse Index

It is to be noted that whatever form of index arrangement is implemented, every time an insertion or deletion or update operation affecting search key is performed, corresponding updates must be made in the index file. As evident, the performance of index-sequential file organization degrades as number of records increase. To counter this, indexes based on **B+ Tree** data structure is used to maintain efficiency despite insertion or deletion of data (discussed on section 8.2.1.1.4). Although a B+ tree structure imposes performance and space overhead on insertion and deletion, it is acceptable as file reorganization is avoided.

The B+ tree index structure, while efficient for many operations, faces challenges with random writes, especially when the index is too large to fit entirely in memory. When writes or inserts don't follow the order of the index, each operation often touches a different leaf node in the B+ tree. With a large number of leaf nodes and limited buffer memory, most operations require random reads followed by subsequent writes to update the leaf pages back to disk. For systems using magnetic disks, the time taken for seek

operations significantly limits the number of writes/inserts per second per disk, often restricting it to around 100 writes/inserts per second. While flash-based SSDs offer faster random I/O, the cost of a page write, particularly the eventual page erase, remains a significant operation, affecting the overall performance for random writes.

8.3.1.2. Log-Structured Merge (LSM) Tree

In scenarios with high write rates, an alternative to B+ tree can be Log-Structured Merge Tree (LSM Tree), a popular data structure designed to optimize write operations. Since, we are talking about higher write rates, Log based data structures come into play as they internally use Linked Lists (remember, Linked List have faster write time i.e. O(1); all they have to do is to append at the end of the node). However, this also means that the read operations are going to be considerably slower i.e. O(N). Now, in an enterprise set up, of course we want to speed up writes but not at the cost of reads (look ups or search queries) taking a significant hit. So how do we bring the best of both worlds? As, B+ trees are good at read or to say balanced read/write operations and linked list-based structures like Logs are good at writes but poor at reads.

So, what other data structures we can utilize for efficient reads? Answer would be Sorted Arrays, which have a read time of O(Log(N)). So, the idea is to sort the information or data supposed to be written in the database and then persist it. Sorted String Tables (SSTables) are used for the purpose of maintaining key-value pairs of data and persisting on disk.

8.3.1.2.1. Working of LSM Tree

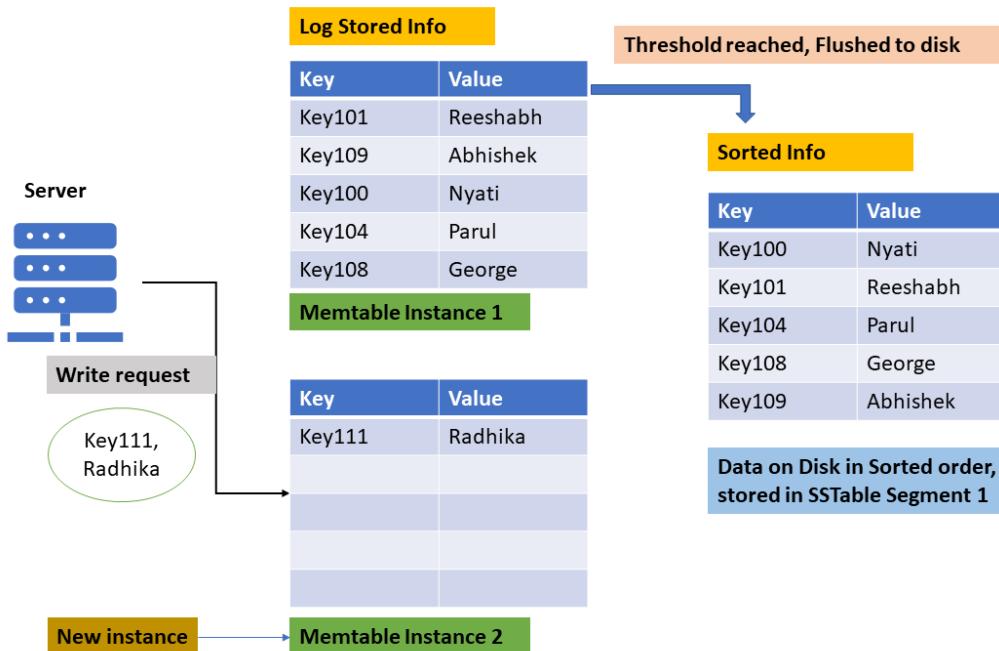


Figure 8-0-14 LSM Tree Implementation

In this arrangement (see [fig 8-0-14](#)), incoming writes are buffered in memory (RAM) in a structure known as the **memtable** (typically implemented as self-balancing tree data structure, such as red-black tree, AVL tree, etc.) and the keys within the memtable are kept in a sorted order, facilitating efficient iteration or traversal over the keys. Later, once the memtable reaches a certain size threshold (e.g., a few megabytes); the system stops further writes to that instance of the memtable. A new instance of the memtable is initiated to resume accepting incoming writes and the content of the previous memtable, which has reached its size

threshold, is flushed to disk in sorted runs or segments, typically organized as **sorted files or SSTables (Sorted String Tables)**. Writing out the memtable as an SSTable file is efficient since the in-memory tree maintains keys in a sorted order. The newly created SSTable file becomes the most recent segment of the database, holding sorted key-value pairs. While an SSTable is being written to disk, new writes continue to a new instance of the memtable, ensuring uninterrupted write operations.

There is always a possibility of data loss with in-memory data structures as in case of a crash, data not written to disk might get lost completely. To mitigate this, we can use a separate log on disk to maintain a record of in-memory writes provides a mechanism to recover data in the event of a database crash or system failure. This strategy, often referred to as a **write-ahead log (WAL)** or journaling, ensures data durability as recent writes are captured in a persistent medium (disk) and facilitates recovery as the log serves as a backup, containing unflushed writes. The system can always regenerate the memtable from the log on recovery, thus restoring the in-memory writes that weren't yet flushed to disk. Once the contents of the memtable are successfully flushed to disk (stored in SSTables), the log becomes redundant and can be discarded or truncated. However, we must be cognizant of the fact that maintaining a write-ahead log incurs additional disk writes, which might impact performance.

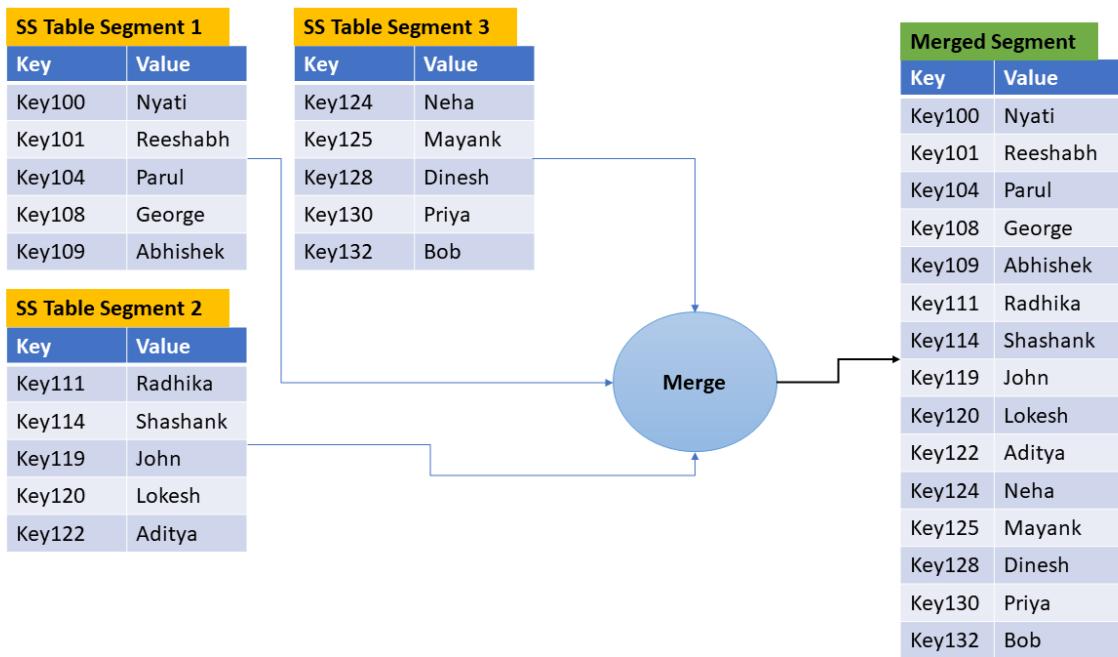


Figure 8-0-15 Merging & Compaction in LSM Tree

SSTables on the disk are present in multiple levels as we are flushing every instance of memtable, and each level containing progressively larger and more consolidated segments of data (see [Fig 8-0-15](#)). Periodically, a background process initiates merging and compaction operations on the SSTable files, which is done to optimise the reads and speed up the read process. Merging combines segment files, discards overwritten or deleted values, and optimizes the overall storage structure. By merging these segments while maintaining the sorted order of keys within each segment, compaction enhances read efficiency, minimizing the number of disk reads required to access data. Furthermore, compaction helps reclaim disk space by consolidating and removing redundant or obsolete data. There are various compaction strategies like levelling

compaction, which merges smaller SSTables within the same level, and size-tiered compaction that consolidates segments across different levels of the database hierarchy. Since, all keys are stored in sorted order in respective segments, the merged sorted order segment is created in linear time, a big advantage over Hashing based data structures.

Storage engines like Lucene use similar kind of set up for storing its term dictionary, which paves way for implementing full-text search, implemented in ElasticSearch and Solr. For serving a read request, the system first checks the memtable for the key. If not found, it looks in the most recent on-disk segment (latest SSTable), then in older segments successively. To improve efficiency of database read operations, we can utilize a sparse in-memory index table alongside SSTable blocks. This specialized index table acts as a roadmap, holding specific keys paired with their memory locations, typically storing the initial key of each block within the SSTable that comprises multiple key-value pairs (see [Fig 8-0-16](#)). When a read request arrives, this index table becomes instrumental as it swiftly pinpoints the exact block storing the sought-after key. The system navigates through this specific block, scanning through a relatively small subset of entries within the block itself to locate the desired key-value pair. By storing only select keys in the in-memory index table, memory consumption is kept minimal, ensuring these indices comfortably fit within available memory resources. This optimization proves pivotal as it eliminates the need to traverse entire SSTable blocks and dramatically speeds up read requests.

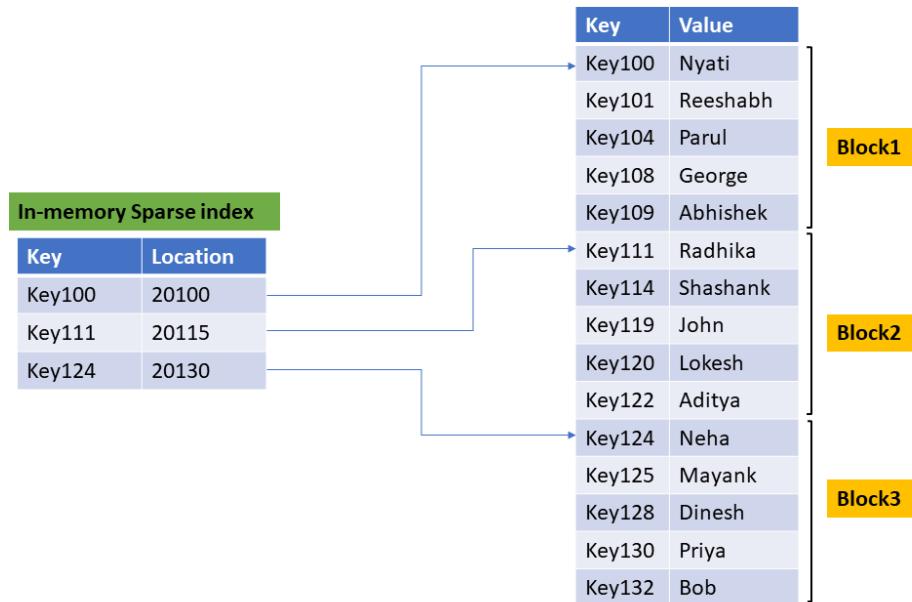


Figure 8-0-16 Use of sparse index to improve read in LSM tree

In some databases, to make the read operations more efficient, storage engines use additional filter like Bloom filter, a memory-efficient data structure for approximating the contents of a set. Benefit of a Bloom filter is that it can tell straight away if a key exists in the disk or not, hence saving unnecessary disk reads for non-existent entity.

In case, we are maintaining in-memory indexes, Hash indices (discussed on section 8.2.1.1.5) are preferred in this scenario. This approach is not commonly used during file organization as its major disadvantage lies in managing collisions. Bitmap indices are designed to query on multiple key and there are a number of

techniques for indexing temporal data, including the use of spatial index and the interval B+ tree specialized index.

8.4. Summary

In this section, we discussed how files can be arranged logically as a sequence of records onto disk blocks and how efficiently we can retrieve them using indexes. We discussed several storage structures which work under the hood in database systems and implementation of indexes. We discussed various storage approaches like database buffer and column-oriented approach. In the next section, we shall be discussing some data models which are used in Database systems. Data Models help us understand how different databases structure their data and maintain relationship between entities.

Chapter 9

Data Models

“

“Bad programmers worry about the code. Good programmers worry about data structures and their relationships.” - Linus Torvalds.

Data modelling is analogous to how we perceive an information and how we interpret it. Humans try to create a mental model of their surroundings and perceive the world according to that lens. Similarly, in software applications, we create a blueprint of how different entities are structured and their interconnection.

When we were discussing Objects in Section I, we looked at different patterns how objects are structured in a program. When we want to store these object structures, we leverage data models such as relational models (tables and views), graph model or we just store them in JSON or XML documents.

Data models are designed to meet the specific needs and requirements of a business. They serve as a blueprint for organizing and structuring data within a system, ensuring that the data is represented accurately and consistently.

Business stakeholders, such as managers, analysts, and subject matter experts, play a crucial role in providing feedback and defining the rules and requirements for the data model. They have a deep understanding of the business processes and goals, and their input helps shape the design of the system.

During the initial stages of building a new system or iterating on an existing one, business stakeholders provide valuable insights into the data that needs to be captured, stored, and processed. They define the relationships between different data entities, specify the attributes and properties that should be associated with each entity, and outline the business rules and constraints that govern the data.

By incorporating this feedback into the data model, the system can accurately represent the business's information needs. This enables efficient data storage, retrieval, analysis, and reporting, ultimately supporting the business in making informed decisions and achieving its objectives.

However, data models are not static and can evolve over time. As the business requirements change or new insights are gained, the data model may need to be updated and refined. Therefore, it's an iterative process that involves ongoing collaboration between business stakeholders and technical teams to ensure that the data model remains aligned with the evolving needs of the business.

9.1. Designing Data Models

The design process of a database system typically starts at a high level of abstraction and gradually becomes more detailed and specific as the design progresses.

At the initial stages, the design process focuses on understanding the business requirements and objectives. This involves gathering information from stakeholders, conducting interviews, and analysing existing systems and processes. The emphasis is on capturing the overall goals, functionality, and scope of the system.

Once the high-level requirements are established, the next step is to create a **conceptual data model**. This model represents the business entities, their relationships, and the key attributes without delving into implementation details. It provides a high-level view of the system and helps stakeholders visualize the structure and organization of the data. **Entity-Relationship model (ER model)** is a high-level conceptual model which is frequently used for the conceptual design of database applications, and many database design tools employ its concepts ([Figure 8-1](#)).

From the conceptual data model, the design process moves to the **logical data model**. This involves transforming the conceptual model into a more detailed representation that aligns with the chosen data management technology, such as relational databases, NoSQL databases, or a combination of different technologies. The logical data model defines the entities, attributes, relationships, and constraints in a technology-independent manner.

Once the logical data model is in place, the design process progresses to the **physical data model**. This step involves mapping the logical data model to the specific features and capabilities of the chosen data management technology. It includes considerations such as defining data types, indexing strategies, partitioning schemes, and optimizing performance.

Underneath the data model, databases employ physical storage mechanisms to optimize access paths and improve performance. These mechanisms are designed to efficiently store and retrieve data based on the database management systems (DBMS) architecture and the specific requirements of the database.

In following sections, we shall be discussing key data models and their usability in modern software development.

9.2. Relational Data Models

Relational data models are used in database management systems (DBMS) to organize and structure data in a tabular format. They are based on the principles of the relational model proposed by Edgar F. Codd in his seminal paper published in 1970.

9.2.1. Inception of Relational Data Models

 In the early days of computing, data management was often complex and involved multiple levels of data structures. This complexity made it challenging to handle and retrieve data efficiently. There was an urgent need for data independence, where applications should be insulated from the underlying physical storage structures. This would enable changes to the database's internal storage without affecting the applications using the data.

 Edgar F. Codd in the 1970s proposed to organize data into relations (table) and each relation as an unordered collection of tuples (rows). Codd's work was rooted in mathematical set theory and relational algebra. He formalized the relational model based on these mathematical principles, providing a solid theoretical foundation for data management. And till date, relational data models hold the primary position in designing commercial enterprise applications.

The relational model gained popularity due to its simplicity, which made it easier for programmers to work with compared to earlier models like the hierarchical and network models. The table-based structure of the relational model, where data is organized into rows and columns, provided a straightforward and intuitive representation of data. This simplicity facilitated the development of applications and eased the job of programmers, leading to widespread adoption. The relational model introduced the concept of referential integrity and defined rules for maintaining consistency and relationships between data entities.

Over its half-century of existence, the relational model has continuously evolved by incorporating new features and capabilities. The model has expanded beyond its initial foundations to include object-relational features, such as support for complex data types and stored procedures. These additions enhanced the expressiveness and power of the relational model, allowing it to handle a broader range of application requirements.

To meet the demands of modern data processing, the relational model has incorporated various features to support new data types and formats such as **JSON** or **XML**. This flexibility has allowed the model to handle semi-structured and unstructured data alongside traditional structured data.

One of the key strengths of the relational model is its independence from specific low-level data structures. While various underlying storage mechanisms, such as **B-trees or hash indexes**, are used to optimize performance, the relational model abstracts away these details. This abstraction has enabled the relational model to persist and remain relevant, even with the advent of new approaches to data storage, including modern column-stores designed for large-scale data mining.

The relational model has embraced advancements in tools and technologies to support emerging data requirements. For instance, relational databases have incorporated features like **data warehousing, online analytical processing (OLAP), and data mining** to facilitate complex analytics and reporting. These tools and technologies have extended the capabilities of the relational model and enabled it to handle diverse data processing scenarios.

The relational model has benefited from standardization efforts, such as the SQL language, which provides a common interface for interacting with relational databases. The wide adoptions of **relational database management systems (RDBMS)**, including commercial offerings like **Oracle, SQL Server, PostgreSQL, and MySQL**, have contributed to the enduring dominance of the relational model. The availability of robust RDBMS products, along with a vast ecosystem of tools, frameworks, and expertise, has reinforced the position of the relational model in the industry.

9.2.2. Structuring of Relational Data Models

Informally, each relation resembles a table of values or, to some extent, a flat file of records. It is called a flat file because each record has a simple linear or flat structure. Let us discuss the key components of underlying structure in detail.

1. **Tables and Rows:** Tables are the central component of the relational model. Each table represents a distinct entity or concept and consists of rows, also known as tuples. Rows represent individual instances or records within the table. For example, in a customer table, each row would represent a unique customer entry.
2. **Columns and Attributes:** Columns, also referred to as attributes or fields, define the characteristics or properties of the data stored in a table. Each column has a specific data type (e.g., integer, string, date) and represents a particular piece of information. For example, in a customer table, columns could include attributes such as customer ID, name, email, and address.
3. **Primary Keys:** A primary key is a column or a combination of columns that uniquely identifies each row within a table. It provides a way to establish the identity of the records and serves as a reference for maintaining data integrity and establishing relationships with other tables. Primary keys ensure that each row has a **unique identifier**, making it easy to locate and retrieve specific data.
4. **Foreign Keys and Relationships:** Foreign keys establish relationships between tables by referencing the primary key(s) of another table. These relationships define dependencies and

associations between entities. For example, in a relational model for an e-commerce system, a foreign key in the "Orders" table could reference the primary key of the "Customers" table, creating a relationship between orders and customers. Foreign keys ensure referential integrity and enable the retrieval of related data through join operations.

5. **Normalization:** Normalization is a process used to eliminate redundancy and improve data integrity in relational models. It involves breaking down larger tables into smaller, more focused tables, reducing data duplication and ensuring that each piece of data is stored only once. Normalization follows a set of rules called normal forms, with the most commonly used being the first, second, and third normal forms (1NF, 2NF, and 3NF). Normalization helps in avoiding data anomalies, minimizing storage requirements, and promoting efficient data manipulation.
6. **Indexing:** Indexes are used to enhance query performance in relational models. An index is a data structure that allows for faster data retrieval by creating an ordered representation of specific columns. Indexes speed up searches by reducing the need for full table scans and enabling direct access to specific rows based on the indexed column(s). Common types of indexes include B-trees, hash indexes, and bitmap indexes.
7. **Denormalization:** Denormalization is the opposite of normalization and involves selectively reintroducing redundancy into the relational model to improve performance. Denormalization aims to optimize query performance by reducing the number of joins or by precomputing certain aggregations. It is often used in data warehousing and analytical scenarios where query speed is prioritized over strict adherence to normalization principles.
8. **Schema Design and Constraints:** Schema design involves planning the overall structure of the database, including tables, columns, relationships, and constraints. Constraints, such as uniqueness, nullability, and data type constraints, ensure data integrity and enforce business rules. Well-designed schemas help ensure the efficiency, accuracy, and maintainability of the database.

Following is an ER Diagram for a university database, which tries to depict relation between entities and their internal structure. Table names are in color headings, primary keys are the attributes marked in bold font and foreign key relations are depicted via connectors. Attributes are defined alongside the data types used for storing information.

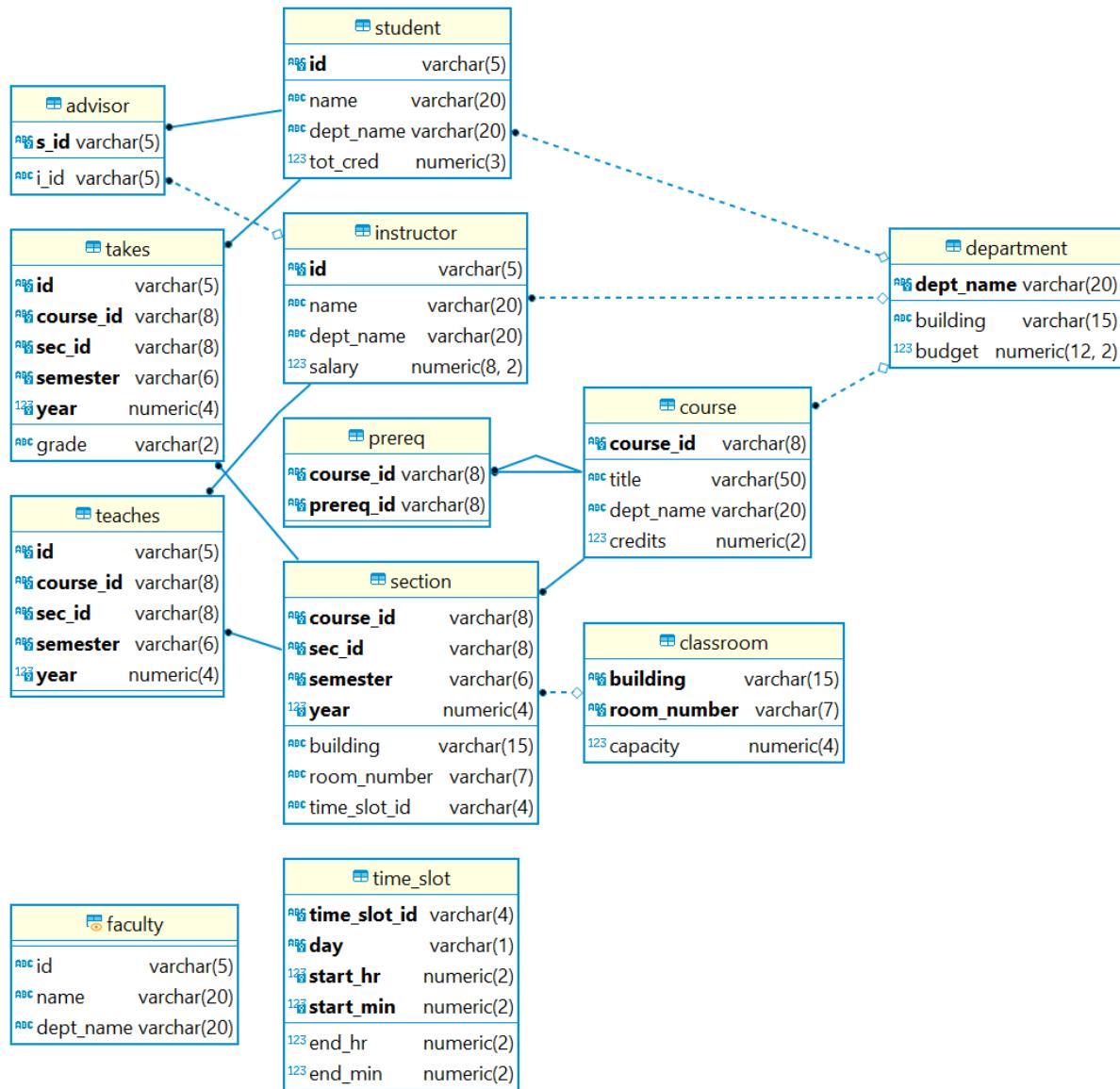


Figure 8-0-1 University DB

 Structuring relational models involves organizing data into tables with rows and columns, establishing primary and foreign keys to define relationships, and applying normalization techniques to eliminate redundancy and improve data integrity. Indexing and constraints play vital roles in optimizing performance and enforcing data consistency. The design of the schema and adherence to best practices contribute to the effectiveness of the relational model in storing and retrieving data accurately and efficiently.

9.2.3. Query Language

Structured Query Language (SQL), used with relational database management systems (**RDBMS**), is the most widely used declarative query language, based on relational models. Although, it is called a query language, it is much more than the tag. SQL also allows to define the schema of a database using DDL statements. It helps to specify integrity constraints, which define the rules and conditions that the data stored

in a database must satisfy. These constraints ensure data integrity and help maintain the consistency and accuracy of the database.

Let us have a brief look at some SQL operations being performed on the [University DB](#).

9.2.3.1. DDL Statements

SQL DDL statements are used to define the schema and constraints that data entities must adhere to.

```
CREATE TABLE course (
    course_id varchar(8) NOT NULL,
    title varchar(50) NULL,
    dept_name varchar(20) NULL,
    credits numeric(2) NULL,
    CONSTRAINT course_credits_check CHECK ((credits > (0)::numeric)),
    CONSTRAINT course_pkey PRIMARY KEY (course_id)
);
ALTER TABLE course ADD CONSTRAINT course_dept_name_fkey FOREIGN KEY (dept_name)
REFERENCES department(dept_name) ON DELETE SET NULL;
```

The above statements are SQL commands used to create a table named "*course*". The table has the following columns with data types mentioned:

- *course_id*: A varchar(8) column that cannot be null.
- *title*: A varchar(50) column that can be null.
- *dept_name*: A varchar(20) column that can be null.
- *credits*: A numeric(2) column that can be null, with a constraint that ensures the value is greater than 0.

Additionally, the table has two constraints:

- *course_credits_check*: This constraint ensures that the "*credits*" column contains values greater than 0.
- *course_pkey*: This constraint defines the primary key of the table as the "*course_id*" column.

There is also an ALTER TABLE statement that adds a foreign key constraint to the "*course*" table:

- The *course_dept_name_fkey* constraint references the "*dept_name*" column of the "*department*" table.
- The *ON DELETE SET NULL* option specifies that when a referenced row in the "*department*" table is deleted, the corresponding value in the "*dept_name*" column of the "*course*" table will be set to *NULL*.

9.2.3.2. DML Statements

Let us look at some sample DML statements which help us query the database records, modify, update, or delete records.

```
-- Retrieve course_id, title, dept_name, and credits from the course table
SELECT course_id, title, dept_name, credits
FROM course;
-- Insert a new row into the course table
INSERT INTO course
(course_id, title, dept_name, credits)
VALUES('BIO-101', 'Intro. to Biology', 'Biology', 4);
```

```
-- Retrieve course_id, title, dept_name, and credits from the course table where
course_id is 'BIO-101'
SELECT course_id, title, dept_name, credits
FROM course
WHERE course_id='BIO-101';
-- Update the title, dept_name, and credits of the row with course_id 'BIO-101' in
the course table
UPDATE course
SET title='Introduction to Biology', dept_name='Biology', credits=4
WHERE course_id='BIO-101';
-- Delete the row with course_id 'BIO-101' from the course table
DELETE FROM public.course
WHERE course_id='BIO-101';
```

9.2.3.3. Benefits of SQL as Declarative Language

The wide acceptance of SQL as query language for relational models is due to its declarative nature.

SQL offers a concise and intuitive way to express database operations compared to imperative languages such as IMS and CODASYL. With SQL, developers can describe their data needs using declarative statements that specify what data is required without specifying how to retrieve it. This abstraction allows developers to focus on the desired results rather than the detailed steps to achieve those results. This makes SQL more approachable and easier to work with, particularly for developers who may not have extensive programming experience.

Declarative languages like SQL have inherent advantages in parallel execution. As CPUs evolve, they are increasingly adding more cores rather than significantly increasing clock speeds. Imperative code that specifies instructions in a particular order can be challenging to parallelize across multiple cores and machines, as it requires careful synchronization and coordination.

9.3. NoSQL Data Models

NoSQL data models provide a non-relational approach to data storage and management. Unlike traditional models, NoSQL models do not use a fixed schema and are designed to handle large volumes of unstructured or semi-structured data with high horizontal scalability. Currently we have several NoSQL data models, each catering to different types of data and use cases. But, first let us investigate the reason for its inception.

9.3.1. Inception of NoSQL

Post internet boom and the proliferation of digital technologies, there has been an unprecedented amount of data generation and it led to exponential growth in data storage and management requirements. The internet's global reach connected billions of users worldwide. Social media platforms, e-commerce websites, and online services like email, streaming, and cloud computing cater to an enormous user base, generating vast amounts of user-generated data and interactions. The consumption of multimedia content, including images, videos, and audio, has exploded. Social media, video-sharing platforms, and online entertainment services contribute to the growth of multimedia data. IoT devices, such as sensors, wearables, and smart appliances, generate a massive influx of data from various sources. IoT devices continuously collect and transmit data, contributing significantly to the data deluge. The demand for real-time data analytics and decision-making led to the need for data storage systems capable of ingesting, processing, and serving data rapidly.



Traditional relational models, which rely on structured schemas and **ACID (Atomicity, Consistency, Isolation, Durability)** transactions, faced challenges in meeting the demands of these emerging data storage requirements. They faced performance bottlenecks when handling large-scale data, as scaling them horizontally requires complex sharding and partitioning techniques. The rigid schema of relational databases hinder adaptability to dynamic, semi-structured, or unstructured data. High write-throughput and real-time analytics demanded by modern applications strain traditional data models, designed primarily for read-heavy workloads. Complex joins and normalization lead to performance degradation when dealing with complex queries involving multiple tables. The cost of scaling traditional databases to handle the growing data volumes and traffic proved to be prohibitive for many organizations.



Hence, to address these challenges and cater to the needs of modern web-scale applications, NoSQL data models emerged as an alternative approach. NoSQL data models offer **flexible schema designs, allowing the storage of semi-structured and unstructured data** without predefining fixed schemas. They support various data models (e.g., document, key-value, columnar, graph), allowing developers to choose the most appropriate model for their application's requirements. They often provide eventual consistency models, prioritizing availability, and partition tolerance over strict consistency. This ensures that data is **eventually consistent across distributed nodes**. NoSQL databases are designed to scale horizontally, allowing them to handle very large datasets and high write throughput. Traditional relational databases, while capable of vertical scaling, can become bottlenecks as data and traffic volumes increase. NoSQL databases provide a more efficient and cost-effective solution for applications that demand massive scalability.

In a longer run, they can be **cost-effective**, particularly for applications with massive data storage requirements. They often run on commodity hardware and do not require expensive licensing fees. Many organizations and developers prefer using free and open source software due to its accessibility, transparency, and cost-effectiveness. NoSQL databases like MongoDB, Apache Cassandra, and CouchDB are popular open-source options that align with the growing trend of choosing open-source solutions over commercial products.

NoSQL databases offer specialized data models and query capabilities that are not easily supported by the relational model. For instance, document-based databases excel at storing and retrieving unstructured or semi-structured data, while graph databases are designed to efficiently handle complex relationships and network structures.

Traditional relational databases impose a rigid schema that can be cumbersome and restrictive when dealing with evolving data structures. NoSQL databases provide a more dynamic and expressive data model, allowing developers to store and process data with varying or changing formats without the need for schema migrations.

With the rise of cloud computing, NoSQL databases have become popular choices due to their distributed nature and ability to work seamlessly in cloud-native architectures. They offer easy deployment, auto-scaling, and the ability to leverage cloud infrastructure efficiently.

9.3.2. Different NoSQL Systems

As the solutions were designed with no fixed schema design in place, different organisations developed their own proprietary systems, although some were made public later, based upon their business requirements and challenges. These solutions were designed to suit specific use cases. Hence, the onus falls

on the developer community to choose the right model after deliberately evaluating the solution, trade-offs, and cost. Let us look at some popular NoSQL systems:

1. **Google's BigTable:** Google developed BigTable, a proprietary NoSQL system, to handle large-scale data storage for applications like Gmail, Google Maps, and website indexing. BigTable introduced the concept of **column-based or wide column stores**, which efficiently manage and store data in a column-oriented manner. Apache HBase is an open-source NoSQL system based on similar principles.
2. **Amazon's DynamoDB:** Amazon created DynamoDB as a NoSQL system available through its cloud services. DynamoDB falls into the category of **key-value** data stores, providing high-performance storage and retrieval based on key-value pairs or tuples.
3. **Facebook's Cassandra:** Facebook developed Cassandra, which is now open-source and known as Apache Cassandra. This NoSQL system combines concepts from **key-value** stores and **column-based** systems, offering a robust and scalable solution for managing large datasets.
4. **MongoDB and CouchDB:** Several software companies developed their own NoSQL solutions, such as MongoDB and CouchDB, which are **document-based** NoSQL systems or document stores. These systems store and manage data in flexible, self-describing documents, making them suitable for semi-structured and unstructured data.
5. **Graph-based NoSQL Systems:** Another category of NoSQL system is graph-based databases, including Neo4J and GraphBase. These systems excel in representing complex relationships and are particularly useful for applications like social networks and recommendation systems.
6. **OrientDB:** Some NoSQL systems, like OrientDB, combine concepts from multiple categories, offering a versatile and feature-rich data management solution.

Each type of NoSQL system has its strengths and is designed to serve specific use cases and data requirements, offering organizations the flexibility and scalability needed for modern data-intensive applications.

9.3.3. NoSQL & CAP Theorem

NoSQL systems were designed not to be just schema agnostic but also to be easy for horizontal scaling when time comes. Applications started using NoSQL systems in order to ensure availability and consistency. Availability was guaranteed by data replication and in a read heavy system, serialized consistency was overlooked for eventual consistency. Eventual consistency is helpful to enhance write performance.

However, NoSQL databases also present challenges. The eventual consistency model, which some NoSQL systems adopt for high availability, can lead to data conflicts and complexities in maintaining data integrity. Schema flexibility, while beneficial for agility, may require careful planning and management to ensure data quality and consistency across applications.

9.3.3.1. CAP Theorem

The CAP theorem, proposed by computer scientist Eric Brewer, states that in a distributed system, it is impossible to achieve all three of the following properties simultaneously: **Consistency** (all nodes see the same data at the same time), **Availability** (every request receives a response, without guarantee of the data being the most recent), and **Partition Tolerance** (the system continues to function despite network partitions).

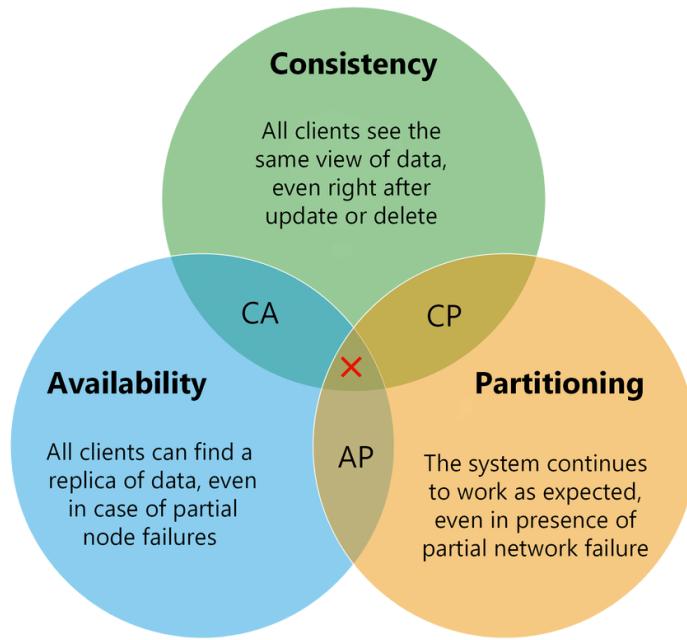


Figure 8-0-2 CAP Theorem

The CAP theorem has significant implications for the design and operation of distributed systems, including NoSQL databases. It forces architects to make trade-offs between consistency, availability, and partition tolerance when designing their systems. NoSQL systems are typically categorized based on which two properties they choose.

- **CP systems (also known as strong consistency systems)** guarantee consistency and partition tolerance. However, this means that availability may be sacrificed if a partition occurs. When a partition occurs, CP databases may choose to halt or slow down operations until the partition is resolved to maintain strict consistency. Once the partition is resolved, the data in all nodes is guaranteed to be consistent. CP systems are typically used for applications where consistency is critical, such as financial transactions. Some examples of CP systems include Cassandra, HBase, and MongoDB.
- **AP systems (also known as eventual consistency systems)** guarantee availability and partition tolerance. However, this means that consistency may be sacrificed if a partition occurs. These databases focus on providing continuous service and responsiveness even in the presence of network partitions. They may accept writes and reads even during a partition, which can lead to potential data inconsistencies. AP systems are typically used for applications where availability is critical, such as real-time streaming applications. Some examples of AP systems include DynamoDB, Redis, and Memcached.
- **CA systems (also known as strong eventual consistency systems)** guarantee consistency and availability. However, this means that partition tolerance may be sacrificed. CA systems are typically used for applications where consistency and availability are critical, but partition tolerance is not as important. Some examples of CA systems include Google Spanner and Cockroach DB.

9.4. SQL (Relational) versus NoSQL (Non-Relational)

When the situation boils down to make a choice between using relation data model or non-relational data model while designing an application, it is not a matter of one being superior to the other; rather, it's a trade-

off based on application needs. We need to scrutinize the relationships of objects, part of the application to be developed, in order to judge the trade-offs of respective data models.

NoSQL databases, especially document databases, leverage hierarchical model to store nested records with one to many relationships. But when it comes to depict many to many relationships, even NoSQL models use something like a document reference, which is a unique identifier resolved at time of follow up queries. It is similar to use of foreign keys by relational models.

While the NoSQL model offers great flexibility and scalability, it may not be the most suitable choice when dealing with complex many-to-many relationships. In such scenarios, the relational model tends to provide better support. The relational model excels at handling complex joins and efficiently representing many-to-many relationships through foreign keys, ensuring data integrity and consistency in a standardized manner.

On the other hand, the document model's approach to representing data in denormalized, nested structures can become cumbersome when dealing with many-to-many relationships. Denormalization may reduce the need for joins, but it introduces the challenge of keeping denormalized data consistent and up-to-date. This can lead to increased complexity in the application code, as the responsibility for maintaining data consistency shifts from the database to the application layer.

Additionally, emulating joins in application code by making multiple requests to the database can lead to slower performance compared to performing specialized joins directly in the database. In relational databases, joins are optimized and executed efficiently using indexing and query optimization techniques, making them a preferred choice for complex relationships.

Let's explore the main arguments in favour of the NoSQL Document data model and the strengths of the relational model:

Document Data Model (NoSQL):

- **Schema Flexibility:** The document data model, typically used in NoSQL databases like MongoDB and CouchDB, provides schema flexibility. Each document can have a different structure, allowing developers to store semi-structured and unstructured data without a rigid schema. This flexibility is beneficial in applications where data structures evolve over time or where there is a wide variety of data formats.
- **Better Performance Due to Locality:** Document-based databases store related data together in a single document, promoting better data locality. This design reduces the need for complex joins and improves read performance, as data retrieval often requires fetching only one document to obtain all the necessary information.

While the locality advantage of the document model can be beneficial for certain scenarios, it is not without its limitations. Document databases load entire documents into memory, even if an application needs only a small portion of the document. This behavior can be wasteful, especially when dealing with large documents, as it consumes unnecessary resources and may impact overall system performance.

When updates are made to a document, the entire document usually needs to be rewritten. Only modifications that don't change the encoded size of a document can be performed in place. This process of rewriting documents can be resource-intensive, particularly for large documents, and may impact write performance.

- **Alignment with Application Data Structures:** In some cases, the document data model is closer to the data structures used by the application itself. This alignment simplifies data processing and minimizes the need for complex data transformations between the application and the database.

Relational Model (SQL):

- **Support for Joins:** The relational model excels at handling complex relationships between entities through joins. Joins allow data from multiple tables to be combined into a single result set based on related keys, making it easy to retrieve data from multiple tables in a single query.
- **Many-to-One and Many-to-Many Relationships:** The relational model efficiently manages many-to-one and many-to-many relationships through foreign keys. These relationships are crucial in representing real-world scenarios, such as linking customers to orders or students to courses.
- **ACID Transactions:** Relational databases offer strong support for ACID (Atomicity, Consistency, Isolation, Durability) transactions, ensuring data integrity and consistency in multi-user environments. Transactions in relational databases are essential in maintaining data accuracy, especially in critical applications like financial systems.

9.5. Convergence

PostgreSQL since version 9.3, MySQL since version 5.7, and IBM DB2 since version 10.5 have a level of support for JSON documents. XML support is also available in relational databases for quite some time now. NoSQL databases like Rethink DB has relational-like joins in its query language. What does this mean?

Traditionally, document databases and relational databases have been seen as distinct and often mutually exclusive choices. Document databases excel at schema flexibility and handling unstructured or semi-structured data, while relational databases offer robust support for complex joins and data integrity using ACID transactions. However, the rapid evolution of technology and the growing demand for more comprehensive data solutions have led to the recognition of the complementary strengths of both models. The rise of polyglot persistence, where different types of databases are used in conjunction to meet varying data requirements, has played a significant role in driving the convergence of document and relational databases. Developers now recognize that no single database type can address all the needs of a modern application effectively. Instead, a combination of different data models, including document and relational databases, is adopted to match specific use cases and optimize performance.

To keep pace with changing data requirements, some modern relational databases have incorporated features for schema flexibility. These databases offer support for semi-structured and unstructured data, blurring the lines between traditional relational and document-oriented models. This convergence helps address the frustration with the restrictiveness of relational schemas, providing greater agility in data modelling.

The convergence of document and relational databases has given rise to **multi-model databases**. These databases allow developers to choose the data model that best fits the application's needs within a single system. Multi-model databases can handle documents, graphs, key-value pairs, and relational data, providing a unified solution for managing diverse data types.

9.6. Summary

In this chapter, we had a brief overview of different data models and a discussion on their use cases, strengths, and weaknesses. Let us summarize the learnings of the chapter:

A **data model** is a way of organizing data so that it can be stored and retrieved efficiently. There are two main types of data models: relational and non-relational.

Relational data models store data in tables, which are made up of rows and columns. Each row represents a single record, and each column represents a single piece of data about that record. Relational data models are well-suited for storing data that has a clear relationship between different pieces of data, such as customer records or product inventory.

Non-relational data models store data in a variety of ways, including documents, graphs, and key-value pairs. Non-relational data models are well-suited for storing data that does not have a clear relationship between different pieces of data, such as social media data or sensor data.

Feature	Relational Data Models	Non-Relational Data Models
Data structure	Tables	Documents, graphs, key-value pairs
Relationship between data	Clear relationship	No Clear relationship
Applicability	Storing data with a clear relationship	Storing data with no clear relationship
Examples	MySQL, PostgreSQL, Oracle	MongoDB, Cassandra, Redis

The choice of data model for a particular application will depend on the specific requirements of the application. If the application needs to store data with a clear relationship, then a relational data model may be a good choice. However, if the application needs to store data with no clear relationship, then a non-relational data model may be a better choice.

CONVERGENCE: Objects and Data

Enterprise Architecture

“

“The entrepreneur is essentially a visualizer and actualizer... He can visualize something, and when he visualizes it, he sees exactly how to make it happen.” - Robert L. Schwartz

So far, we have discussed two entities, Objects and Data. We discussed their evolution, patterns associated with their creation, storage, and maintenance. Objects and Data prove out to be the building blocks of an enterprise software application.

Data is the foundation of enterprise applications. It is the raw material that is used to create objects. Objects are created from data, represent real-world entities such as customers, products, and orders. and enterprise applications use objects to manage data. Objects have attributes that store data about the entity, and they have methods that perform actions on the entity. In turn, enterprise applications provide users with a way to interact with data and objects. Enterprise applications are software systems that use objects and data to automate business processes. They allow businesses to track their operations, manage their resources, and make better decisions.

An enterprise resource planning (ERP) system uses objects to represent products, inventory, and orders. The system stores data about these objects in a database, and it provides users with a GUI that allows them to interact with the data. The ERP system can be used to automate manufacturing and accounting processes, and it can provide insights into the financial performance of the business.

Enterprise Applications are different than other software applications and have a world of their own. They deal with complex data types, concurrent users, persistent data, user interface and in some cases integration with other enterprise applications.

Architecture is a subjective concept which deals with a high-level design of different parts of the system and defining how they are going to interact with each other. It is a subjective concept because different set of software developers can define different ways for interaction of components that will ultimately achieve the same goal. There is no common solution to a particular business requirement. There are always different design approaches with their pros and cons. It is up to the software stakeholders to judge their trade-offs.

Different business requirements have different entities and relationships and with it comes different set of challenges. Choosing an architecture for a software system is a complex task that requires a deep understanding of the system's requirements and constraints. Overall, it boils down to answering following set of questions:

What are the functional and non-functional requirements of the system? What is the performance, scalability, and availability requirements?

What are the technological constraints of the system? What are the budget and time constraints?

What is the operating environment of the system? What are the security and compliance requirements?

How is the system expected to evolve over time? What are the anticipated changes in requirements or technology?

Once the factors have been considered, the next step is to identify the different architectural options that are available. There are many different architectural patterns that can be used to design software systems. The architecture should be designed to meet the specific needs of the system, and it should be flexible enough to accommodate changes in requirements or technology. But once, the decision is made, an enterprise architecture serves as a blueprint for the organization's current and future state, ensuring that the technology solutions are in line with the business needs. It helps to optimize the organization's operations, reduce redundancy, improve efficiency, and promote interoperability among different systems.

In this chapter, we shall be discussing in brief some of the common Enterprise Architectural Patterns.

10.1. Layered Architecture

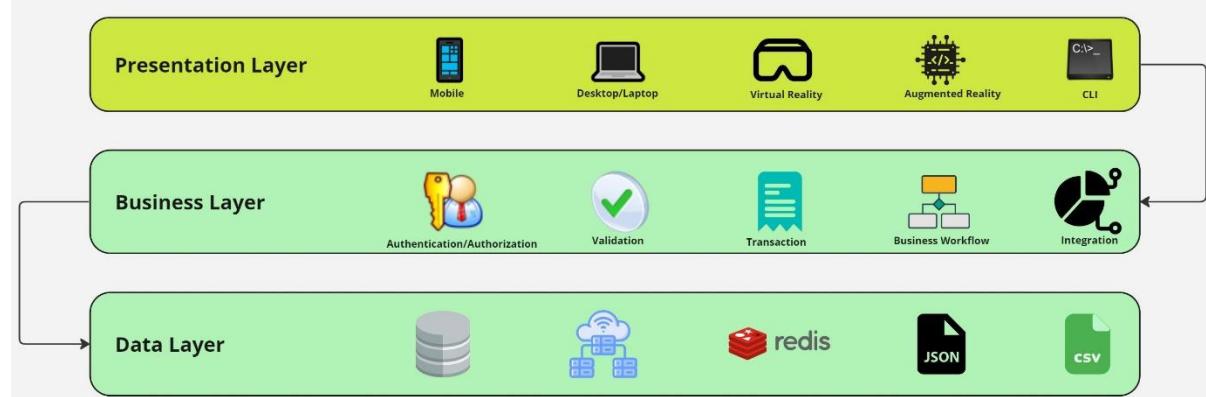


Figure 9-0-1 N-Tier Architecture

Layered Architecture, commonly known as n-tier architecture pattern, is the most common and widely used software architecture pattern in enterprise applications. It is composed of several separate horizontal layers that function together as a single unit of software.

10.1.1. Pattern Description

In a layered architecture, the components or modules of the application are organized into horizontal layers, where each layer is responsible for a specific set of functionalities. These layers are stacked on top of each other (Figure 9-1 represents a 3-tier architecture), with higher layers depending on lower layers for their operations. This enforces a clear **separation of concerns**, ensuring that each layer is focused on a specific aspect of the application's functionality. The separation makes the application easier to understand, maintain, and modify since changes within one layer are less likely to affect other layers. Layers in a layered architecture communicate with each other through well-defined interfaces, promoting encapsulation and information hiding. The internal implementation details of each layer are hidden from the layers above, reducing dependencies and coupling. However, in layered architecture, each layer can only communicate with the immediate layer above or below it. This is called **Isolation of Layers**.

Requests or data must pass through each layer sequentially, starting from the topmost layer (e.g., presentation layer) and moving down to the lower layers (e.g., business layer and persistence layer) until it reaches the bottommost layer (e.g., data or database layer). This strict adherence to layer-to-layer communication ensures a clear separation of concerns and encapsulation of functionalities within each layer.

The number and category of layers in an N-Tier architecture can vary, but a common software setup comprises of the following layers:

10.1.1.1. Presentation Layer

The topmost layer responsible for handling user interface and user interactions. It receives user inputs, presents information, and forwards requests to the next layer.

In web applications, the presentation layer is typically implemented using HTML, CSS, and JavaScript. The HTML mark-up structures the content, CSS styles the presentation, and JavaScript handles user interactions and dynamic content updates. Popular frameworks and libraries like React, Angular, and Vue.js are commonly used to build interactive and responsive web applications. In mobile applications, the presentation layer is built using native development tools or cross-platform frameworks. For command-line applications, the presentation layer is typically text-based and interacts with the user through a command-line interface. GUI applications, such as graphical data visualization tools or image editing software, have sophisticated presentation layers with graphical elements like charts, buttons, menus, and dialog boxes.

The primary goal of Presentation Layer is to provide a user-friendly interface that enables users to interact with the application and access its features easily.

10.1.1.2. Application Layer (Business Logic Layer)

This layer contains the business logic, workflows, and application rules. It processes the requests from the presentation layer and interacts with the data layer to perform necessary operations.

This layer deals with user authentication and authorization processes in an application involving different user roles. It verifies user credentials, checks permissions, and grants access to specific features or data based on user roles and privileges. In an e-commerce application, the Business Layer manages order processing. It validates user orders, calculates totals, applies discounts or promotions, and interacts with the Data Layer to update order status and inventory levels. In enterprise applications, the Business Layer can support reporting and analytics functionalities. It retrieves and processes data from the Data Layer to generate reports and insights for decision-making. In some cases, the Business Layer may perform data transformation or aggregation, preparing data from different sources for presentation in the user interface or reporting purposes.

The Business Layer should encapsulate and manage the application's core business logic, making it reusable and maintainable while promoting separation of concerns within the overall architecture.

10.1.1.3. Data Layer

This layer responsible for data storage, retrieval, and manipulation. It interacts with the database or other data sources to store and retrieve data requested by the application layer.

Relational databases, such as MySQL, PostgreSQL, Oracle, and Microsoft SQL Server, and NoSQL (Non-relational) databases are commonly used as persistence layers in many applications. We have read about relational and non-relational databases in earlier section. In some cases, especially for smaller applications or data storage needs, the persistence layer may involve simple file systems. Data can also be stored in files using various formats like JSON, XML, or CSV, and the application reads and writes data directly to these files. With the rise of cloud computing, cloud-based storage solutions like Amazon S3, Microsoft Azure Blob Storage, and Google Cloud Storage have become popular as persistence layers for handling large volumes of unstructured data, such as images, videos, and documents. Some applications also use In-memory data stores like Redis, etc. for faster data access and retrieval, making them suitable for applications requiring high performance and low latency.

The choice of a specific persistence layer depends on factors like data volume, data structure, performance requirements, scalability needs, and the overall architecture of the application.

10.1.1.4. Intermediate Layers

There can be various layers in between these three prominent layers. For example, many enterprise applications use a ‘**Data Access Layer**’ between Business Layer and Data Layer, to separate the business logic from the data persistence and retrieval operations. It abstracts the underlying database operations, allowing the application to work with a consistent API regardless of the actual database technology used (e.g., SQL, NoSQL, etc.). This abstraction shields the rest of the application from direct database interactions and simplifies the transition between different data storage solutions. ORM frameworks like Hibernate (for Java), Entity Framework (for .NET), and SQLAlchemy (for Python) bridge the gap between the object-oriented code in the application and the relational database.

A sample code for a Data Access layer is presented below.

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from models import User

class UserDao:
    """
    The UserDao class provides methods to interact with the User table in the database.
    It encapsulates the data access operations related to User objects.

    Attributes:
        engine (sqlalchemy.engine.base.Engine): The SQLAlchemy engine used to connect to the database.
        session (sqlalchemy.orm.session.Session): The SQLAlchemy session used for database interactions.
    """

    def __init__(self):
        """
        Initializes the UserDao by creating a database engine and a session.
        """
        engine = create_engine('sqlite:///database.db')
        Session = sessionmaker(bind=engine)
        self.session = Session()

    def find_by_id(self, user_id):
        """
        Retrieve a user by their ID.

        Args:
            user_id (int): The ID of the user to find.

        Returns:
            User or None: The User object if found, or None if no user with the given ID exists.
        """
        return self.session.query(User).filter_by(id=user_id).first()
```

```
def find_all(self):
    """
    Retrieve all users from the database.

    Returns:
        list of User: A list containing all User objects in the database.
    """
    return self.session.query(User).all()

def save(self, user):
    """
    Save a new user to the database or update an existing user.

    Args:
        user (User): The User object to be saved or updated.
    """
    self.session.add(user)
    self.session.commit()
```

In the code above, ‘*User*’ object can further be segregated to be part ‘**Domain Model Layer**’.

A **Domain Model Layer** represents the core business entities and their relationships in the application. In the context of applications like the *User* layer example provided earlier, the Domain Model Layer would consist of classes (Objects) that define the structure and behavior of the main business entities. While the Data Access layer deals with data access and persistence, the Domain Model Layer concentrates on defining the structure and logic of the business entities. This separation of concerns makes the application architecture more modular and maintainable.

A *User* object can be represented as below:

```
class User:
    """
    User class representing a user entity.

    Attributes:
        id (int): The unique identifier of the user.
        name (str): The name of the user.
        email (str): The email address of the user.
    """

    def __init__(self, user_id, name, email):
        self.id = user_id
        self.name = name
        self.email = email

    def __str__(self):
        return f"User(id={self.id}, name='{self.name}', email='{self.email}')"
```

10.1.2. Pattern Analysis

A layered architecture pattern is very simple to understand and implement. In fact, most of small-scale enterprise software applications are usually based on this type of pattern. However, this pattern has its own set of challenges and advantages. Let us discuss them briefly.

The layered architecture tends to lead towards monolithic applications, especially if the layers are not properly separated and managed. A **monolithic application** is one where all components and functionalities are tightly integrated into a single unit, making it challenging to scale and maintain over time. This approach may work well for small applications with simple requirements, but as the application grows in size and complexity, it can present several challenges. Since everything is tightly coupled, any change to one part of the application could potentially affect other parts, requiring the entire application to be redeployed. This can lead to longer downtime and make the deployment process error-prone. It also creates a single point of failure, which can bring down the entire system, impacting the entire application's availability and reliability. If we are dealing with a monolithic layered architecture, a business request might have to go through multiple layers of architecture, leading to low performance. Developers might take an approach of dividing different layers into separate deployable units. Although, it will pose some challenges in terms of deployment, availability, and scalability, but the idea might work for small scale business applications.

A layered architecture can be easy to test and for a small team, it provides ease of development as well, with every component at single place. It becomes a natural choice for small scale business application development. Radical changes are easy to make.

10.2. Microservices Architecture

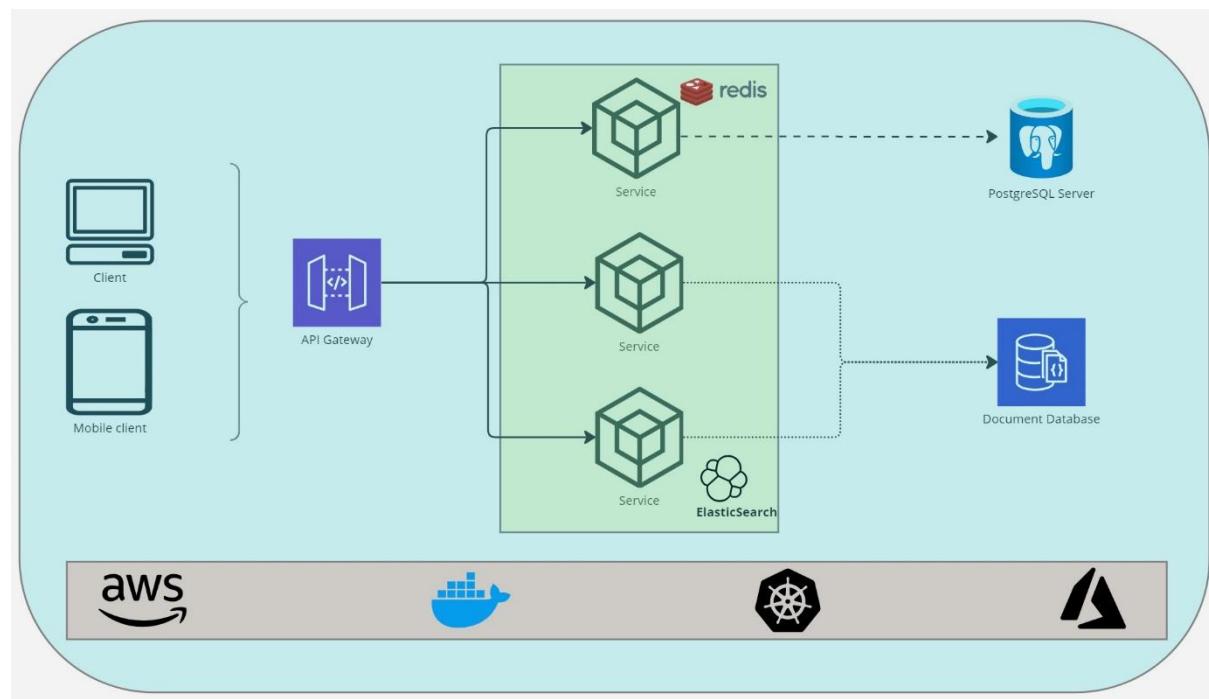


Figure 9-0-2 Sample Micro-services-based architecture setup

This is story of a successful software company named TechCube, they had been delivering cutting-edge software solutions to clients across various industries. TechCube had been running a large and complex monolithic application for a client project for quite some time. This monolith had served them well in the early stages, but as client's customer base grew, they started facing several challenges that prompted them

to consider a transition to a microservices architecture. TechCube's monolithic application was becoming increasingly difficult to manage and scale. Any minor change in one part of the application required the entire monolith to be redeployed, causing significant downtime, and risking the application's stability. The development team struggled to release updates frequently due to the monolith's tightly coupled nature.

After much deliberation and discussions among the company's leadership and tech team, they decided to embark on a journey towards adopting microservices. They recognized that this architectural shift would allow them to build more scalable, maintainable, and flexible applications that could meet the demands of their rapidly expanding user base.

The first step in the transition was to identify the different functionalities of the monolith that could be broken down into separate services. TechCube's architects and developers conducted thorough analysis and defined the boundaries for the new microservices. They identified components that could be decoupled and transformed into independent services, each responsible for a specific business capability.

TechCube started building the foundational infrastructure for the microservices architecture. They invested in a robust service discovery mechanism, a centralized API gateway, and implemented containerization using Docker to ensure consistency in deploying services across different environments.

The transition to microservices was not an overnight process. TechCube team took a gradual approach, starting with a pilot project where they extracted a relatively small, non-critical functionality from the monolith and rebuilt it as a microservice. This gave them valuable insights and experience, allowing them to fine-tune their approach for the larger transformation.

As the team continued to break down the monolith into microservices, they faced some challenges. Ensuring data consistency across distributed services, managing service communication and versioning, and dealing with new complexities in monitoring and logging were some of the hurdles they encountered.

As more microservices were developed and integrated, TechCube started to see the benefits of their efforts. Services could now be deployed and scaled independently, allowing them to handle varying workloads efficiently. Teams could focus on specific business capabilities, leading to faster development cycles and quicker releases.

TechCube didn't stop after the initial transition. They embraced a culture of continuous improvement, regularly reviewing and optimizing their microservices architecture. They invested in automated testing, CI/CD pipelines, and made use of cloud-native technologies to further enhance their system's reliability and scalability.

Over time, TechCube successfully transformed its monolithic application into a resilient, scalable, and flexible microservices-based architecture. Their development teams were happier and more productive, and they could now respond to market demands with agility and confidence. Their customers enjoyed a more reliable and performant software experience, solidifying TechCube's position as a market leader.

With the successful transition to microservices, TechCube continued to innovate and evolve its products. They remained committed to staying at the forefront of technology, constantly seeking new ways to improve and deliver value to their clients. The journey towards microservices had been challenging, but it had transformed TechCube into a company capable of tackling any future technological challenges that lay ahead.

A service is a small application that implements narrowly focused functionality, such as order management, customer management, and so on. The high-level definition of microservice architecture is an architectural style that functionally decomposes an application into a set of services.

10.2.1. Pattern Description

Since, we have defined what is a microservice architecture, then let us also formally define what a service is expected to be.

A service is a standalone, independently deployable software component that implements focussed useful functionality.

A **microservice architecture** is also called as distributed architecture as the components within the architecture are fully decoupled from one other and accessed through some sort of remote access protocol/APIs (e.g., MS, AMQP, REST, SOAP, RMI, etc.) via **API Gateway**. Services can have their separate datastore or a couple of services can share the same datastore initially and later evolve to have independent datastore for itself.

10.2.1.1. API Gateway

A Gateway encapsulates access to an external system or resources. It is implemented using Adapter Pattern, also known as wrapper pattern, which we have already discussed. In a microservices architecture, each service typically exposes its own API, which can lead to a complex web of API endpoints for clients to manage. The API Gateway addresses this complexity by encapsulating the complex and specialized API code of an external resource into a class that presents a simpler and more user-friendly interface to the rest of the application. It acts as an intermediary between the application and the external resource, translating simple method calls from the application into the appropriate specialized API calls required by the external resource.

It helps decouple the application from the external resource's API and provides a layer of abstraction, making the application code more maintainable, scalable, and user-friendly. It is particularly useful when dealing with external resources that might change or have complex interfaces, as it allows the application to adapt to these changes with minimal impact.

10.2.1.2. Distinguishing Microservices from Service Oriented Architecture

Microservice architecture is an evolutionary architecture pattern, which came to inception due to problems being faced in other patterns. Its close resemblance is argued with **Service Oriented architecture (SOA)**; however, similarities seem to be only on higher level. SOA applications typically use heavyweight technologies such as SOAP, ESB, etc. While microservices use lightweight, open source technologies such as REST, message broker, gRPC, etc. Typically, SOA has a global data model, whereas in contrast, microservices have separate data stores and hence localized data models.

10.2.1.3. Defining Microservices

In the process of defining the microservice architecture for an e-commerce application, the first step is to **identify the system operations**, which represent the external requests the application needs to handle. These system operations are abstractions of commands that update data or queries that retrieve data. They are derived from the functional requirements of the e-commerce application, often expressed as user stories.

For the e-commerce application, some of the identified system operations could be:

- As a customer, I want to add items to my shopping cart, so that I can purchase them later.
- As a customer, I want to place an order, so that I can buy the items in my shopping cart.
- As a seller, I want to manage my inventory, so that I can keep track of available products.

These system operations will serve as the architectural scenarios that illustrate how the microservices will collaborate to fulfil these user requests.

The second step in the process is to **determine the decomposition** of the e-commerce application into services. One of the strategies mentioned is organizing services around business capabilities or **domain-driven design** subdomains. The goal is to create services that are organized around business concepts rather than technical concepts.

For our e-commerce application, the identified services could be:

- Cart Service: Responsible for handling operations related to the shopping cart, such as adding items, removing items, and managing the cart's contents.
- Order Service: Responsible for managing the order lifecycle, including order placement, payment processing, and order fulfilment.
- Inventory Service: Responsible for managing the inventory of products, tracking stock levels, and updating product availability.

The third step in defining the architecture is to **determine each service's API**. This involves assigning the system operations to the corresponding services. Some services may be able to handle operations independently, while others might need to collaborate with other services to fulfil certain requests.

For example:

- The Cart Service might implement the *addItemsToCart()* and *removeItemsFromCart()* operations on its own.
- The Order Service might implement the *placeOrder()* and *processPayment()* operations but could collaborate with the Cart Service to retrieve the cart contents.

To enable communication between the services, appropriate **IPC (Inter-Process Communication)** mechanisms need to be chosen, such as **REST** or messaging brokers.

Throughout this decomposition process, there can be obstacles and challenges. Some of these include network latency, the need to maintain data consistency across services, and dealing with god classes (classes that are used throughout an application and hinder decomposition). However, solutions can be found, such as using self-contained services to reduce synchronous communication and employing sagas to maintain data consistency.

By following this three-step process of identifying system operations, determining services, and defining service APIs and collaborations, the e-commerce application can be effectively decomposed into a microservices architecture, facilitating better scalability, maintainability, and flexibility to meet the demands of its growing customer base.

The actual implementation of a microservice architecture is a more intricate process that involves various considerations, such as service decomposition, communication protocols, data management, scalability, and fault tolerance, among others. It requires careful planning and design to ensure the success of the architecture, and requires a detailed discussion, which is out of scope for our agenda. For students or anyone interested in learning more about microservices, there are plenty of excellent resources available. Books, articles, tutorials, and online courses offer in-depth discussions and practical examples to help understand the concepts and best practices involved in building microservices.

10.2.1.4. Managing Transactions in Microservices

In microservice architecture, transactions typically span multiple services, which can make managing them more complex than in a monolithic architecture. To simplify this process, some useful strategies are implemented by architects to overcome challenges of transaction.

Sagas: This strategy uses a sequence of local transactions to achieve a larger, distributed transaction. Each local transaction updates the state of a single service, and the saga coordinates the overall flow by sending commands or events to the other services to tell them to perform their local transactions. If any of the local transactions fail, the saga can compensate by sending commands to undo the changes made by the previous successful transactions.

Event-driven architecture: In this approach, instead of tightly coupling services via transactions, services communicate by emitting and subscribing to events. This allows for a more loosely-coupled and scalable architecture, but it can make it more difficult to ensure consistency across services. We shall be studying this approach in detail in upcoming section.

Distributed databases: We have already discussed about distributed database management where a single database can be partitioned into multiple smaller databases which can be owned by different services, this way the service can handle their own transactions without the need of a coordinator. This approach can lead to more consistency and less complexity but can also lead to more data duplication.

It's worth noting that each of these strategies has its own trade-offs and may be more or less appropriate depending on the specific requirements of your application.

10.2.2. Pattern Analysis

Let us first discuss the benefits presented by Microservice Architecture pattern.

10.2.2.1. Benefits

The microservice architecture provides significant benefits, with one of the most crucial advantages being its ability to enable continuous delivery and deployment of large, complex applications. This aligns with the principles of **DevOps**, which focuses on rapid, frequent, and reliable software delivery. High-performing DevOps organizations can deploy changes into production with minimal production issues.

There are three key parameters that the microservice architecture supports continuous delivery and deployment:

- **Testability:** Automated testing is a fundamental practice in continuous delivery/deployment. In a microservice architecture, each service is relatively small, making it easier and faster to write and execute automated tests. As a result, the application becomes more robust with fewer bugs.
- **Deployment independence:** Each microservice can be deployed independently of other services. When developers make changes to a specific service, they don't need to coordinate with other teams. They can deploy their changes directly. This autonomy in deployment makes it much easier to push changes frequently into production.
- **Autonomous and Loosely Coupled Teams:** The microservice architecture allows the engineering organization to be structured as small, independent teams (often referred to as two-pizza teams). Each team is responsible for developing and deploying one or more related services. These teams can operate independently and scale their services without being tightly coupled to other teams. As a result, the development velocity increases significantly.

The ability to achieve continuous delivery and deployment brings several valuable benefits for businesses. With continuous delivery, the time it takes to go from development to production is significantly reduced.

This enables the business to quickly respond to customer feedback and market demands, staying ahead of the competition. Customers expect reliable and seamless service. Continuous deployment ensures that the latest features and bug fixes are rapidly delivered, enhancing the user experience, and maintaining customer satisfaction.

When development teams spend less time dealing with operational issues and firefighting, they can focus more on delivering valuable features. This increased focus on adding value instead of dealing with problems leads to higher employee satisfaction and motivation.

Services in microservice architecture can be scaled independently and have better fault isolation. Services can have different tech stack and new tech stack can be easily adapted. However, with all the benefits of Microservices discussed till now, there also comes some challenges and complexities.

10.2.2.2. Challenges

Microservices introduce a distributed system environment, where multiple services interact with each other over the network. This complexity can make development, testing, and deployment more challenging. Developers must handle issues such as network latency, service discovery, load balancing, and fault tolerance. Due to its distributed nature, performance takes some toll.

Certain features may span multiple services, and deploying such features requires careful coordination. Changes in one service might impact others, making it crucial to have effective communication and synchronization between teams to avoid breaking the application.

Maintaining data consistency across multiple services can be challenging, particularly when transactions involve multiple services. Implementing distributed transactions and managing eventual consistency can be complex tasks.

Testing in a microservices architecture becomes more intricate as services need to be tested individually and as part of an integrated system. Ensuring proper test coverage and end-to-end testing across services is vital to maintain the application's reliability.

Deciding on the appropriate service boundaries and defining clean and efficient APIs are critical for the success of a microservices architecture. Poorly defined boundaries can lead to service dependencies and hinder the ability to evolve and scale the system independently.

It is not a good idea to go for a microservice approach right from the scratch. Ultimately, organizations need to weigh the pros and cons, conduct a thorough cost-benefit analysis, and carefully plan their microservices journey or transition from monolith to microservices.

10.3. Event Driven Architecture

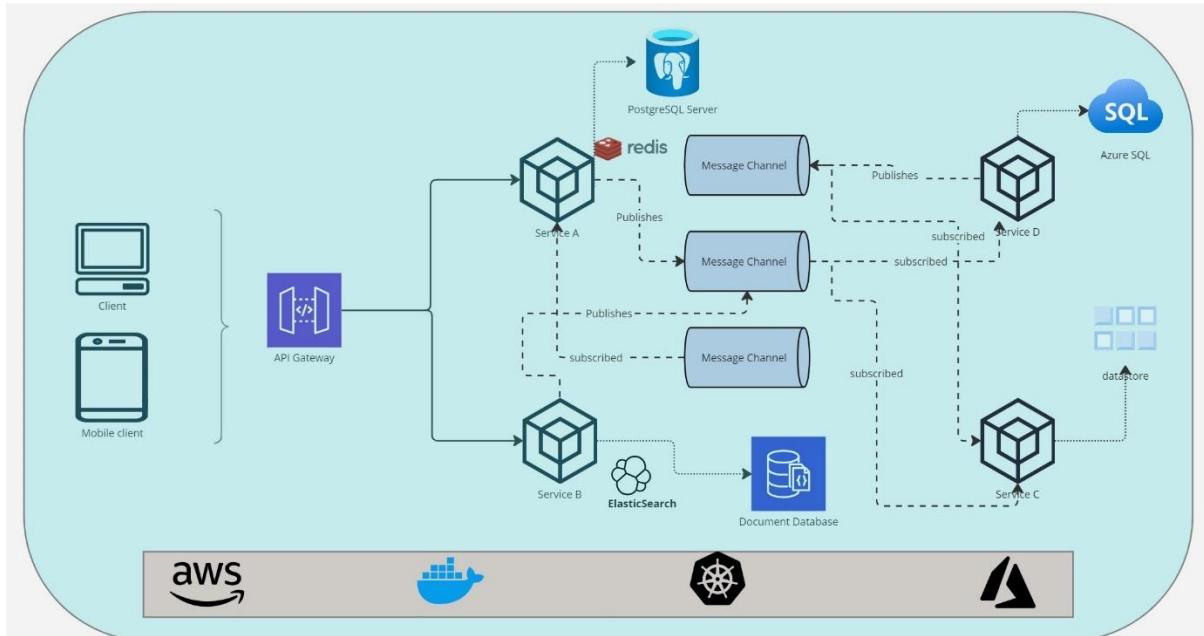


Figure 9-0-3 Event based Microservices architecture

In the earlier two architectures, services or service layers have been communicating via API interfaces. We observed the potential downside of this mode of communication when system gets complexed. Synchronous communication at times tend to become point of failure and thus affecting all other entities in the chain of action. As a request goes through multiple business layers (Layered Architecture) or passes through a bunch of services (Microservices), sender of the request goes into waiting state unless proper response is received. And if any service, in the chain of requests, is unavailable or down or there is some exception not handled in the code, request is bound for failure. As system grows complex, chances of this scenario become highly probable. Hence, software architects decided to leverage asynchronous communication in enterprise application development.

Story of TechCube continues as they keep adapting and innovating to maintain their edge among its competitors in Software industry.

After implementing successful microservice based project for a client, TechCube took on a massive project for a financial institution that required seamless integration between different systems and services. They followed a traditional Microservices architecture, where services communicated with each other through synchronous API calls. As the system evolved, they noticed a recurring issue: a single point of failure that affected the entire chain of actions.

When a request passed through multiple services, it would trigger a synchronous call to the next service, and the sender of the request had to wait for a response. If any service in the chain was unavailable or experienced an exception, it caused a ripple effect, leading to cascading failures. The system became brittle and prone to service outages, impacting their client's business operations.

Recognizing the limitations of synchronous communication, the software architects at TechCube began to explore alternative approaches. They discovered the power of Event-Driven Architecture (EDA) and its potential to revolutionize their development approach.

Coincidentally, around the same time, TechCube was approached by MelodyMasters, a renowned music event management company. MelodyMasters was planning to launch a new music festival streaming platform, StreamBeat, which aimed to provide live music experiences to fans worldwide.

MelodyMasters shared the challenges they were facing in handling real-time interactions, scaling to accommodate a massive audience, and providing seamless experiences during live streaming. The TechCube team realized that the very challenges faced by MelodyMasters were similar to those they encountered in their financial project.

TechCube saw this as an opportunity to introduce a paradigm shift in their development approach. They decided to transition to Event-Driven Architecture (EDA) to meet the growing requirements of their projects, including the StreamBeat platform.

TechCube's team started decoupling their services, breaking down their API-driven interactions into independent components that could produce and consume events. They implemented Kafka as a robust message queue, acting as the central communication channel. Kafka allowed services to publish messages and enabled other services to consume them asynchronously.

Different services in the system became message producers, generating messages based on specific actions or state changes. Meanwhile, other services became message consumers, listening for relevant messages and responding accordingly. Asynchronous communication provided fault tolerance. If a service became unavailable or experienced an issue, the system could continue functioning without being affected by the failing component.

For the StreamBeat platform, EDA allowed MelodyMasters to provide real-time updates to users during live streaming events, offering an engaging and dynamic user experience.

With Event-Driven Architecture at the core of their projects, TechCube experienced a transformative journey. The StreamBeat platform became an unprecedented success, captivating music enthusiasts worldwide with its seamless streaming experiences and interactive features. As they applied EDA to their financial project, the once brittle system evolved into a robust, scalable, and resilient architecture. The systems were now capable of handling high loads and maintaining responsiveness, even during peak hours.

EDA became a cornerstone of TechCube's enterprise application development, driving innovation and empowering them to tackle even the most complex challenges with grace.

In early days of EDA, **event bus** was used as mode of communication. An event bus is a central communication channel that facilitates the distribution of events to interested consumers within an application or system. It allows event producers to publish events without knowing the specific consumers, promoting loose coupling between components. But it had some drawbacks which were improved by usage of **message queues** like Kafka, RabbitMQ, etc.

An **event bus** typically acts as a central broker for events, managing the distribution of events to multiple consumers. It might use various transport mechanisms like publish-subscribe or topic-based messaging. In contrast, a **message queue** explicitly follows a message-based communication model, where messages are sent to specific queues and consumed by consumers that explicitly listen to those queues. Message queues, like Kafka and RabbitMQ, also offer message persistence, ensuring that messages are not lost even if a consumer is not immediately available. This durability is crucial in scenarios where data loss is unacceptable. Message queues can guarantee message ordering within a partition or queue, which is essential for maintaining event order in certain event-driven scenarios. Message queues are designed to handle high-throughput message processing and can scale horizontally to accommodate growing demands.

10.3.1. Pattern Description

Event-Driven Architecture (EDA) is an architectural pattern that enables the design of highly scalable, flexible, and loosely coupled software systems. Unlike traditional architectures that rely on synchronous communication and tight coupling between components, EDA emphasizes asynchronous communication through events, promoting better responsiveness, modularity, and fault tolerance. In an event-driven system, components communicate by producing and consuming events, which represent meaningful occurrences or state changes within the system.

10.3.1.1. Pattern Topologies

Event-Driven Architecture (EDA) can be implemented using various topologies to facilitate communication between event producers and consumers. Two common topologies are the **Mediator Topology** and the **Broker Topology**. Each topology has its characteristics, benefits, and use cases.

In the **Mediator Topology**, the communication between components is centralized through a mediator component. The **mediator** acts as an intermediary that receives events from event producers and dispatches them to the appropriate event consumers. The mediator determines which events are sent to which consumers based on its knowledge of registered consumers and their interests. Event producers and consumers do not need to know about each other directly, improving component independence.

When a client generates an event, it sends the event to an **Event Queue**. The Event Queue acts as a message buffer that temporarily stores the event until it is picked up and processed by the Event Mediator. The **Event Mediator** serves as the central orchestrator in the Mediator Topology. It receives the initial event from the Event Queue and is responsible for managing the event flow throughout the system. The Event Mediator determines which steps of the process need to be executed and in what order. It may decide to send additional asynchronous events to Event Channels to execute each step of the process. **Event Channels** are the pathways through which the Event Mediator communicates with different Event Processors. Each Event Channel is associated with a specific step or task in the event processing flow. **Event Processors** are the components that listen on the Event Channels and receive the events sent by the Event Mediator. Each Event Processor is responsible for executing specific business logic to process the event. The processing may involve data manipulation, calculations, updating databases, or triggering further actions. By having multiple Event Processors for different Event Channels, the system can distribute the processing load and handle events concurrently.

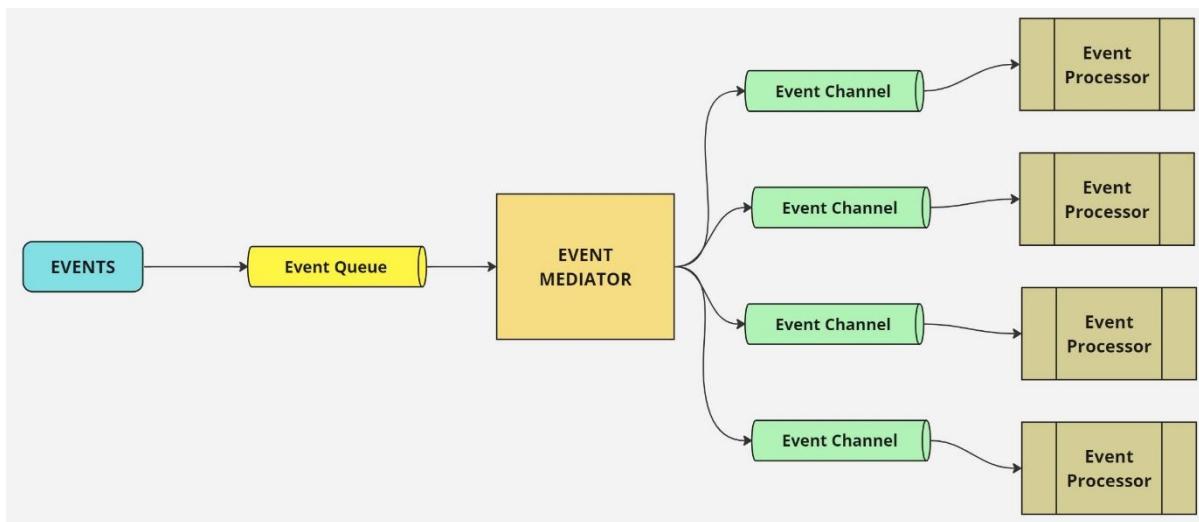


Figure 9-0-4 Mediator Topology

This topology promotes loose coupling between components and enables a more dynamic and flexible system. Components can interact indirectly through the mediator without direct dependencies, making it easier to add or remove components without affecting the entire system.

In the **Broker Topology**, communication between event producers and consumers is indirect, facilitated by a central **message broker**. The message broker acts as an intermediary that receives events from producers and forwards them to the appropriate consumers based on subscriptions or routing rules. Event producers publish events to the message broker, and consumers subscribe to specific topics or queues based on their interests. Unlike the mediator topology, components interact indirectly without a central hub, making the system more decentralized. This topology promotes scalability and fault tolerance by enabling asynchronous and decoupled communication.

The **Broker Component** is the central hub in the Broker Topology. It is responsible for handling event distribution and management of event channels. The broker can be centralized, where all events are managed by a single broker instance, or federated, where multiple broker instances work together to manage the events. The Broker Component contains all the **event channels** that are used for event flow. These event channels can be implemented as message queues, message topics, or a combination of both, depending on the messaging system in use. **Event processors** subscribe to specific event channels within the broker to receive events relevant to their functionalities.

The choice of topology depends on the specific requirements and characteristics of the application. Both topologies have their strengths and use cases. Mediator Topology is suitable for scenarios where a central orchestrator is beneficial for event coordination, dynamic interaction management, and centralized control. Broker Topology is well-suited for decentralized systems with asynchronous communication, scalable event processing, and the need for indirect interactions between components.

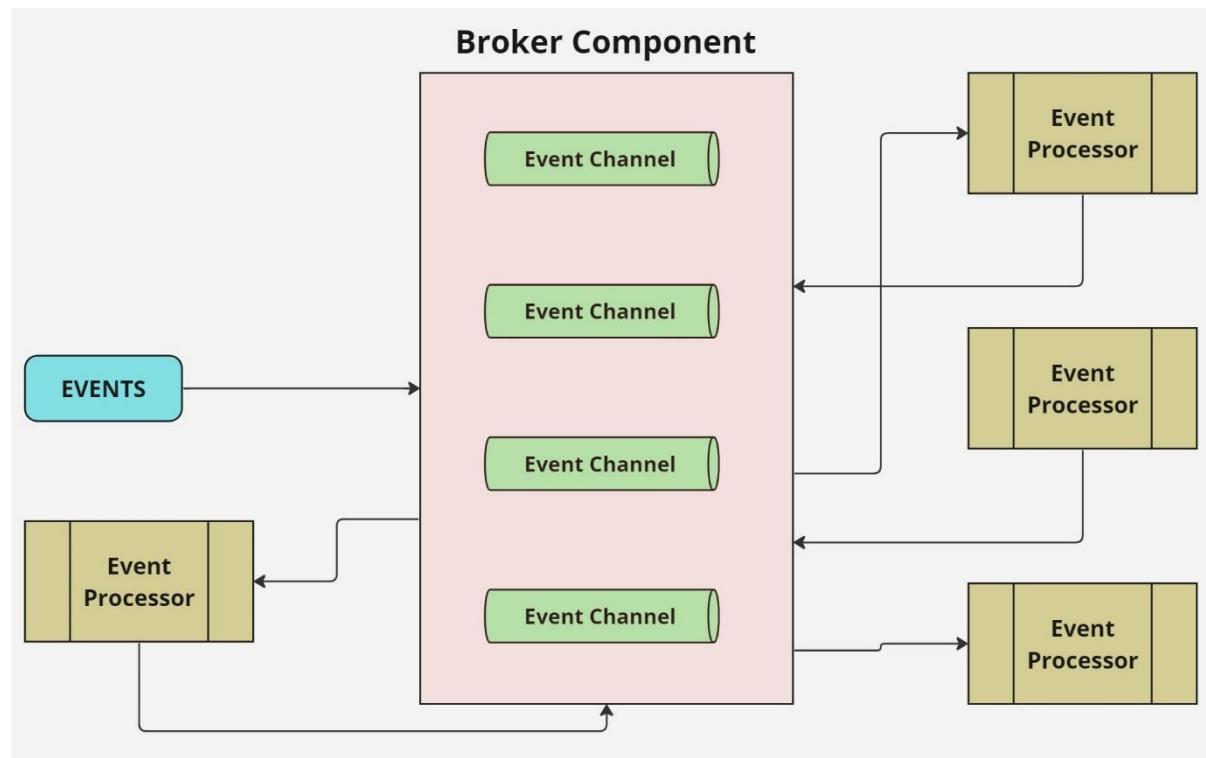


Figure 9-0-5 Broker Topology

In practice, some applications may employ a combination of both topologies to achieve the desired level of centralization and decentralization based on different functional areas and requirements. The flexibility of Event-Driven Architecture allows for a variety of topology designs tailored to the unique needs of the application.

10.3.1.2. Use of Message Queues in EDA

A message queue can act as both a mediator and a broker depending on the context and the design of the system. The roles of a mediator and a broker in the context of a message queue are not mutually exclusive, and a message queue can fulfil the responsibilities of both roles simultaneously. Let us take a scenario of an e-commerce application based on EDA.

Imagine an e-commerce platform that needs to process incoming customer orders, handle inventory management, and notify shipping services for order fulfilment. We'll consider a simplified scenario where three main components are involved:

- 1. Order Service: Responsible for receiving and processing customer orders.*
- 2. Inventory Service: Manages the inventory and updates product quantities after order processing.*
- 3. Shipping Service: Receives order details for fulfilment.*

The message queue acts as a mediator by facilitating indirect communication between the components. When a customer places an order, the Order Service will publish an "OrderPlaced" event to the message queue. The Inventory Service and Shipping Service will be subscribers to the "OrderPlaced" event.

Order Service (Producer): Publishes "OrderPlaced" event to the message queue when a customer places an order.

Inventory Service (Consumer): Subscribes to the "OrderPlaced" event to receive order details and update product quantities in the inventory.

Shipping Service (Consumer): Subscribes to the "OrderPlaced" event to receive order details for fulfilment.

The message queue acts as a mediator by ensuring that both the Inventory Service and Shipping Service receive the necessary order information without direct coupling between them. If the system later requires additional services to be notified about order events, they can be easily added as subscribers to the "OrderPlaced" event in the message queue.

The message queue acts as a broker by facilitating communication between the components through message routing. Let's consider a scenario where the Inventory Service and Shipping Service need to communicate with each other indirectly through the message queue.

Inventory Service (Producer): After updating the inventory, the Inventory Service publishes an "InventoryUpdated" event to the message queue.

Shipping Service (Consumer): Subscribes to the "InventoryUpdated" event to receive inventory update details.

In this case, the message queue acts as a broker by forwarding the "InventoryUpdated" event from the Inventory Service to the Shipping Service. The message queue handles the message routing, ensuring that the Shipping Service receives the inventory update information.

In the scenario discussed above, the message queue fulfils both the mediator and broker roles to facilitate communication between the components. As a mediator, it ensures loose coupling between the *Order Service, Inventory Service, and Shipping Service*, allowing them to interact indirectly through events. As a broker, it routes messages from the Inventory Service to the Shipping Service, enabling them to communicate indirectly. The choice of using a message queue as a mediator, a broker, or both depends on the system's requirements, the level of decoupling desired, and the complexity of communication between components.

10.3.2. Pattern Analysis

EDA comes with many benefits due to its leveraging of asynchronous way of communication. It allows for easy scaling of individual components. When the load on a specific service increases, additional instances of that service can be deployed to handle the increased event processing, ensuring efficient resource utilization. Events can trigger real-time updates to user interfaces and applications. Users receive instantaneous feedback and updates, leading to a more engaging and dynamic user experience. EDA is closely related to the Microservices architectural pattern. Microservices often adopt EDA to enable communication between various microservices without tight dependencies.

However, maintaining event order across distributed components can be challenging. Ensuring eventual consistency across the system is crucial to avoid data integrity issues. As the application evolves, event schemas may need to change. Handling event schema evolution and backward compatibility becomes important to avoid breaking existing consumers. The lack of atomic transactions for a single business process is a trade-off that comes with the distributed and decoupled nature of event-driven systems.

Since, each service processes events independently, and there is no central coordinating authority to enforce atomicity across services. As a result, ensuring that a sequence of events representing a single business process all succeed or fail together becomes challenging.

While EDA ensures high performance with ease in scalability, development of EDA in itself can be a challenging task to undertake. Developing and managing applications based on an event-driven architecture can introduce complexities due to its asynchronous nature and the need for handling potential failures effectively. Events might be processed out of order due to variations in event propagation time, leading to potential issues with data consistency. Developers must implement mechanisms to handle out-of-order processing. Event-driven systems often embrace eventual consistency, where services may eventually reach a consistent state after processing events. This may require additional logic to handle temporary inconsistencies. Developers need to be considerate of potential race conditions and thread safety to ensure proper handling of events while developing event handlers and call-backs. Use effective monitoring and observability tools to gain insights into system behavior, identify performance bottlenecks, and detect anomalies, becomes imminent step in EDA development. For complex business processes spanning multiple services the Saga pattern is considered to manage distributed transactions and ensure consistency.

In summary, Event-Driven Architecture is a powerful approach that facilitates the development of scalable, responsive, and loosely coupled systems. By embracing asynchronous communication and events, businesses can create flexible and adaptable software solutions that meet the demands of modern, distributed applications.

10.4. Summary

In this chapter, we explored three fundamental architectural paradigms that play a crucial role in shaping modern software systems: **Layered Architecture, Microservices Architecture, and Event-Driven Architecture (EDA)**. Layered Architecture provides a structured approach to software design, where

Objects, Data & AI: Build thought process to develop Enterprise Applications

components are organized into layers based on their responsibilities, promoting separation of concerns and maintainability. Microservices Architecture fosters a decentralized system, breaking down functionalities into small, independent services that can be developed, deployed, and scaled individually, allowing for flexibility and rapid iteration. Event-Driven Architecture embraces asynchronous communication and loose coupling through events, enabling responsiveness, scalability, and dynamic interactions. Each paradigm offers unique advantages, and the choice of architecture depends on the specific requirements and complexity of the application. As software architects and engineers, understanding these architectural paradigms empowers us to design robust, scalable, and adaptable systems that align with the ever-evolving needs of modern technology.

Rise of Artificial Intelligence (AI)

Rise of Big Data and Machine Learning

“

“Things get done only if the data we gather can inform and inspire those in a position to make a difference.” - Dr. Mike Schmoker

11.1. From static web pages to Social Media

In the 1990s and early 2000s, the growth of the World Wide Web brought about a significant need to store and query data on a scale never seen before. While relational databases were sufficient for managing enterprise data, they were ill-equipped to handle the vast volumes of information generated by web servers and user interactions. Initially, the web primarily consisted of static content, but behind the scenes, web servers were generating extensive log files containing valuable information about users' behaviors.

Web logs recorded details of users' interactions with web pages, providing insights into which pages were accessed, when, and by whom. This data proved to be a goldmine for companies looking to understand their users better and target advertisements and marketing campaigns effectively. By combining user profile data, such as age, gender, and income level, with web logs, businesses gained deeper insights into their target audience's preferences and behaviors. Transactional websites, such as online shopping platforms, added more data to the mix, including users' browsing and purchase histories.

The 2000s witnessed a meteoric rise in user-generated data, largely driven by the emergence of social media platforms. Social networks like Facebook, founded in 2004, Twitter in 2006, and LinkedIn in 2003, brought millions of users together on their respective platforms. Facebook alone attracted over 1.3 billion users worldwide, with approximately 800 million active users daily. Twitter boasted an estimated 980 million users by early 2014, producing a staggering one billion tweets per day as of October 2012.

As more users joined these platforms, a deluge of data was created in the form of status updates, tweets, photos, videos, and interactions. This avalanche of data led to the term "**Big Data**" gaining prominence. The term refers to the massive datasets that are too vast, complex, and unstructured to be processed and analysed using traditional database management systems and analytical tools.

The evolution of Big Data was fuelled not only by social media but also by the early search engines such as AltaVista and Lycos. Although these pioneers were eventually overshadowed by the likes of Google and Bing, they played a crucial role in laying the groundwork for handling the growing volume of web data.

11.2. Empowering the Data-Driven Enterprise

Big Data has emerged as a transformative force, offering an abundance of opportunities for businesses to glean valuable insights and make informed decisions. By harnessing the power of analytics, organizations can deliver timely information to consumers, enabling them to make data-driven choices, identify needs, and improve performance. IBM, in its book "Analytics Across the Enterprise: How IBM Realizes Business Value from Big Data and Analytics," highlights the significance of enterprise-wide big data analytics and its impact on shaping new products, services, and business models.

Descriptive and Predictive Analytics: The foundation of data analytics lies in descriptive and predictive analytics. Descriptive analytics focuses on reporting historical data and analysing past events to understand why they occurred. This knowledge helps organizations gain insights into their operations, customer

behavior, and market trends. Predictive analytics, on the other hand, employs statistical and data mining techniques to forecast future events and outcomes based on patterns in historical data. By leveraging predictive analytics, businesses can anticipate customer needs, optimize processes, and make proactive decisions to stay ahead in a competitive landscape.

Prescriptive Analytics: Going beyond prediction, prescriptive analytics is about recommending actions. By combining insights from descriptive and predictive analyses, organizations can develop decision support systems that suggest the best course of action to achieve desired outcomes. This empowers decision-makers with actionable intelligence and optimizes resource allocation for improved efficiency and effectiveness.

Social Media Analytics: The rise of social media platforms has generated vast amounts of user-generated content. Social media analytics involves sentiment analysis to gauge public opinion on various topics or events. Additionally, it allows businesses to discern behavioral patterns and preferences of individuals, aiding them in crafting personalized offerings. Understanding the sentiment and behavior of consumers enables industries to target goods and services in a customized manner, resulting in higher customer satisfaction and loyalty.

Entity Analytics: This emerging area focuses on consolidating and analysing data related to entities of interest, such as customers, products, or events. By aggregating information and using machine learning algorithms, entity analytics uncovers hidden insights about these entities. This knowledge is invaluable for businesses looking to optimize customer segmentation, improve supply chain management, or identify potential risks.

Cognitive Computing: Cognitive computing represents a cutting-edge field aiming to develop systems that can interact with humans to provide enhanced insights and advice. These systems leverage artificial intelligence, natural language processing, and machine learning to understand complex queries and deliver contextually relevant information. Cognitive computing empowers users with comprehensive data exploration, enabling them to make well-informed decisions with confidence.

The convergence of Big Data and analytics has brought forth a new frontier, promising a data-driven revolution across industries. As more businesses recognize the potential of analytics-oriented applications, productivity, quality, and growth are poised to soar. Organizations that embrace Big Data and analytics will gain a competitive edge, as they can unlock the hidden value within their data, extract meaningful patterns, and anticipate future trends. In this data-rich era, the path to success lies in leveraging analytics to drive innovation, improve customer experiences, and unlock new possibilities in the ever-evolving landscape of business.

In this digital age, Big Data continues to evolve, driven by emerging technologies such as the Internet of Things (IoT), artificial intelligence, and edge computing. The journey from static web pages to social media and beyond has created a new paradigm, where data has become the lifeblood of modern economies and societies, shaping the future of innovation and decision-making. As we venture further into this era of data abundance, the possibilities and challenges presented by Big Data are boundless, transforming how we understand the world and harness its potential for the greater good.

11.3. Characteristics of Big Data

The concept of Big Data is not merely defined by its sheer volume; it encompasses several distinguishing characteristics that set it apart from traditional data management practices. As we delve into the world of Big Data, we try to understand four fundamental traits that shape its uniqueness: **Volume, Variety, Velocity, and Veracity.**

Volume: The first hallmark of Big Data is its colossal volume. The datasets utilized in Big Data applications are vast, typically ranging in the petabyte (PB) scale, and are projected to reach the zettabyte (ZB) realm with the rise of Internet-of-Things (IoT) applications. To put this into perspective, Google reported that in 2016, daily user uploads to YouTube required a staggering 1PB of new storage capacity. This exponential growth shows no signs of abating, with storage additions expected to witness a 10x increase every five years. Alibaba's record is no less impressive, generating 320 PB of log data in a mere six-hour period due to customer purchase activity.

The Industrial Internet of Things (IIoT) or IoT is poised to usher in a transformative revolution, significantly impacting enterprises and their operational efficiency. One of the key factors contributing to the volume characteristic of Big Data is the massive influx of data generated by billions of connected devices. The ever-expanding network of IoT devices, ranging from sensors on factory floors to wearable devices and smart appliances, creates a constant stream of data, adding to the already vast repositories of information. The magnitude of data in Big Data applications demands robust infrastructure and scalable solutions to handle the deluge of information effectively.

Variety: Unlike traditional databases that predominantly deal with well-structured data and conform to rigid schemas, Big Data applications contend with a diverse range of data types. The data can encompass images, text, audio, video, and more, often referred to as multimodal data. A significant portion of generated data today is unstructured, making up approximately 90% of the dataset.

Structured data adheres to a formal data model, such as the relational model, where information is organized in tables with rows and columns, or in hierarchical databases like IMS, featuring record types and fields within records. On the other end of the spectrum lies unstructured data, which lacks an identifiable formal structure. Examples of unstructured data include e-mails, blogs, PDF files, audio, video, images, clickstreams, and web contents. The emergence of the World Wide Web in the 1990s catalysed a tremendous growth in unstructured data, presenting a significant challenge in today's Big Data systems.

Big Data systems must possess the versatility to manage and process these diverse data types seamlessly. The challenge lies in maintaining coherence and extracting valuable insights from data that lacks a predefined structure, fostering a need for innovative techniques in data processing and analysis.

Velocity: The speed at which data is generated and processed is another pivotal characteristic of Big Data. Some applications deal with data arriving at the system in real-time and at high-speed, demanding immediate processing. For instance, Facebook handles an astounding 900 million photos uploaded by users daily, while Alibaba had to process 470 million event logs per second during peak periods. These time-sensitive scenarios necessitate real-time capabilities, as data cannot be stored before processing. Systems must be agile enough to process incoming data swiftly and efficiently, enabling organizations to respond promptly to emerging trends and changing customer behaviors.

Veracity: Veracity addresses the trustworthiness and reliability of the data used in Big Data applications. Data from multiple sources may not always be entirely reliable or consistent, leading to noise, bias, and deliberate misinformation, aptly termed "dirty data." The presence of dirty data can have significant economic implications, costing billions of dollars in losses annually. Big Data systems need to implement mechanisms to clean and validate data while preserving its provenance. Ensuring data is trustworthy is vital in reasoning about its credibility and making informed decisions. Additionally, veracity entails safeguarding the "truthfulness" of data, guarding it against noise, bias, or intentional manipulation.

11.4. Storage Systems for Big Data

In the digital age, the explosive growth of data has given rise to applications with an immense user base, creating an urgent need for highly scalable storage solutions. Popular applications catering to hundreds of millions of users face the challenge of managing and processing vast volumes of data. To meet the demands of such applications, data management requires an innovative approach. Over the past two decades, several specialized storage systems have been developed and deployed to handle the storage and processing needs of Big Data applications. Let us look at a brief overview on these diverse storage systems and their applications.

Distributed File Systems: Distributed File Systems have emerged as a foundational solution for handling large files and supporting record storage. These systems allow files to be distributed across multiple machines while providing a familiar file-system interface for access. Their ability to efficiently store and manage large files, such as log files, makes them suitable for applications that generate massive amounts of data. Additionally, Distributed File Systems serve as the storage layer for systems supporting record storage. They are instrumental in storing records efficiently and ensuring data availability even under high loads. By distributing data across multiple nodes, these systems achieve high scalability and resilience, vital qualities for Big Data applications.

Sharding: Sharding is a crucial technique employed to partition records across multiple systems or databases. This process ensures that records corresponding to different users or entities are divided and distributed efficiently. In this scenario, each database acts as a traditional centralized database, unaware of other databases' contents. The responsibility of managing record partitioning lies with the client software, which routes queries to the appropriate database. This approach allows applications to handle a large number of users and diverse data sets effectively. Sharding contributes significantly to scalability, as it facilitates horizontal data scaling by distributing data across multiple nodes.

Key-Value Storage Systems: Key-Value Storage Systems present a unique approach to storing and retrieving data based on keys. These systems offer high-speed access to records by leveraging keys as identifiers for data retrieval. Additionally, they may provide limited query facilities, but they are not comprehensive database systems and, as such, are often referred to as NoSQL systems. These systems are optimized for handling massive volumes of simple data structures, making them well-suited for applications with real-time data requirements. Key-Value Storage Systems, with their ability to rapidly store and retrieve data, form an integral part of Big Data solutions that prioritize fast and efficient data access.

Parallel and Distributed Databases: Parallel and Distributed Databases bridge the gap between traditional database interfaces and the need for distributed data storage and processing. These systems offer the familiar database query interface while distributing data across multiple machines for enhanced scalability and performance. By performing query processing in parallel across multiple nodes, these databases can efficiently handle complex queries and high-concurrency workloads. They enable organizations to scale their data storage and processing capabilities seamlessly, making them an essential component of Big Data applications that require sophisticated querying and analytical capabilities.

11.5. Unravelling the Synergy: Big Data Analytics and Machine Learning

In the rapidly evolving landscape of modern technology, Big Data and Machine Learning (ML) have emerged as indispensable mechanisms for shaping the future of data-driven decision making. We have studied about the evolution of Machine Learning in our introductory chapter.

Big Data Analytics and Machine Learning have revolutionized how businesses harness the power of vast datasets to uncover valuable insights. Machine Learning acts as the catalyst that propels Big Data Analytics to new heights. Machine Learning is a subset of artificial intelligence that equips computer systems to learn from data and improve their performance without being explicitly programmed. By leveraging sophisticated algorithms, Machine Learning identifies patterns, makes predictions, and uncovers valuable insights from Big Data. These disciplines are inextricably linked, forming a symbiotic relationship that empowers organizations to navigate the complexities of the digital era.

The relationship between Big Data and Machine Learning is mutually beneficial. For Machine Learning algorithms to make accurate predictions, they require extensive training on large datasets. The massive volume of data offered by Big Data becomes the fuel that powers the learning process. As the algorithms ingest more data, they become more adept at recognizing complex patterns and making refined predictions, making them invaluable assets for businesses seeking accurate decision-making.

Big Data Analytics enriches the Machine Learning process by providing essential context and domain knowledge. By integrating diverse data sources, organizations can gain a comprehensive understanding of their operations, customers, and markets. For example, when forecasting stock prices, historical stock data can serve as the foundation for Machine Learning algorithms to identify patterns and correlations that impact stock movements. Such insights derived from Big Data Analytics enhance the predictive capabilities of Machine Learning models, enabling businesses to make informed and strategic choices.

Machine Learning algorithms play a vital role in data segmentation, labelling, and analytics, facilitating more efficient data processing at scale. Through scenario simulation, businesses can simulate potential outcomes based on various parameters, enabling them to optimize their strategies and mitigate risks effectively.

11.6. Anatomy of Machine Learning: Unveiling the Core of Artificial Intelligence

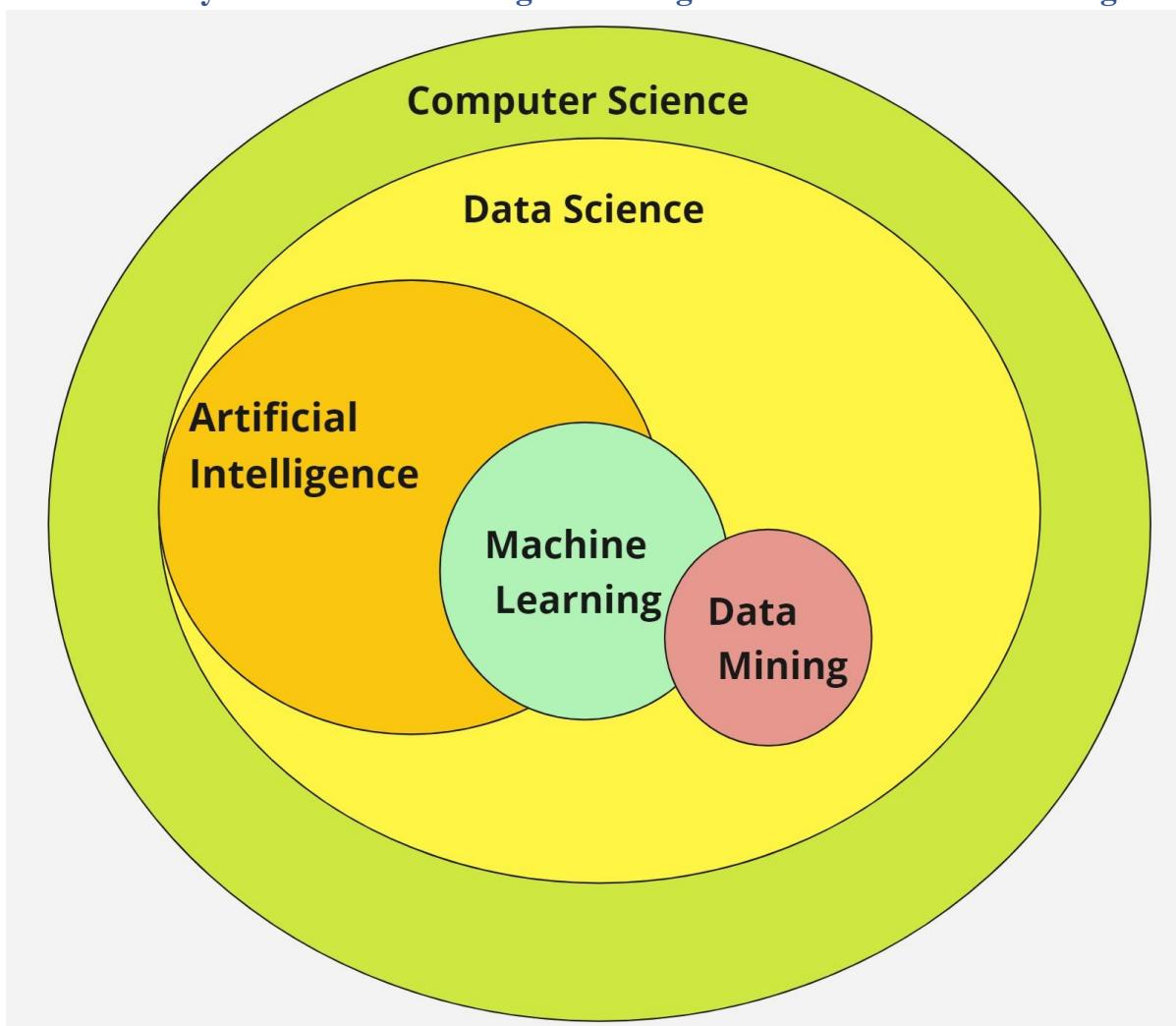


Figure 10-1 AI in Computer Science

Machine Learning, an integral subfield of Artificial Intelligence (AI), lies at the forefront of modern technological advancements. Rooted in computer science and data science, Machine Learning focuses on the development of algorithms and systems that enable machines to learn from data, improve their performance, and carry out intelligent tasks. Machine Learning emerged as a natural evolution from these disciplines, leveraging computational techniques to process vast datasets and derive meaningful patterns and predictions. By harnessing the power of inferential methods and probabilistic reasoning, Machine Learning unveils a practical lens of study that paves the way for AI's cognitive capabilities.

Machine Learning extends from the broader domain of AI, which revolves around machines simulating intelligent and cognitive tasks. Just as the Industrial Revolution ushered in machines to replicate physical tasks, AI now drives the development of machines capable of emulating cognitive abilities. Machine Learning serves as a crucial catalyst for AI's growth, providing the means for machines to learn and adapt from data, thereby exhibiting intelligent behavior.

11.6.1. Machine Learning and Data Mining

Within Machine Learning, there exists an overlap with **Data Mining**—a sister discipline that revolves around discovering patterns in large datasets. Both disciplines share inferential methods, where predictions are made based on existing outcomes and probabilistic reasoning. Algorithms such as principal component analysis, regression analysis, decision trees, and clustering techniques find applications in both Machine Learning and data mining. This synergy allows Machine Learning to draw from a rich repertoire of techniques and extend its reach to various domains, including perception and natural language processing. Despite their similarities and overlapping techniques, they serve different purposes and exhibit varying characteristics. Understanding the key differences between Machine Learning and Data Mining is essential for leveraging their respective strengths and applications effectively.

One of the key distinctions between the two disciplines lies in their autonomy and learning approach. Machine Learning emphasizes self-learning through exposure to data, allowing machines to adapt and improve their performance iteratively. In contrast, Data Mining is a less autonomous technique focused on extracting hidden insights from data using predefined algorithms and methods.

Data Mining primarily focuses on analysing input variables to predict new output variables or discover associations between them. In contrast, Machine Learning extends its scope to analyse both input and output variables. This includes **supervised learning** techniques, which use known combinations of input and output variables to discern patterns and make predictions.

Additionally, **reinforcement learning** in Machine Learning randomly trials a massive number of input variables to produce desired output, aiming to optimize performance through rewards and penalties. **Unsupervised learning** is a technique within Machine Learning that generates predictions based on the analysis of input variables with no known target output. While this technique can overlap with data mining, it deviates from standard data mining methods like association and sequence analysis. Unsupervised learning is often used in combination or preparation for supervised learning, referred to as **semi-supervised learning**.

In summary, Machine Learning's emphasis on self-learning and the analysis of both input and output variables differentiates it from Data Mining's focus on extracting insights from input variables to predict new output variables or discover associations.

11.7. Summary

The rise of Big Data has transformed the landscape of data management and analysis for enterprises worldwide. This chapter explored the characteristics and storage systems associated with Big Data, emphasizing the need for scalable and efficient solutions to handle the vast amount of data generated by applications with millions of users. To unlock the potential insights buried within this sea of data, Big Data analytics emerges as a powerful tool, and it is here that Machine Learning, a subset of Artificial Intelligence, plays a pivotal role.

As we delve into the subsequent chapters, our focus will shift towards understanding the concepts of Machine Learning, exploring various algorithms, and diving into data engineering practices that enable the efficient utilization of Machine Learning techniques. Machine Learning empowers computers to learn from data, improve their performance, and make predictions without being explicitly programmed. Its application in the realm of Big Data analytics becomes paramount in extracting valuable patterns, trends, and correlations hidden within the massive datasets.

With an emphasis on self-learning and adaptability, Machine Learning offers a practical and narrower lens of study in comparison to the broader realm of Artificial Intelligence. The discussion revealed that Machine

Objects, Data & AI: Build thought process to develop Enterprise Applications

Learning overlaps with data mining, sharing inferential methods and a similar assortment of algorithms. However, its unique focus on analysing both input and output variables set it apart, allowing for supervised, unsupervised, and reinforcement learning techniques.

Towards the conclusion of this section, we shall venture into a brief overview of Generative AI, a fascinating branch of Artificial Intelligence that has experienced significant growth in recent years. Generative AI empowers machines to create new data, including images, text, and even music. This emerging field promises to push the boundaries of creativity and human-machine collaboration, opening new possibilities across various industries.

Machine Learning: A Brief Introduction

“

“Machine learning is the field of study that gives computers the ability to learn without being explicitly programmed.” - Arthur Samuel

Learning is a multifaceted phenomenon that encompasses various processes, from acquiring new knowledge to developing cognitive and motor skills through instruction and practice. Additionally, learning involves organizing newly acquired knowledge into effective representations and, remarkably, discovering new facts and theories through observation and experimentation. In the era of computers, researchers have ardently pursued the ambitious goal of instilling such learning capabilities in machines, thus giving birth to the fascinating domain of **Artificial Intelligence (AI)**.

As researchers sought to unlock the potential of computers to learn and adapt, the study and computer modelling of diverse learning processes emerged as the focal point of a captivating discipline: **Machine Learning**. This field delves into the intricacies of enabling computers to mimic and execute various manifestations of learning, encompassing the acquisition of declarative knowledge, the refinement of skills, the formation of general representations, and the discovery of new insights.

Machine learning remains a profoundly challenging and evolving pursuit in the realm of AI. The quest to endow computers with the capacity to learn from data and experiences has sparked unparalleled innovation and ground-breaking advancements. With the advent of modern computing technologies, machine learning has blossomed into a transformative force, revolutionizing industries, from healthcare and finance to autonomous vehicles and natural language processing.

As we traverse the landscapes of algorithms, models, and applications, we uncover the true essence of machine learning—an awe-inspiring domain that continues to push the boundaries of human-computer interaction and redefine the possibilities of AI.

12.1. Stepping away from Explicit Programming

Arthur Samuel, a pioneer in the field of Machine Learning, succinctly captured the difference between Machine Learning and Explicit Programming with his famous quote:

“Explicit programming is telling the computer exactly what to do, while Machine Learning is training the computer to learn from data and improve its performance over time.”

Arthur Samuel's definition highlights the essence of Machine Learning's self-learning aspect. He observed that machines do not need to be explicitly programmed for every task they perform. Instead, they rely on the data provided during the training process to learn from examples and derive insights. This learning process enables them to autonomously adjust their internal parameters and models, allowing them to adapt and improve their performance over time.

The distinction between upfront programming and data-driven learning is key to understanding the power and potential of Machine Learning. In traditional explicit programming, the programmer must anticipate all possible scenarios and provide explicit instructions for each one. This approach can become complex and impractical for tasks with vast and dynamic data sets. On the other hand, Machine Learning allows

computers to analyse data, find patterns, and make informed decisions based on that data, without the need for manual coding of all possible scenarios.

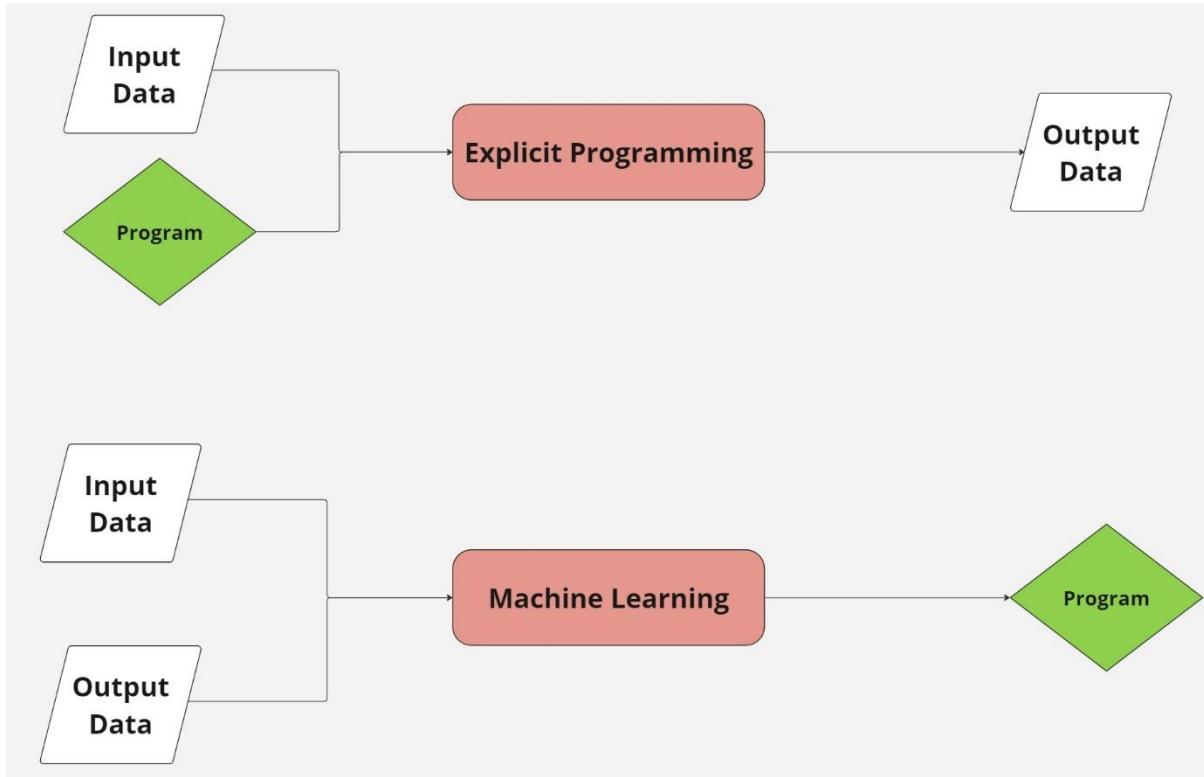


Figure 11-0-1 Programming vs ML

In explicit programming, programmers write specific instructions and rules that dictate the computer's behavior in handling various tasks. These instructions are fixed and do not change unless manually updated by the programmer. The computer follows these predefined rules and executes tasks accordingly. This approach works well for problems with clear-cut solutions and well-defined rules.

On the other hand, Machine Learning takes a different approach. Instead of explicitly programming every rule and instruction, Machine Learning algorithms are trained on large datasets, allowing them to learn patterns and relationships from the data. The algorithms iteratively refine their performance by adjusting their internal parameters based on the examples they encounter during training. Once the training process is complete, the Machine Learning model becomes capable of making predictions and decisions based on new, unseen data.

The concept of **self-learning** is a fundamental and defining feature of Machine Learning. It refers to the ability of machines to improve their performance and make decisions based on data and empirical information, without the need for explicit programming commands.

However, it is crucial to understand that while Machine Learning does not require direct programming commands for specific tasks, it is still heavily dependent on computer programming. Programmers play a vital role in designing and implementing Machine Learning algorithms, selecting appropriate models, and defining the parameters and hyperparameters of the algorithms. The programming aspect comes into play during the development, training, and evaluation phases of the Machine Learning process.

In traditional programming, the focus is on code. In machine learning, the focus shifts to representation.

12.2. Data & Models

What we commonly refer to as "data" represents a collection of observations derived from real-world phenomena, offering valuable insights into various aspects of reality. These observations range from stock market data, encompassing daily stock prices and earnings announcements, to personal biometric data, capturing minute-by-minute measurements of heart rate, blood sugar levels, and blood pressure. Each piece of data acts as a small window into a limited aspect of reality, and when aggregated, these observations construct a mosaic that forms a more comprehensive picture of the whole.

The amalgamation of data from diverse sources introduces complexity and noise. The real-world nature of data leads to measurement errors and missing information, making the data landscape inherently messy. Machine learning techniques, though powerful, need to navigate this messiness and make sense of the diverse data points to generate meaningful insights and predictions.

In machine learning, data plays a pivotal role in the training and evaluation of models. The data is divided into two essential components: **training data and test data**.

The initial reserve of data used to develop and fine-tune the model is termed **training data**. For instance, in the context of spam email detection, the training data would include examples of both spam and non-spam emails. During the training phase, the model learns from these examples and establishes patterns and rules to distinguish between spam and non-spam emails.

Think of a **model** as a mathematical framework that maps input data to output predictions or decisions. A model refers to a mathematical representation or algorithm that describes the relationships between different aspects of the data. It serves as the core component of the learning process, as it enables the system to understand patterns, make predictions, or perform tasks based on the given data.

A model predicting stock prices uses a formula that considers factors like a company's earning history, past stock prices, and industry to estimate the future stock price. Similarly, a music recommendation model assesses the similarity between users based on their listening habits and recommends artists or songs that align with the preferences of similar users.

The process of building a model involves training it with a vast dataset, where the model learns from examples and adjusts its internal parameters to better fit the data. During this **training phase**, the model refines its ability to make accurate predictions or recommendations based on the relationships it discovers within the data. A well-developed model should demonstrate good generalization, meaning it can make reliable predictions or recommendations on new, unseen data beyond the training set. It should be capable of capturing underlying patterns and trends without overfitting to noise or irrelevant details present in the training data.

However, the learning process is not always perfect. False positives and false negatives may occur during the training phase, where certain legitimate emails might be erroneously categorized as spam or vice versa, songs recommended to listeners are not the kind of songs they like, etc. To address these errors, modifications and additional rules need to be integrated into the model to enhance its accuracy and minimize misclassifications.

Once the training phase is complete, the model is tested on the remaining data known as **test data**. This data serves as an independent evaluation set, allowing the machine learning model to demonstrate its proficiency in making accurate predictions on new, unseen data. By assessing the model's performance on the test data, practitioners gain valuable insights into its generalization ability and ensure that it can effectively filter incoming emails and make reliable decisions on email categorization.

12.3. Classification of Machine Learning Systems

Machine Learning offers a multitude of approaches and techniques to empower computers to learn and make informed decisions. Within this vast landscape, we discuss three key dimensions for classifying machine learning systems:

12.3.1. Learning Strategies

The first dimension of classification is based on the underlying learning strategies used by the systems. These strategies vary in terms of the amount of inference and data processing performed by the learning system. At one end of the spectrum, we have **supervised learning**, where the system is provided with labelled training data, and the learning process involves inferring patterns and relationships from this labelled information. **Unsupervised learning**, on the other hand, operates without labelled data and focuses on discovering patterns and structures within the data on its own. **Reinforcement learning** lies in between, employing a trial-and-error approach where the system learns from feedback in the form of rewards or penalties, progressively refining its decision-making abilities. Let us discuss some of the key learning strategies in more detail.

12.3.1.1. Supervised Learning

Imagine a child learning to identify fruits from parents. Initially parents show various fruits from a basket to the child and tell what each one is called (labelling). Child's brain tries to learn the distinct characteristic of each fruit. They observe the colours, shapes, and other features that distinguish an apple from an orange or a banana. Parents do this process repeatedly, as during the initial stages, a child brain does not process the information correctly and it takes some practice and effort to hone its ability. As child repeatedly practices identifying fruits, his/her brain refines its ability to distinguish between different fruits. If child misidentifies an orange as an apple, he/she will receive feedback from parents, and they will learn from this feedback to improve their accuracy. With time, child's cognitive ability improves, and classification task is performed with better ability. Parents try to present child with fruits baskets containing some new fruits, so that child's knowledge of fruits keeps improving from time to time.

The primary objective of supervised learning is to learn a mapping or relationship between input features (also known as predictors) and output labels, allowing the model to make predictions on new, unseen data.

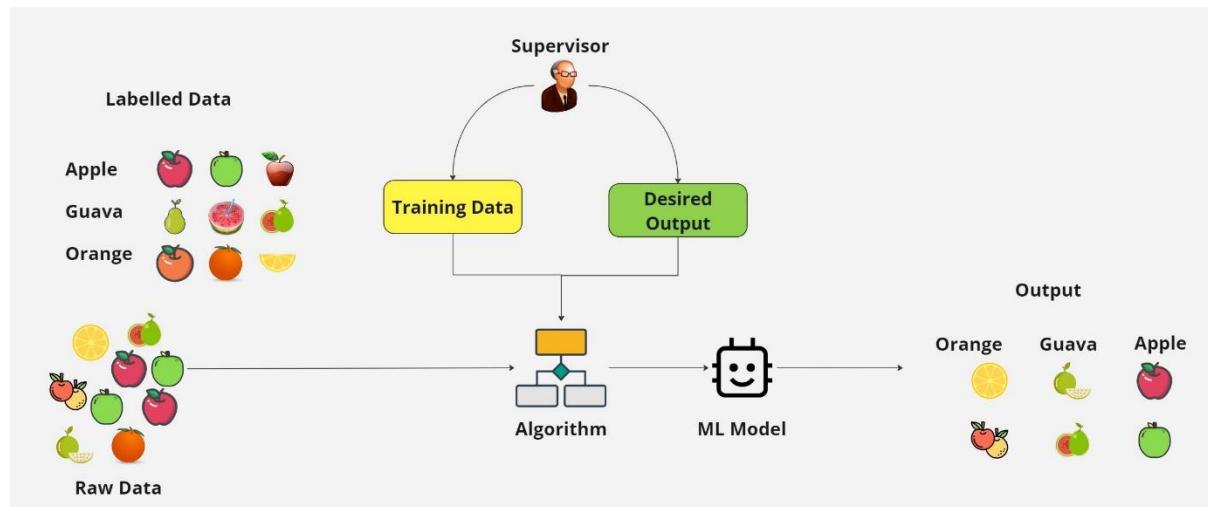


Figure 11-0-2 Supervised Learning

In supervised learning, the ML model is trained on a labelled dataset, where each data point is associated with a corresponding target or label. The model learns to map input features to their corresponding output labels, allowing it to make predictions on new, unseen data.

The labelled dataset used in supervised learning consists of paired examples, where each example consists of input features and the corresponding known output label. For example, in a spam email classification task, the input features might include the email's text and sender, while the output label is a binary value indicating whether the email is spam or not. During the training phase, the supervised learning model processes the labelled data and learns from the patterns and relationships between input features and output labels. The model iteratively adjusts its internal parameters to minimize the difference between its predicted output and the true output labels in the training data.

To measure the discrepancy between the predicted output and the true label, supervised learning utilizes a **loss function** (also known as a cost function or objective function). The goal of the training process is to minimize this loss function, effectively reducing the model's prediction errors. After the training phase, the model is equipped to make predictions on new, unseen data. It takes the input features of the new data and generates a predicted output label based on the relationships learned during training. The performance of the supervised learning model is evaluated on a separate, independent dataset, which is the test set, as discussed above. This evaluation set ensures that the model's ability to generalize to unseen data is tested. Metrics such as accuracy, precision, recall, and F1 score are commonly used to assess the model's performance.

12.3.1.1.1. Popular Supervised Learning based algorithms

- Linear Regression: Linear regression is used for predicting a continuous numeric value. It finds the best-fitting linear relationship between the input features and the target variable.
- Logistic Regression: Logistic regression is used for binary classification, where the goal is to predict one of two possible classes based on input features.
- Support Vector Machines (SVM): SVMs are versatile algorithms used for both classification and regression tasks. They find the optimal hyperplane that best separates classes or fits a regression line while maximizing the margin.
- Decision Trees: Decision trees are tree-like structures that recursively split the data into subsets based on input features. They can be used for both classification and regression tasks.
- Random Forest: A random forest is an ensemble of decision trees, where multiple trees are trained on different subsets of data and their predictions are combined to make a final prediction. It's often used for improving prediction accuracy and handling overfitting.
- Gradient Boosting: Gradient boosting is another ensemble technique that combines weak learners (usually decision trees) into a strong model. It builds trees sequentially, focusing on correcting the mistakes of the previous trees.
- Naive Bayes: Naive Bayes is a probabilistic algorithm based on Bayes' theorem. It's commonly used for text classification tasks, such as spam detection or sentiment analysis.
- K-Nearest Neighbours (KNN): KNN is used for classification and regression tasks. It makes predictions based on the majority class or the average of the k-nearest neighbours in the feature space.
- Neural Networks: Neural networks consist of interconnected nodes (neurons) that mimic the structure of the human brain. They are used for various tasks, including image and speech recognition, natural language processing, and more.

- Support Vector Regression (SVR): Like SVMs for classification, SVR is used for regression tasks. It aims to find a hyperplane that fits within a specified margin of error around the training data points.

These algorithms cover a wide range of tasks, from simple linear relationships to complex patterns in high-dimensional data. Supervised learning can be applied in various domains:

- Image and Object Recognition: Classifying objects in images or identifying objects' attributes based on labelled training data.
- Natural Language Processing: Tasks such as sentiment analysis, language translation, and named entity recognition.
- Speech Recognition: Converting spoken language into text, enabling voice assistants and speech-to-text applications.
- Medical Diagnostics: Predicting diseases and diagnoses based on patient data and medical records.
- Financial Prediction: Forecasting stock prices, predicting credit risks, and fraud detection.
- Autonomous Vehicles: Enabling self-driving cars to recognize objects and make decisions based on sensor data.

12.3.1.2. Unsupervised Learning

Remember the kid from last example, who was learning to classify fruits. Parents of the kids now assign the kid a task of clustering items from a grocery basket, to further improve cognitive skills of the child. Child is presented with a cart filled with groceries, including fruits, vegetables, dairy products, and snacks, but they are all mixed, and kid doesn't know which category each item belongs to (unlabelled dataset). Child starts observing the characteristics of each grocery item, such as its appearance, texture, and nutritional properties. He/she may notice that apples and oranges share similar color and shape, while carrots and cucumbers are both elongated vegetables.

Child's brain implicitly employs a "clustering algorithm" to group the groceries based on their similarities. It tries to find patterns in the data without any predefined labels, forming clusters of items that are more similar to each other than to items in other clusters. As child continues observing and thinking about the groceries, brain starts discovering clusters of similar items. Child identifies a cluster for fruits, another for vegetables, and additional clusters for dairy products and snacks. Finally, kid starts organizing the groceries into separate groups based on the clusters discovered.

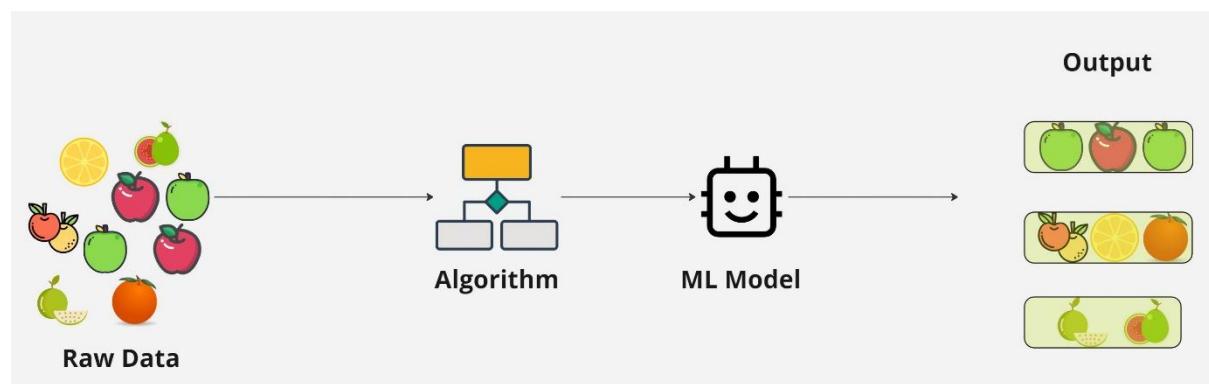


Figure 11-0-3 Unsupervised Learning

In unsupervised learning, the ML model is presented with an unlabelled dataset, and it seeks to identify inherent patterns and structures within the data. The goal is to uncover hidden relationships and group data

points based on similarity or other criteria. Unsupervised learning is commonly used for tasks like clustering, dimensionality reduction, and anomaly detection, where the model must learn from the inherent structure of the data without explicit guidance.

12.3.1.2.1. Popular Unsupervised Learning Algorithms

- K-Means Clustering: K-means is a popular clustering algorithm that groups similar data points into clusters. It aims to minimize the distance between data points within the same cluster while maximizing the distance between different clusters.
- Hierarchical Clustering: Hierarchical clustering builds a tree-like structure (dendrogram) to represent the relationships between data points. It can create clusters of different sizes and shapes based on the similarity between data points.
- DBSCAN (Density-Based Spatial Clustering of Applications with Noise): DBSCAN identifies clusters based on data density. It's effective in discovering clusters of varying shapes and sizes and can also identify noise points.
- Principal Component Analysis (PCA): PCA is a dimensionality reduction technique used to transform high-dimensional data into a lower-dimensional representation while retaining as much variance as possible. It helps uncover the most important features or patterns in the data.
- t-SNE (t-Distributed Stochastic Neighbour Embedding): t-SNE is another dimensionality reduction technique that is particularly useful for visualizing high-dimensional data in a lower-dimensional space. It emphasizes preserving pairwise similarities between data points.
- Autoencoders: Autoencoders are neural network architectures used for unsupervised learning. They are trained to encode and decode data, effectively learning compact representations of the input data. They are used for tasks like feature extraction and anomaly detection.

Unsupervised learning finds applications in various real-world scenarios, such as:

- Customer Segmentation: In marketing, unsupervised learning can be used to segment customers based on their shopping behavior, preferences, or demographics.
- Anomaly Detection: Unsupervised learning can help detect unusual patterns or anomalies in data, such as fraudulent transactions in financial transactions.
- Image Clustering: In image processing, unsupervised learning can group similar images together based on visual features, leading to image clustering.
- Topic Modelling: In natural language processing, unsupervised learning can be used to discover latent topics in a collection of documents.

12.3.1.3. Semi-Supervised Learning

In real-world scenarios, obtaining labelled data can be time-consuming, costly, or even impractical due to various reasons. However, unlabelled data is often readily available in abundance.

Let's consider the task of sentiment analysis for customer reviews in the e-commerce industry. Companies often receive numerous customer reviews but labelling each review as positive or negative sentiment can be resource-intensive.

In a semi-supervised learning approach, the model can be trained on a subset of labelled reviews where the sentiment is explicitly known (e.g., reviews with star ratings). However, most reviews remain unlabelled. Using the partially labelled data, the model can infer the sentiment of the unlabelled reviews based on patterns and context in the labelled reviews.

For instance, if the model has seen multiple reviews with positive sentiment that mention terms like "excellent," "great," or "highly recommend," it can infer that similar language in the unlabelled reviews likely indicates positive sentiment. By pseudo-labelling the unlabelled reviews with these inferred sentiments, the model can leverage the larger pool of data to refine its sentiment analysis capability.

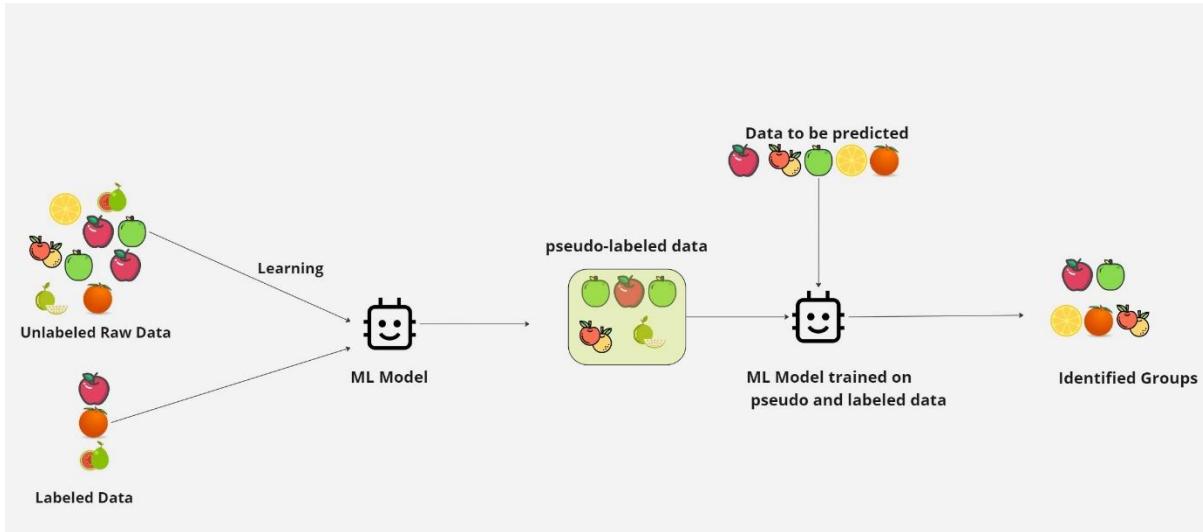


Figure 11-0-4 Semi Supervised Learning

Semi-supervised learning lies at the intersection of supervised and unsupervised learning. It uses a combination of labelled and unlabelled data during training. The model leverages the labelled data to learn from explicit feedback while also utilizing the vast amounts of unlabelled data to capture underlying patterns and enhance its performance. Unlabelled data often contains valuable but "hidden" information that can contribute to better generalization and improved model performance. Semi-supervised learning algorithms aim to extract relevant information from the unlabelled data and incorporate it into the learning process. One approach in semi-supervised learning involves using the model's own predictions on unlabelled data to create "**pseudo-labels**." These pseudo-labels are then treated as if they were true labels during the training phase, allowing the model to learn from the additional information present in the unlabelled data.

Semi-supervised learning is particularly valuable when obtaining large labelled datasets is challenging or costly. It enables models to learn from a larger pool of data, leading to improved generalization and potentially higher accuracy compared to models trained solely on labelled data.

12.3.1.3.1. Popular Semi-supervised Learning Algorithms

- **Self-Training:** Self-training is a simple technique where a model is initially trained on a small labelled dataset. The model is then used to make predictions on unlabelled data, and the high-confidence predictions are added to the labelled dataset. The model is retrained using the expanded labelled dataset, and the process iterates.
- **Co-Training:** Co-training involves training two separate models on different sets of features or views of the data. Each model makes predictions on unlabelled data, and their predictions are used to label the data. The labelled data is then used to improve both models through iterative training.

- **Multi-View Learning:** Multi-view learning combines information from different views or representations of the same data to improve performance. Each view can provide complementary information, enhancing the model's ability to learn from limited labelled data.
- **Expectation-Maximization (EM) Algorithm:** EM is an iterative optimization algorithm used for probabilistic models. In semi-supervised learning, it can be used to estimate the model parameters using both labelled and unlabelled data.

Semi-supervised learning can be applied at various use cases pertaining to unsupervised learning or supervised learning:

- **Sentiment Analysis:** As described in the example earlier, sentiment analysis in natural language processing can benefit from semi-supervised learning. By using a combination of labelled customer reviews and unlabelled text data, models can learn to identify sentiments (positive, negative, neutral) in a more efficient and accurate manner.
- **Image Classification:** In image classification tasks, obtaining large labelled datasets can be expensive and time-consuming. Semi-supervised learning allows models to utilize unlabelled images in combination with a smaller set of labelled images to improve classification accuracy and generalization.
- **Speech Recognition:** Training speech recognition systems with fully labelled speech data is challenging due to the need for extensive transcription efforts. Semi-supervised learning techniques can use a mix of labelled and unlabelled speech data to enhance the performance of speech recognition models.
- **Anomaly Detection:** In scenarios where normal data is abundant, but labelled anomalies are scarce, semi-supervised learning can be employed for anomaly detection. The model learns from the normal data while leveraging the information present in the unlabelled data to detect abnormal patterns.
- **Medical Diagnosis:** Semi-supervised learning is beneficial in medical diagnosis, where obtaining large labelled medical datasets can be difficult. By utilizing both labelled medical records and unlabelled data, models can improve diagnostic accuracy and assist healthcare professionals in making informed decisions.
- **Fraud Detection:** Detecting fraudulent activities in financial transactions can be challenging due to the scarcity of labelled fraud instances. Semi-supervised learning can help identify patterns of fraud by learning from both labelled fraud examples and vast amounts of unlabelled transaction data.
- **Recommendation Systems:** Recommendation systems can benefit from semi-supervised learning to better understand user preferences and generate personalized recommendations. Unlabelled user behavior data, combined with a smaller set of explicitly labelled user preferences, can lead to more accurate recommendations.
- **Machine Translation:** Semi-supervised learning can enhance machine translation systems by leveraging large amounts of unlabelled text data from different languages. Combining this data with smaller labelled translation datasets can lead to improved translation quality.
- **Social Network Analysis:** Analysing social networks, identifying communities, and detecting influential nodes can be challenging without sufficient labelled data. Semi-supervised learning can help uncover hidden patterns and structures in social networks using both labelled and unlabelled data.

12.3.1.4. Self-Supervised Learning

Good performance of Machine Learning models usually requires a decent number of labels, but collecting manual labels is expensive and even more expensive to be scaled up. Self-supervised learning is a specialized form of learning, where the model generates its own pseudo-labels from the input data unlike supervised learning, where explicit labels are provided. It accomplishes this by creating auxiliary tasks or objectives based on the data itself. Idea is to get labels for unlabelled data and train unsupervised dataset in a supervised manner.

In self-supervised learning, the model leverages the inherent structure or relationships within the unlabelled data to generate "pseudo-labels" for itself. These pseudo-labels serve as proxy labels for training the model. The model is presented with transformed versions of the original data. These transformations might involve randomly cropping, rotating, flipping, or altering the data in various ways. The objective here is to train the model to reconstruct the original, unaltered data from the transformed version. Essentially, the model learns to predict the original data by comparing it with the transformed data. The reconstruction tasks in self-supervised learning are often referred to as **pretext tasks** since they serve as intermediate objectives/representations to guide the learning process.

These learned representations, also known as **latent variables**, are expected to capture important semantic or structural information about the data, making them valuable for various downstream tasks.

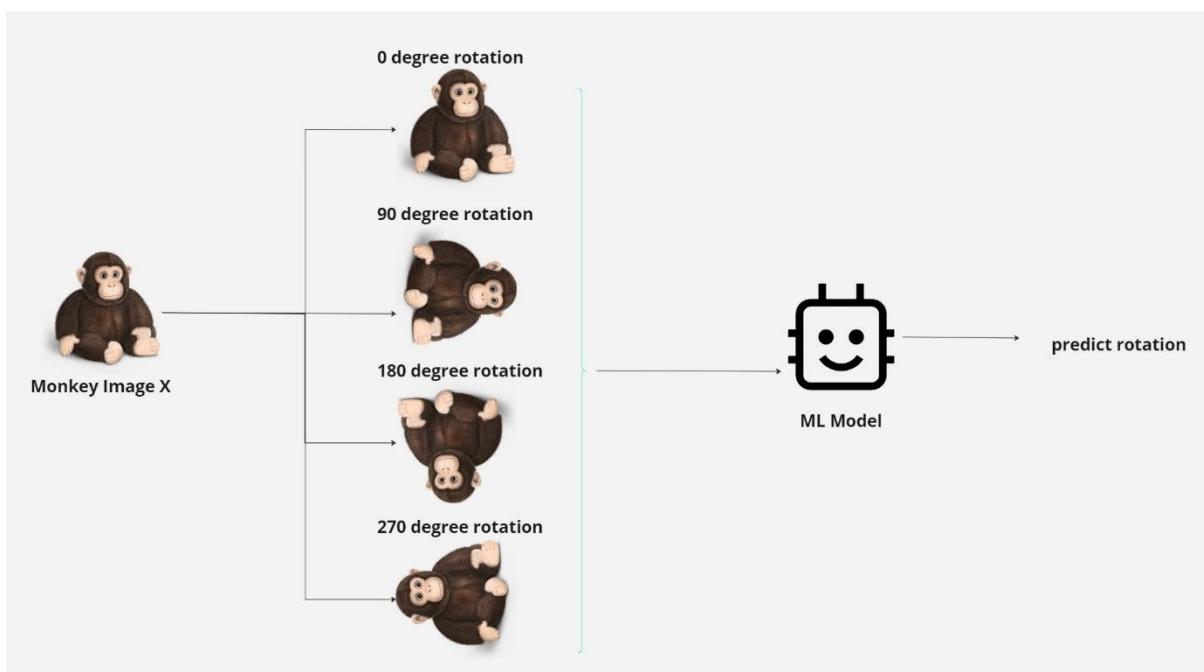


Figure 11-0-5 Self-Supervised Learning

Using the example of image rotation, where the model is trained to predict the rotation angle of each input image, the rotation prediction task is indeed an invented task with unimportant accuracy. The pretext task of predicting image rotations encourages the model to learn high-level object parts and their relative positions, rather than focusing on local patterns or low-level features. By requiring the model to understand the spatial relationships between different parts of an object, self-supervised learning drives the learning process towards acquiring meaningful and semantically rich representations of the data.

When the model predicts the rotation angle of an image, it needs to capture the global structure and layout of the object to identify the correct orientation. This means that the model has to recognize high-level object parts, such as heads, noses, and eyes, and understand how they are spatially arranged in the image.

For example, in the context of recognizing a monkey face, the model must learn to identify the key facial features, such as eyes, nose, and mouth, and their positions in relation to each other. It needs to recognize that eyes are typically above the nose and mouth and that the nose is below the eyes. By understanding these spatial relationships, the model can correctly predict the rotation angle of the face.

In contrast, if the model were to focus only on local patterns or low-level features, it might struggle to recognize the overall structure of the object and its parts. It could easily mistake a rotated face for a completely different object because it lacks the understanding of the global object layout.

By training the model on the pretext task of predicting rotations, it learns to extract meaningful and high-level features that are crucial for understanding the objects' semantic concepts. These learned representations are not only useful for the rotation prediction task itself but also carry valuable information about the objects' underlying structure, making them transferable and beneficial for other downstream tasks, such as:

- **Object Recognition:** The learned representation can be transferred to construct an object recognition classifier with only a few labelled samples. The model's understanding of the object's orientation might generalize well to recognize objects in different orientations, even with limited labelled training data.
- **Image Retrieval:** The learned features can be utilized to measure similarity between images, enabling effective image retrieval where similar images can be retrieved based on their latent representation.
- **Fine-Grained Classification:** The model might implicitly learn features specific to object details, such as texture or shape, which can be useful for fine-grained classification tasks where distinguishing similar object categories is challenging.
- **Image Generation:** The learned representation can be employed in generative models to synthesize realistic images with specific rotations or orientations.

Let us Consider the task of learning representations for text data using self-supervised learning. Given a large corpus of unlabelled text, self-supervised learning can be employed to generate pseudo-labels for the text. One common pretext task for text data is "masked language modelling."

In this pretext task, random words in a sentence are masked, and the model is trained to predict the masked words based on the surrounding context. For example, given the sentence "The cat chased the __," the model might be trained to predict the masked word "mouse" based on the context "The cat chased the". By solving the masked language modelling task on a large amount of text data, the model learns to represent words and sentences in a way that captures semantic relationships and contextual information.

After generating pseudo-labels through masked language modelling, the model treats these masked words as the ground truth and uses them to train a supervised learning algorithm for a specific downstream task, such as sentiment analysis or text classification.

12.3.1.4.1. Popular Self-supervised Learning Algorithms

- **Autoencoders:** Autoencoders are neural network architectures that aim to encode input data into a compact representation (encoding) and then decode it back to the original data. They are trained to

minimize the reconstruction error between the input and the output, effectively learning useful features without explicit labels.

- **Contrastive Learning:** Contrastive learning involves training a model to distinguish between positive pairs (similar samples) and negative pairs (dissimilar samples) in the dataset. It learns to pull similar samples closer in the feature space while pushing dissimilar samples apart.
- **Predictive Coding:** Predictive coding is a framework where a model predicts future data points based on past observations. It learns by minimizing the prediction error, leading to the discovery of meaningful features that capture temporal relationships.
- **Temporal Order Prediction:** In this approach, a model is trained to predict the temporal order of data points within a sequence. By learning the temporal structure, the model gains insights into the underlying patterns.
- **Contextual Word Embeddings:** Contextual word embeddings, such as those produced by models like BERT (Bidirectional Encoder Representations from Transformers), learn to predict missing words in a sentence based on the surrounding context. These embeddings capture semantic relationships between words.
- **Jigsaw Puzzle Solving:** In this technique, images are divided into pieces, shuffled, and the model is trained to predict the correct arrangement. It learns to capture spatial relationships and visual patterns.
- **Image Inpainting:** Models learn to fill in missing parts of an image. By predicting the missing content, the model learns to understand the context and structure of images.
- **Video Frame Prediction:** Similar to image inpainting, models predict future frames in a video sequence. This encourages the model to capture motion and temporal dependencies.
- **Colorization:** Models predict color information for grayscale images. By understanding context and relationships between objects and their colours, the model learns useful features.
- **Rotation Prediction:** A model learns to predict the rotation angle applied to an image. This helps the model capture geometric transformations and orientations.

Self-supervised learning techniques are valuable for pretraining models on large amounts of unlabelled data. The learned representations can then be fine-tuned for downstream tasks with limited labelled data, often resulting in improved performance compared to starting from scratch. Each algorithm employs specific data transformations and reconstruction tasks based on the nature of the data. Let us discuss some of its real-life applications.

In Natural Language Processing, Pretext tasks like masked language modelling (as discussed earlier) or predicting word order in a sentence help create rich language representations, which can be utilized for tasks like text classification, sentiment analysis, and machine translation.

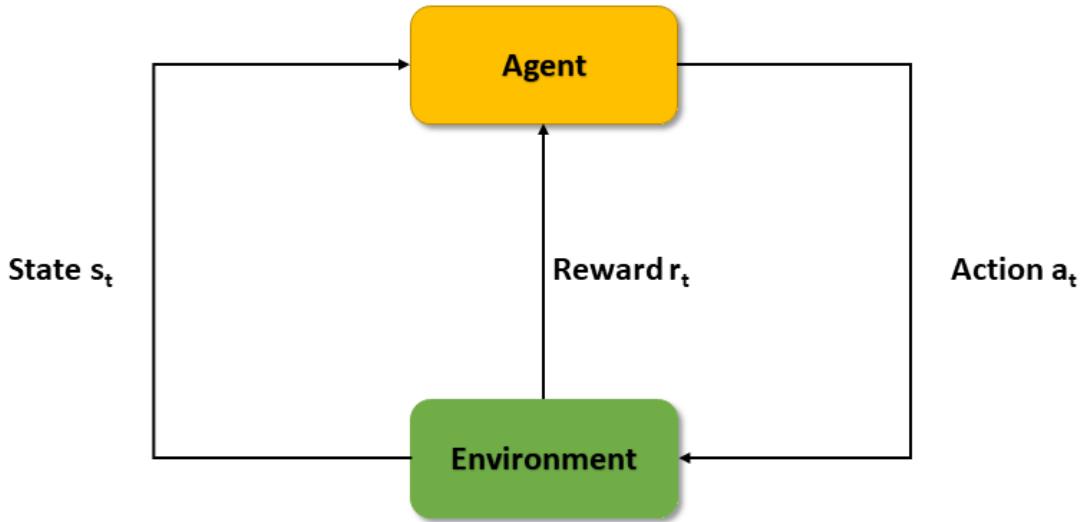
Tasks like predicting the next audio frame or recognizing speaker identity can aid in creating representations for various speech and audio processing applications, including speech recognition and audio scene analysis.

12.3.1.5. Reinforcement Learning

Reinforcement learning is a distinct paradigm where an agent (learning system) learns to make decisions by interacting with an environment. The agent receives feedback in the form of rewards or penalties based on its actions, guiding it towards maximizing cumulative rewards over time.

Say, a child is presented with two glasses of water in a stainless-steel container, one very hot and other at normal temperature. But the child is unaware of the consequence of hot or cold water. Child holds the hot

one and removes hand instantly but feels comfortable with cold one. Child learns to avoid hot objects in future.



Reinforcement Learning- Learning through feedback

Figure 12-0-6 Reinforcement Learning

In the beginning of this process, the agent explores the environment by trying different actions to understand their consequences and rewards. As the agent gains more experience, it starts to exploit its knowledge by selecting actions that have previously resulted in higher rewards. It updates its policy based on the rewards received after each action and aims to find the best policy that maximizes the expected cumulative reward. Training a machine to play a game of chess can be a good example to study the analogy of Reinforcement Learning (RL).

To train a machine to play chess using supervised learning, we would need a dataset consisting of expert moves and their corresponding labels (i.e., the correct next move). However, acquiring such a dataset would be challenging and resource-intensive, as it requires human expertise to annotate the data. Additionally, supervised learning would only enable the machine to mimic human behavior, limiting its potential to surpass human-level performance.

RL, on the other hand, takes a different approach. Instead of relying on a fixed dataset, RL trains the machine by allowing it to interact with the chessboard environment and learn from the outcomes of its actions. The state at any given moment represents the current configuration of pieces on the board. The machine plays chess against itself, making a sequence of moves, and receives rewards or penalties based on the outcomes of those moves. The rewards indicate positive outcomes (e.g., winning the game), while penalties indicate negative outcomes (e.g., losing the game).

Through this trial-and-error process, the machine learns to optimize its decision-making strategy to maximize the total reward over time. It explores different moves, evaluates their outcomes, and adjusts its strategy based on the feedback received from the environment. Over many iterations, the machine gradually improves its chess-playing abilities, becoming more adept at making strategic moves.

RL enables the machine to engage in self-play, where it plays against its past versions or other instances of itself. This self-play encourages the machine to learn from its own experiences and continuously improve its performance without relying on external data or human expertise.

12.3.1.5.1. Popular Reinforcement Learning Algorithms

- Q-Learning: Q-learning is a model-free RL algorithm that learns a value function (Q-values) to estimate the expected cumulative reward for taking a particular action in a specific state. It uses the Bellman equation to update Q-values based on the reward received and the estimated future rewards.
- Deep Q-Network (DQN): DQN is an extension of Q-learning that uses deep neural networks to approximate Q-values. It's particularly effective for high-dimensional state spaces, such as images, and has been applied to playing video games.
- Policy Gradient Methods: Policy gradient methods directly optimize the policy (strategy) of the agent to maximize the expected cumulative reward. They use gradient descent to update policy parameters based on the reward signal.
- Actor-Critic: Actor-Critic is a hybrid approach that combines elements of policy gradient and value-based methods. It uses an actor network to determine the agent's actions and a critic network to estimate the value function.
- Proximal Policy Optimization (PPO): PPO is a policy optimization method that aims to improve policy performance while ensuring that policy updates do not deviate too far from the previous policy. It balances exploration and exploitation.

Reinforcement learning is well-suited for tasks that involve decision-making in dynamic and complex environments, such as game playing, robotics, and autonomous systems. It has been highly successful in game playing, where agents can learn to play video games. Robots can learn to perform tasks like grasping objects, navigating through environments, and completing assembly tasks with Reinforcement Learning. It can be leveraged to control energy consumption in smart grids, optimizing power usage and distribution.

Reinforcement Learning implementation through human feedback is extensively discussed in the last chapter of this book with respect to Large Language Models.

12.3.2. Knowledge Representation

The second dimension of classification delves into how the learners represent and store the knowledge or skills acquired during the learning process. This dimension is diverse, encompassing various approaches such as symbolic representations, statistical models, decision trees, neural networks, and more. Symbolic representation involves using explicit rules and logical statements, making it interpretable but often rigid. Statistical models leverage probabilities and statistical techniques to model relationships between variables, offering flexibility and adaptability. **Neural networks**, inspired by the human brain, create interconnected layers of artificial neurons, enabling deep learning and complex pattern recognition.

12.3.3. Application Domain

The third dimension of classification revolves around the application domain of the performance system for which knowledge is acquired. Machine learning systems are applied across diverse domains, ranging from image and speech recognition to **Natural Language Processing**, finance, healthcare, and beyond.

The application domain influences the selection of learning algorithms and knowledge representation methods, as each domain poses unique challenges and requirements.

Note: We shall be studying about different ML algorithms, Natural Language Processing (NLP) techniques and Neural networks in detail in upcoming chapters.

12.4. Challenges with Machine Learning

By far we have got an overview on various Learning strategies. However, implemented these strategies is easier said than done. In Machine learning, the primary task is to select a suitable model and suitable data to move forward, however, these are the two very places where things can go haywire. Challenges faced in machine learning can be categorized into two groups: based on data and based on algorithms.

Let us first discuss the challenges related to data or to say bad data.

12.4.1. Challenges Based on Data

- Insufficient Quantity of Training Data: One of the fundamental requirements for machine learning algorithms to perform effectively is having enough labelled training data. For complex problems like image or speech recognition, millions of examples may be needed to train a model effectively. Obtaining large and diverse datasets can be costly and time-consuming.
- Nonrepresentative Training Data: The success of a machine learning model depends on how well it generalizes to new, unseen data. If the training data is not representative of the real-world scenarios, the model may fail to make accurate predictions on new instances. Careful consideration of data collection and ensuring the diversity and inclusiveness of the training data is crucial.
- Poor-Quality Data: Low-quality training data, such as data with errors, outliers, or missing values, can lead to suboptimal model performance. Data cleaning and pre-processing are essential steps to remove noise and ensure the reliability of the training data.
- Irrelevant Features: Having too many irrelevant features in the training data can hinder the learning process and lead to overfitting. Feature engineering is a critical aspect of machine learning, involving selecting the most relevant features and creating new ones to improve model performance.

12.4.2. Challenges Based on Algorithms

- Overfitting the Training Data: Overfitting occurs when a model performs well on the training data but fails to generalize to new data. It happens when the model is too complex relative to the available training data. Techniques like model simplification, gathering more data, and reducing noise can help combat overfitting.
- Underfitting the Training Data: Underfitting is the opposite of overfitting, where the model is too simple to capture the underlying patterns in the data. This results in poor performance both on the training data and new data. Solutions to underfitting include using more powerful models, better feature engineering, and relaxing model constraints.
- Model Selection and Hyperparameter Tuning: Selecting the right model and tuning its hyperparameters are critical tasks in machine learning. The choice of model architecture and hyperparameter values can significantly impact the model's performance. Proper validation techniques and cross-validation are essential to find the optimal model settings.
- Computational Complexity: Some machine learning algorithms, especially deep learning models, are computationally intensive and require significant computational resources. Training large models on vast datasets can be time-consuming and resource-intensive.

12.5. Summary

So far, we have come to understand Machine Learning's fundamental essence: **the art of enabling machines to improve at specific tasks through learning from data**. Unlike traditional programming, where explicit rules are coded, machine learning empowers systems to learn patterns and relationships from vast datasets, adapting and optimizing their performance over time.

Throughout our discussion, we have encountered various types of machine learning systems, each designed to address different learning scenarios. Supervised learning, the most common type, involves learning from labelled data, where the algorithm is guided by provided examples and corresponding outcomes. Unsupervised learning, on the other hand, delves into the realm of discovering patterns from unlabelled data, allowing the system to identify hidden structures within the data without explicit guidance. Reinforcement learning introduces a different perspective, where an agent interacts with an environment, receiving rewards or penalties for its actions, and learns the best strategy to maximize cumulative rewards.

However, we also discussed several challenges that lie ahead in the world of machine learning. The first set of challenges pertains to data, the foundation upon which machine learning systems are built. Insufficient quantity of training data can hinder model performance, and nonrepresentative training data may lead to poor generalization. Additionally, poor-quality data, riddled with errors and noise, can impede the detection of underlying patterns, necessitating data cleaning, and pre-processing efforts. Selecting relevant features and creating new ones through feature engineering are vital to ensuring the model's capability to learn and generalize.

The second set of challenges lies in the algorithms themselves. Overfitting and underfitting are twin challenges, each representing a state where the model either becomes overly complex and specialized to the training data or too simplistic to grasp the underlying relationships. Balancing the model's complexity and generalization is crucial for optimal performance. The selection of appropriate models and tuning hyperparameters are essential tasks, impacting the overall effectiveness of the learning system. Furthermore, computational complexity can become a significant constraint, especially with deep learning models, requiring substantial computational resources for training and inference.

In coming chapters, we shall be dealing with various methods and algorithms which help us implement Machine Learning strategies and overcome the challenges.

Objects, Data & AI: Build thought process to develop Enterprise Applications

Feature Engineering

“

“'Applied machine learning' is basically feature engineering” - Prof. Andrew Ng

Picture a puzzle game - a complex mosaic of information (in form of raw data) scattered across the canvas. Now imagine that we are trying to teach a computer (machine) to solve this puzzle, to find patterns and make predictions. But here's the catch: the computer doesn't understand the puzzle pieces the way we do. It needs a translator, a way to turn these puzzle pieces into something it can work with – and that's where feature engineering comes in.

Feature engineering is like transforming these puzzle pieces into a language the computer can speak. It's the art of picking and choosing the right aspects of the data and turning them into numbers (*because computers understand numbers only! Well, honestly not numbers but binary numbers, but numbers can be easily transformed to binary representations*), which the computer can crunch. These transformed numbers are called "**features**". Think of features as the building blocks that the computer uses to understand and make sense of the data.

We do feature engineering in real life in our subconscious. A property dealer does not have to consult a computer to give an estimated price of the property, he/she can predict the price via experience and exposure to various nuances of this field over the years, that leads to development of an intuition for the correct price. While estimating or predicting the price of a house, the common features for consideration are land area or size of the house or plot, number of rooms, locality, floors in an apartment, amenities around, etc. These details act as clues via which a property dealer makes sense of what the probable price of the property should be. Likewise, if we want to model something similar for computers, so that this price estimation functionality is also available for the customers, we would have to feed the model these clues and model would need to figure out what are important clues or parameters and in accordance what should be the correct estimated price of the property.

Feature engineering is like handpicking the most important clues and presenting them to the computer in a way it can grasp. If you choose the right clues, the computer becomes a prediction magician, but if you choose the wrong ones, it's like giving the magician the wrong ingredients – the magic just won't happen. It's the secret sauce that can make or break a machine learning project.

Feature Engineering is something like writing or compiling a book. We carefully select the most interesting and relevant parts to captivate your audience. In the same way, feature engineering helps us choose the right parts of the data to captivate the computer, making it a smart predictor. However, finding the perfect balance is crucial. Say, while writing this book, I would keep on extending the scope of the subject, the book will ultimately lose its purpose and reader would get overwhelmed and bored, as the entire computer science or even a smaller field like study of Objects, Databases or Machine Learning, cannot be compiled in a single book. Neither is my purpose of presenting you with an encyclopaedia. I have a defined purpose of presenting you with various aspects of Software paradigms and concepts which help in build of a successful enterprise software product. And to do so, I carefully chose to include certain topics, exclude certain topics, elaborated on some topics and just presented an overview on some. Another writer might

have presented the content for the same purpose in a very different way and that is the beauty of knowledge. We try to see things via different lens and formulate a perspective.

Feature engineering works the same way. If we use too many unnecessary details, the computer might get overwhelmed and confused, leading to bad predictions. On the flip side, if we leave out important details, the computer won't have enough information to make accurate predictions. It is like crafting a recipe for success in machine learning. It's about selecting the right ingredients (**features**) that will make our dish (**model**) not only accurate but also insightful. It's the bridge between the real world and the computer's world, ensuring that the computer can understand and unravel the mysteries hidden within the data.

13.1. Scalers and Vectors

Since, computers can't digest texts or images or other formats of data, it becomes essential to represent data in numerical format in Machine Learning (ML). ML models need input to be transformed or encoded into numbers. Scalers, Vectors, and matrices represent inputs like text and images as numbers, so that we can train and deploy our models.

Linear Algebra is the mathematical foundation that solves the problem of representing data as well as computations in machine learning models.

Note: Elaborate discussion on Linear Algebra and explicit details of Feature Engineering is out of scope for our discussion. However, we would be overlooking at the core concepts to build an intuition for the subject.

Let us look a small sample dataset to better our understanding.

	Height (cm)	Weight (kg)	Age
Person 1	170	80	30
Person 2	160	75	20
Person 3	158	78	25

Each dataset resembles a table-like structure consisting of rows and columns, where each row represents observations, and each column represents features/Variables. And a table like structure is represented using Matrices in Linear Algebra. In computer language, it is represented using arrays.

13.1.1. Scalers

Scalars are single numerical values that represent a single feature or data point. They are simple and easy to understand, as they only represent one piece of information. In above dataset, each individual height and weight is a scalar. For example, the height of Person 1 (170 cm) is a scalar value, and the weight of Person 2 (75 kg) is also a scalar value. Each scalar represents a single attribute of a person.

13.1.2. Vectors

Vectors are 1-dimensional arrays that can represent multiple features together. Each element in the vector corresponds to a specific feature or attribute. For our example dataset, we can create two vectors to represent the dataset. One vector will represent the heights, and the other vector will represent the weights. Each vector will have four elements, corresponding to the heights and weights of the four individuals.

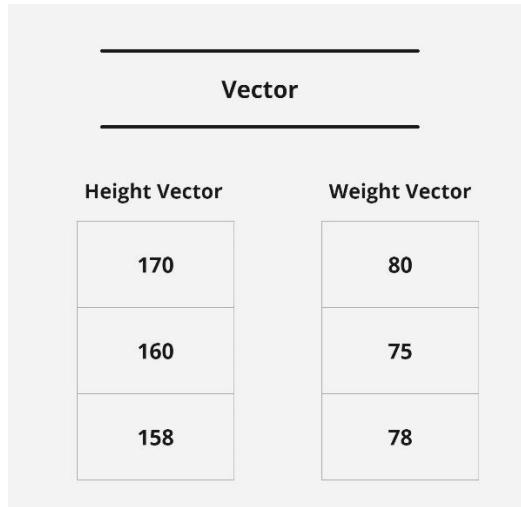


Figure 12-0-1 Vector Representation of Features

The creation of mathematical representations involving scalars and vectors is central to various statistical techniques and machine learning algorithms. These techniques often involve transforming raw data through mathematical operations, such as normalization, aggregation, or combination. The resulting representations, enriched with scalar and vector components, help machine learning models to understand complex patterns and relationships within the data.

13.2. Data Visualization and Analysis

Data visualization plays a crucial role in feature engineering, as it helps data scientists and machine learning practitioners understand the characteristics of the data and make informed decisions about which features to use in their models. It can reveal relationships and correlations between different features. Visualization techniques make use of vectors and matrices to display the data in various graphical formats, helping us gain insights and understanding of the underlying patterns.

Let us consider a sample dataset for house price prediction model.

Bedroom	Area (sq. ft)	Age	House Price (in \$)
3	2000	5	300,000
4	2500	8	400,000
2	1500	2	250,000
5	3000	3	500,000

We can represent this dataset as a matrix. Each row in the matrix represents a single house, and each column represents a specific feature. In this dataset, we have three data points (houses) and four features (Bedrooms, Square Footage, Age, and House Price). The first three columns represent the **input features**, and the last column represents the **target** variable (House Price) that we want to predict.

Say, we want to analyse the relation of house area with house price. We can do so by plotting the features on a 2-dimensional plane to have a better visualization.

```
import matplotlib.pyplot as plt

# Data
```

```
square_footage = [2000, 2500, 1500, 3000]
house_price = [300000, 400000, 250000, 500000]

# Create the graph
plt.scatter(square_footage, house_price)
plt.xlabel('Square Footage')
plt.ylabel('House Price')
plt.title('Housing Data')
plt.grid(True)
plt.show()
```

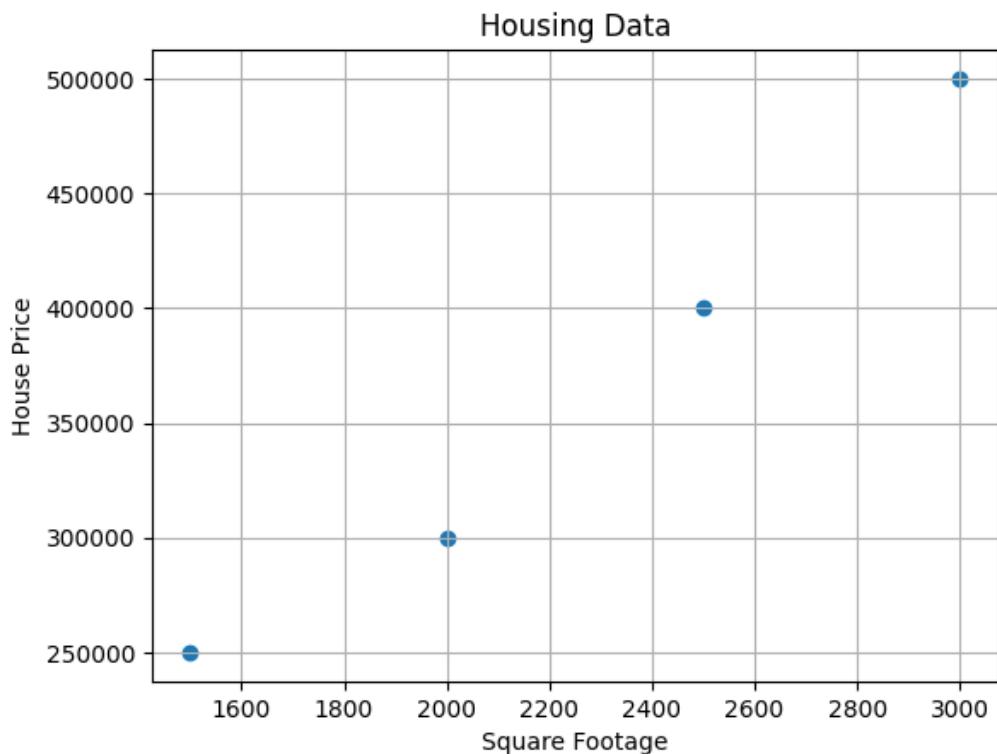


Figure 12-0-2 Scatter Plot

After plotting the selected features on a 2-D plane, we can easily recognize the linear relationship between house area in square footage and house price. As house area increases, the price of house increases.

The above process is often called as **Exploratory Data Analysis (EDA)**. It involves systematically examining and summarizing data to gain insights, discover patterns, detect anomalies, and formulate hypotheses. EDA helps researchers and analysts understand the structure of the data, uncover potential relationships between variables, and make informed decisions about subsequent analysis or modelling.

13.2.1.1. Benefits of EDA

Visualizations are a cornerstone of EDA. By creating graphs, charts, and plots, analysts can visually explore data patterns and relationships. Common types of visualizations include histograms, box plots, scatter plots, line charts, and bar graphs. Visualizations help identify trends, clusters, gaps, and potential outliers. EDA also involves examining the distribution of data.

Scatter plots, pair plots, and correlation matrices help us identify which features are strongly correlated with the target variable or with each other. Correlated features may need to be removed or combined to avoid multicollinearity issues. Exploring data visually often sparks creative ideas for new features. For example, combining or binning features based on certain patterns or domain knowledge might lead to more informative representations.

Data visualization techniques like histograms, box plots, and density plots allow us to visualize the distribution of individual features. Understanding the distribution helps in identifying outliers, skewed data, and potential data transformation requirements. Data visualization can also help to identify missing data patterns. Understanding which features have missing values and how they relate to other features can inform imputation strategies or lead to the creation of new features based on missing data indicators.

13.3. Dealing with Numerical Data

As we ultimately convert raw data into numerical form to feed our models, numerical feature engineering becomes a foundational tool in the field of data science and machine learning. It can be applied whenever raw data is converted into numeric features, and its application is not limited to a specific domain or problem type. Whether it's structured data, unstructured data, time series data, or any other data format, numeric feature engineering plays a crucial role in extracting meaningful insights and improving the performance of machine learning models.

In the age of Big Data, where vast amounts of data are generated and collected at an unprecedented rate, the quality and relevance of features play a crucial role in building accurate and efficient machine learning models. Numerical feature engineering involves transforming raw data into meaningful and informative numerical representations, which can significantly impact the performance and success of machine learning algorithms.

Suppose we have a dataset containing the number of purchases made by different customers in an online store.

Customer Id	Purchase Count
C1	5
C2	3
C3	10
C4	2
C5	8
C6	4
C7	6
C8	20
C9	7
C10	15

Note: For visualization's sake, we have considered a small dataset, however, for an online store like Amazon, E-bay, etc. data entries would be in millions.

Our goal is to extract information about buying behaviors of respective customers, so that customers with frequent buying tendency can be recognized and rewarded with some add-ons.



Let us start with performing EDA before doing actual feature engineering on our dataset.

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Sample data
# Note: Dataset can be read via Excel, CSV files as well.
data = {
    'CustomerID': ['C1','C2','C3','C4','C5','C6','C7','C8','C9','C10', ],
    'PurchaseCount': [5, 3, 10, 2, 8, 4, 6, 20, 7, 15 ]
}

df = pd.DataFrame(data)
```

Let us plot the data frame on a histogram to understand the distribution of data by dividing the data into bins and displaying frequency or count of data points falling into each bin.

```
plt.figure(figsize=(8, 5))
sns.histplot(df['PurchaseCount'], bins=10, kde=True)
plt.xlabel('Purchase Count')
plt.ylabel('Frequency')
plt.title('Histogram of Purchase Count')
plt.show()
```

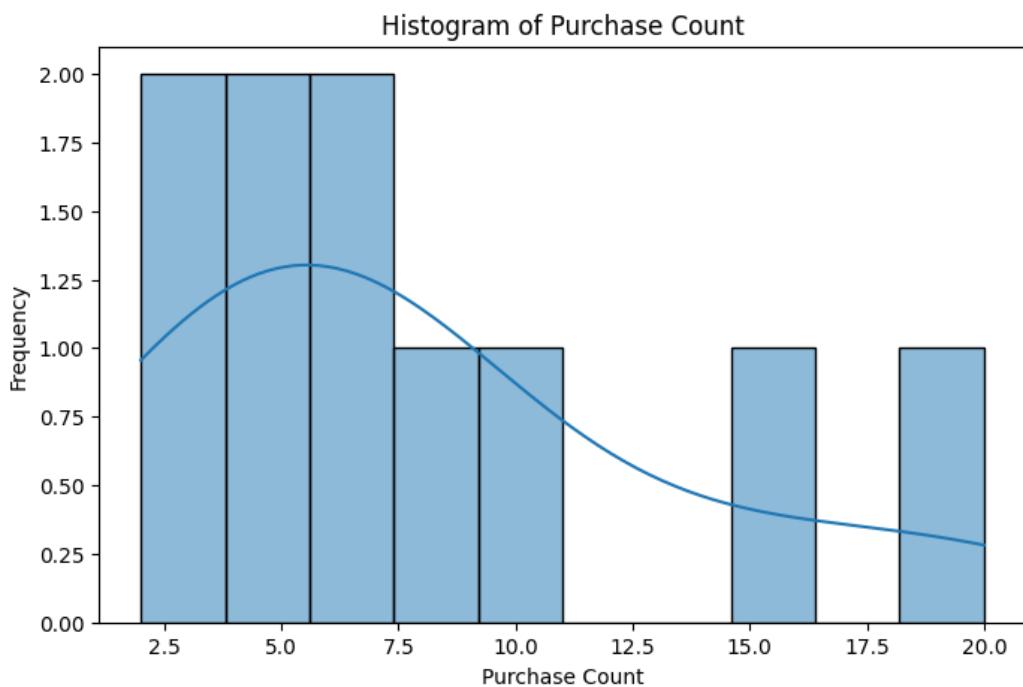


Figure 12-0-3 Histogram representation of Purchase Count

If we look at the histogram in Fig 12-3, we see that there are many customers with purchase counts between 1 and 7, and only a few customers with purchase counts above 7, this would be an indication of positive skewness.

In the context of the purchase count data, a positively skewed distribution suggests that there are relatively more customers with lower purchase counts, while fewer customers have higher purchase counts. The majority of customers are making fewer purchases, while a smaller number of customers are making more purchases, causing the histogram to stretch towards the right.

Understanding the skewness of data is essential because it can impact the performance and assumptions of certain statistical and machine learning models. In positively skewed data, the mean tends to be larger than the median, as the long right tail pulls the mean towards higher values. The median is less affected by extreme values, making it a better measure of central tendency in skewed data.

```
# Binarization
threshold = 7
df['FrequentBuyer'] = df['PurchaseCount'].apply(lambda x: 1 if x > threshold else 0)

print(df)
```

CustomerID	PurchaseCount	FrequentBuyer
0	C1	0
1	C2	0
2	C3	1
3	C4	0
4	C5	1
5	C6	0
6	C7	0
7	C8	1
8	C9	0
9	C10	1

13.3.1. Binarization

Binarization is the process of converting numerical data into binary values based on a threshold. In this case, we might want to create a binary feature to represent whether a customer is a frequent buyer or not. Let's say we set the threshold at 7 purchases. Customers who made 7 or more purchases will be labelled as "FrequentBuyer," and those who made less than 7 purchases will be labelled as "NonFrequentBuyer."

So far, we have only numerical features. We can use purchase count of customers to create a categorical feature like "Purchase Activity", which would give more insight into buying behavior of customers.

```
# Binning
bins = [0, 3, 6, float('inf')] # Bins: [0-3], (3-6], (6-inf)
labels = ['Low', 'Medium', 'High']
df['PurchaseActivity'] = pd.cut(df['PurchaseCount'], bins=bins, labels=labels)

print(df)
```

CustomerID	PurchaseCount	FrequentBuyer	PurchaseActivity
0	C1	5	Medium
1	C2	3	Low
2	C3	10	High
3	C4	2	Low
4	C5	8	High
5	C6	4	Medium
6	C7	6	Medium
7	C8	20	High
8	C9	7	High
9	C10	15	High

13.3.2. Binning

Binning is the process of dividing a continuous numerical feature into discrete bins or intervals. We can use binning to group customers based on their purchase count into categories like "Low," "Medium," and "High" purchase activity.

Binning helps in capturing patterns related to purchase behavior and allows us to interpret customer purchase activity based on these categories (Low, Medium, or High). It also reduces the impact of small variations in the purchase count and can lead to a more robust and interpretable model, especially in cases where there are nonlinear relationships between the feature and the target variable.

In the context of above example, we discussed that the purchase count distribution is right skewed as there are very few customers with high purchase count. This can lead to misinterpretation of dataset. One possible scenario can be an overestimation of average spending. The **mean (average)** purchase count would be pulled towards the higher values, potentially suggesting higher spending than what is typical for most customers. If we are analysing customer behavior based on "PurchaseCount," a right-skewed distribution could misrepresent the typical purchasing behavior. The average might indicate more frequent purchasing than is actually common among most customers.

Hence to mitigate skewed data from ruining our data analysis, we use a powerful tool called **Log transform**. Mathematically, the log function compresses the range of large numbers and expands the range of small numbers. The larger x is, the slower log(x) increments. Let us plot a logarithmic curve to understand this.

```
import numpy as np
import matplotlib.pyplot as plt

# Generate data
x = np.linspace(1, 10, 100) # Generate values from 1 to 10
y = np.log(x) # Apply the natural logarithm to the data

# Create the plot
plt.figure(figsize=(8, 6))
plt.plot(x, y, label='Logarithmic Curve', color='blue')
plt.xlabel('X')
plt.ylabel('Log(X)')
plt.title('Logarithmic Curve')
plt.legend()
plt.grid(True)
```

```
plt.show()
```

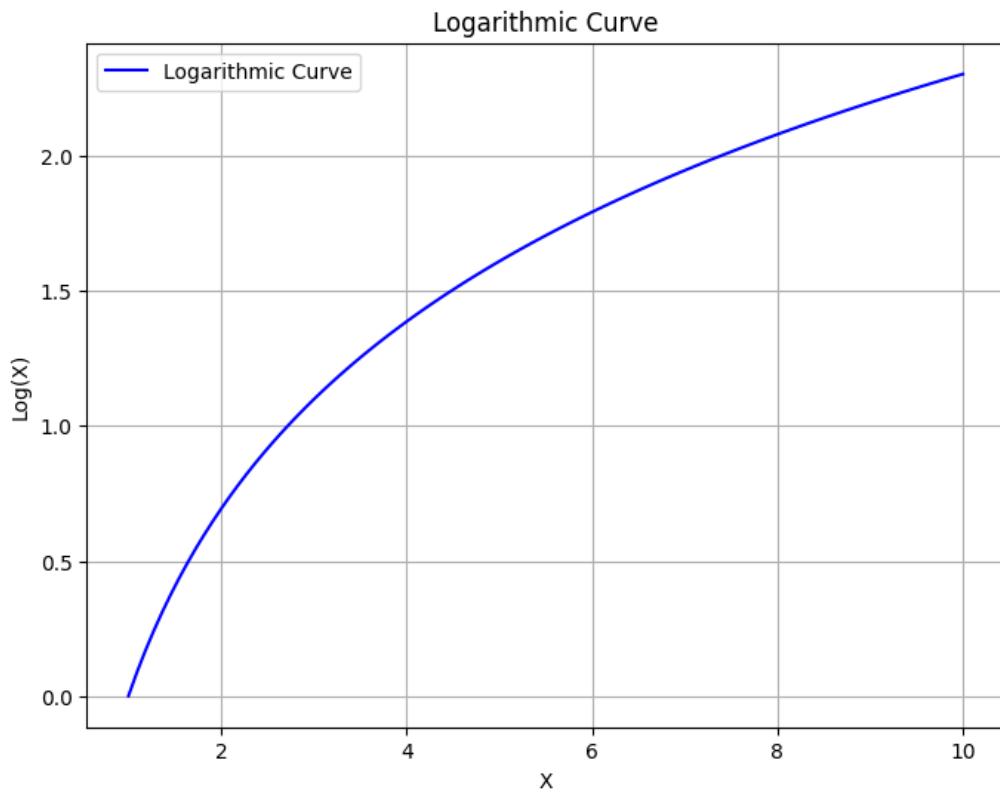


Figure 12-0-4 Logarithmic Curve

As we can see, the x-values increase linearly while the y-values increase logarithmically. The curve starts steep and gradually flattens out as x increases. We use this concept to transform our data set to address heavy tailed distribution.

```
import numpy as np

# Log transformation
df['Log_Purchase'] = np.log(df['PurchaseCount'])

# EDA - Histogram after log transformation
plt.figure(figsize=(8, 6))
sns.histplot(df['Log_Purchase'], bins=10, edgecolor='black', kde=True)
plt.xlabel('Log_Purchase')
plt.ylabel('Frequency')
plt.title('Log Transformed Purchase Distribution')
plt.show()
```

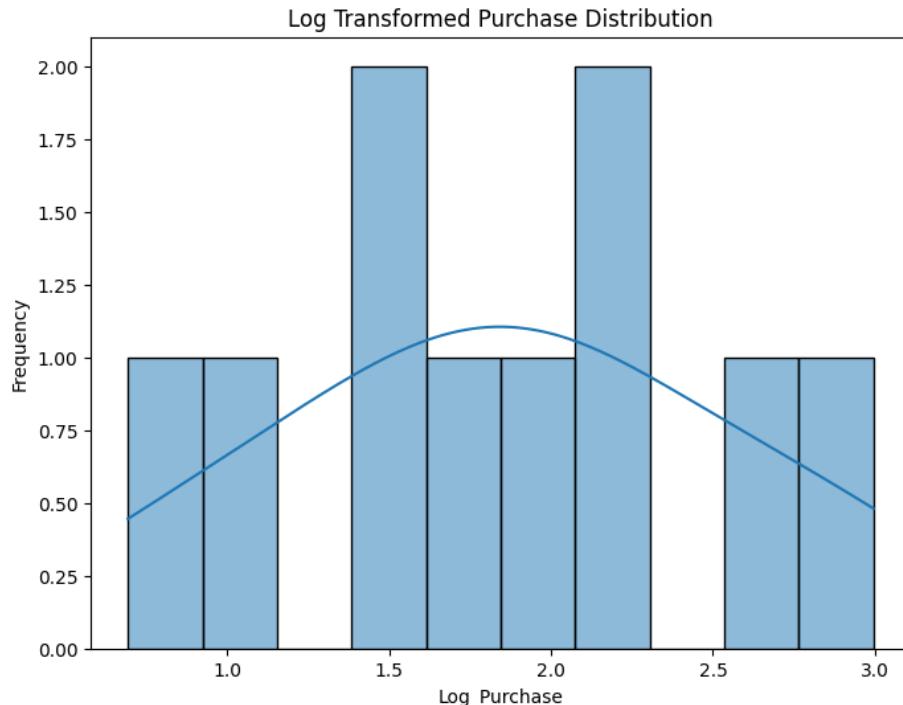


Figure 12-0-5 Log transformed data set presents a normalized curve

Compare the histogram curve of Fig 12-5 with curve of [Fig 12-3](#), you can clearly see a bell shaped curve with normalized distribution after performing Log transformation on dataset.

However, the decision to apply a log transform depends on the distribution of the data and the requirements of the analysis or modelling task. It's essential to assess the data distribution and consider the implications of the transformation on the specific use case before applying any data transformations.

13.3.3. Feature Scaling

A dataset contains features at different scales, for example latitude, longitude, percentage value, temperature measurements, etc. have bounded values within a specific range, while other features, like counts, number of occurrences, etc. can increase without any upper limit.

Models that involve smooth functions of input features, such as linear regression and logistic regression, are sensitive to the scale of the input. The scale of features can impact the coefficients or weights assigned to each feature in the model. Any model which uses matrix operations will get impacted if feature scaling is not performed before using the model since matrix operations are sensitive to the magnitude of values.

Let us revisit the [housing dataset](#) we used for EDA example earlier:

Bedrooms	Area (sq. ft)	Age	House Price (in \$)
3	2000	5	300,000
4	2500	8	400,000
2	1500	2	250,000
5	3000	3	500,000

The "(Number of)Bedrooms" feature values are relatively small integers (e.g., 2, 3, 4), while the "Area in Square Footage" feature values are larger numbers. Without scaling, algorithms that are sensitive to scale

might give more weight to "Square Footage" due to its larger numerical range. This could lead to underestimating the importance of "Number of Bedrooms." One such algorithm is **K-Nearest-Neighbour Algorithm**, which is used in machine learning for both classification and regression tasks, where the goal is to predict a target variable based on a set of input features. The algorithm is particularly useful in cases where the relationship between the input features and the target variable is complex or nonlinear and cannot be easily represented by a simple mathematical function. For distance-based algorithms like KNN, the "Square Footage" feature might dominate the distance calculations, causing inaccurate nearest-neighbour classifications.

Note: We shall be looking at implementation of such algorithms in coming chapters.

To avoid this risk, **Feature scaling**, also known as **normalization**, is performed as a crucial pre-processing step in feature engineering that ensures all features (variables) have a similar scale. This is important because many machine learning algorithms, particularly those based on distance or gradient descent, are sensitive to the scale of the input features. Normalizing the features helps the algorithms converge faster and can lead to better model performance.

Let us look at some feature scaling techniques which result in different distribution of feature values but serve our purpose.

Min-Max Scaling: It is also known as min-max normalization, scales features to a specific range (usually $[0, 1]$) based on the minimum and maximum values of each feature. When we have prior knowledge that the data should be bounded within a specific range (e.g., probabilities, percentages), or we want to preserve the original distribution of the data while scaling, we can consider using this technique. However, Min-max scaling is sensitive to outliers (skewed data) and may amplify their effects.

$$X_{new} = (X - X_{min}) / (X_{max} - X_{min})$$

Here, X is the value of a feature for a particular observation and X_{max} and X_{min} are maximum and minimum values of the particular feature vector.

Z-Score Scaling: It is also known as standardization and is useful when data follows a normal distribution. It scales features to have a mean of 0 and a standard deviation of 1. It negates the effect of mean.

$$X_{new} = (X - \mu) / \sigma$$

Here, (μ) is Average or Mean of the feature values and (σ) is Standard Deviation which measures the spread of the feature values.

Let us look at a sample code which does feature scaling using above two techniques on housing data.

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

#Note: Larger Datasets can be read via CSV files
data = {
    'Number of Bedrooms': [3, 4, 2, 5],
    'Square Footage': [2000, 2500, 1500, 3000],
    'Price': [300000, 400000, 250000, 500000]
}
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
df = pd.DataFrame(data)

# Min-Max Scaling
def min_max_scaling(feature):
    min_val = feature.min()
    max_val = feature.max()
    scaled_feature = (feature - min_val) / (max_val - min_val)
    return scaled_feature

scaled_df_minmax = df.copy()
scaled_df_minmax[['Number of Bedrooms', 'Square Footage', 'Price']] = df[['Number of Bedrooms', 'Square Footage', 'Price']].apply(min_max_scaling)
print("Min-Max Scaled Data:")
print(scaled_df_minmax)
```

Min-Max Scaled Data:

	Number of Bedrooms	Square Footage	Price
0	0.333333	0.333333	0.2
1	0.666667	0.666667	0.6
2	0.000000	0.000000	0.0
3	1.000000	1.000000	1.0

```
# Z-Score Scaling
def z_score_scaling(feature):
    mean = feature.mean()
    std = feature.std()
    scaled_feature = (feature - mean) / std
    return scaled_feature

scaled_df_zscore = df.copy()
scaled_df_zscore[['Number of Bedrooms', 'Square Footage', 'Price']] = df[['Number of Bedrooms', 'Square Footage', 'Price']].apply(z_score_scaling)

print("Z-Score Scaled Data:")
print(scaled_df_zscore)
```

Z-Score Scaled Data:

	Number of Bedrooms	Square Footage	Price
0	-0.387298	-0.387298	-0.563735
1	0.387298	0.387298	0.338241
2	-1.161895	-1.161895	-1.014722
3	1.161895	1.161895	1.240216

Note: It's important to consider that scaler libraries often include additional optimizations and handling of edge cases, so using a well-established library like scikit-learn is recommended for most practical applications.

Let us try to observe the impact of not performing feature scaling on a sample housing dataset when using the k-nearest neighbours (KNN) algorithm. In this example, we would be using external libraries on a larger dataset.

```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error

# Generate synthetic housing dataset
np.random.seed(42)
n_samples = 100
square_footage = np.random.randint(1000, 3000, n_samples)
bedrooms = np.random.randint(1, 5, n_samples)
house_prices = 50000 + 200 * square_footage + 10000 * bedrooms + np.random.normal(0, 50000, n_samples)

data = {'Square Footage': square_footage, 'Number of Bedrooms': bedrooms, 'Price': house_prices}
df = pd.DataFrame(data)

# Split dataset into train and test sets
X_train, X_test, y_train, y_test = train_test_split(df[['Square Footage', 'Number of Bedrooms']], df['Price'], test_size=0.2,
random_state=42)

# KNN without feature scaling
knn_no_scaling = KNeighborsRegressor(n_neighbors=5)
knn_no_scaling.fit(X_train, y_train)
y_pred_no_scaling = knn_no_scaling.predict(X_test)
mse_no_scaling = mean_squared_error(y_test, y_pred_no_scaling)

# KNN with feature scaling
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

knn_with_scaling = KNeighborsRegressor(n_neighbors=5)
knn_with_scaling.fit(X_train_scaled, y_train)
y_pred_with_scaling = knn_with_scaling.predict(X_test_scaled)
mse_with_scaling = mean_squared_error(y_test, y_pred_with_scaling)

# Visualization
plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.scatter(y_test, y_pred_no_scaling)
plt.xlabel('Actual Prices')

```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
plt.ylabel('Predicted Prices (No Scaling)')
plt.title(f'KNN without Scaling (MSE: {mse_no_scaling:.2f})')

plt.subplot(1, 2, 2)
plt.scatter(y_test, y_pred_with_scaling)
plt.xlabel('Actual Prices')
plt.ylabel('Predicted Prices (With Scaling)')
plt.title(f'KNN with Scaling (MSE: {mse_with_scaling:.2f})')

plt.tight_layout()
plt.show()
```

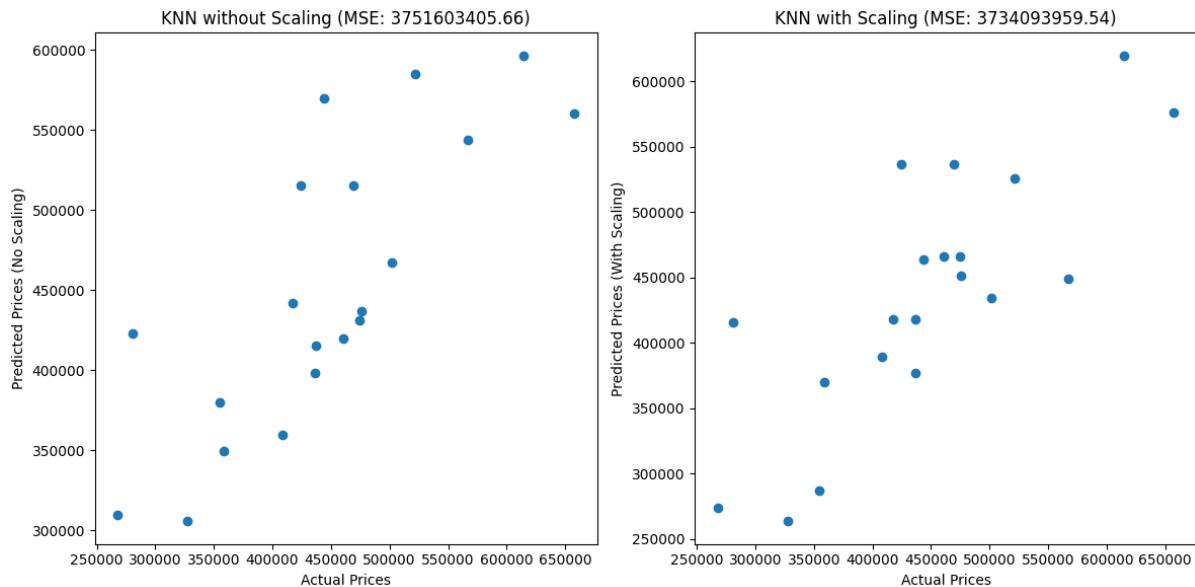


Figure 12-0-6 (Before vs After) Feature Scaling

In this example, we generate a housing dataset with two features: "Square Footage" and "Number of Bedrooms." We then split the dataset into train and test sets and perform KNN regression with and without feature scaling. Finally, we visualize the impact of feature scaling on the predicted prices using scatter plots.

The scatter plots show the relationship between actual prices and predicted prices for both scenarios. We shall notice that in the plot "KNN without Scaling," the predictions are more dispersed and less accurate compared to the plot "KNN with Scaling." This indicates that KNN is more accurate when feature scaling is applied, as it accounts for the differences in feature scales.

Note: In the example above, we have divided the dataset into "training" and "testing" data sets. It is very important not to test your model with the same data that you used for training. The ratio of the two splits should be approximately 70/30 or 80/20. Data sets have been split via rows not columns and this is very important to keep in practice. A good practice is to randomize all rows in dataset to avoid bias in models, as original dataset might be arranged in a sequential manner. Libraries like Scikit-learn provide built-in functionality to divide and shuffle datasets and we are saved from manual efforts on this.

So far, we have discussed some feature engineering techniques for numerical data, however, full treatment to such a detailed topic is outside our scope of discussion. Yet, we can look at the following map to have a gauge on possible feature engineering techniques.

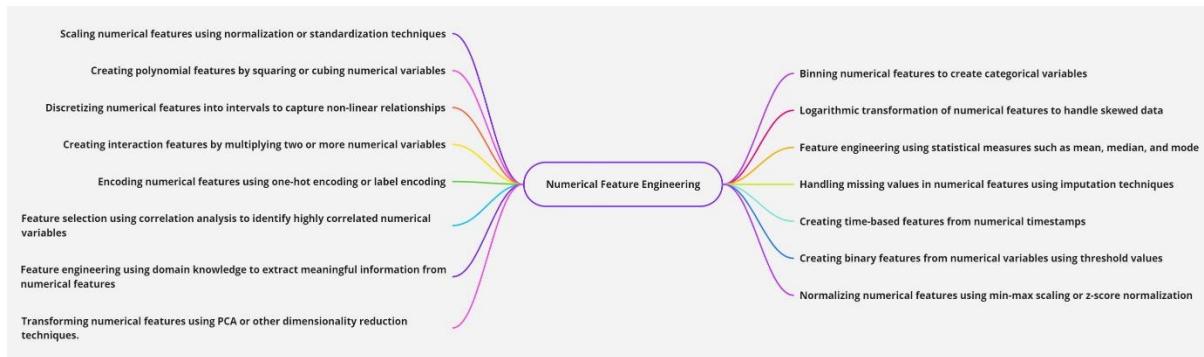


Figure 12-0-7 Possible Feature Engineering approaches

It is important to note that Tree-based models, such as decision trees and random forests, are not affected by the scale of features. These models make binary decisions based on threshold comparisons and do not rely on distance or gradient-based optimization.

13.4. Handle Missing Data

Example datasets which we have encountered so far are kind of ideal datasets with very less impurity and have values against the features present. However, in real world scenario, this is far from the case.

Dealing with missing data is a critical aspect of data pre-processing in feature engineering. Missing values can lead to biased results, inaccurate predictions, and reduced model performance. Fortunately, there are several strategies to handle missing data and minimize their impact on your analysis. Let us discuss them briefly:

Approximate Using Mode (Categorical/Binary Variables): When dealing with missing values in categorical or binary variables, one common approach is to approximate those missing values using the mode. The mode represents the most frequent value in a variable. This approach is effective for variables that have discrete categories or binary outcomes (i.e., two possible values). We replace the missing values in the variable with the mode value.

We can use this technique for variables with smaller number of distinct categories. Suppose we have a dataset of customer information, including their gender (categorical variable) and purchase history. If some gender values are missing, we can approximate the missing values using the mode, which is the most common gender among the observed data.

Approximate Using Median (Continuous Variable): When dealing with missing values in continuous variables (variables that can take any numeric value), an effective approach is to approximate those missing values using the median. The median represents the middle value of a variable when it is sorted in ascending order. We substitute the missing values in the variable with the computed median value.

This technique might not be suitable if variable's value distribution is deviating significantly from normal distribution. Consider a dataset containing information about employee salaries, including years of experience and salary amounts. If some salary values are missing, we can approximate the missing values using the median salary, which represents the middle salary amount when arranged in ascending order.

Remove Rows with missing value: This approach is used a last resort to deal with missing values in a dataset, in cases where there is very small amount of missing values.

Let us consider a sample dataset containing exam scores for students, including their age and whether they completed an exam preparation course.

Student No	Age	Course Completed	Score
1	20	Yes	89
2	22	No	73
3	19	Yes	92
4	21		78
5		No	62

We can see that some of the feature values are missing, hence let us approximate the missing values via following sample code and then look at cleansed dataset.

```
import pandas as pd
from statistics import median

# Read the dataset from an Excel file
file_path = 'student_score.xlsx'
df = pd.read_excel(file_path)

# Display the original dataset
print("Original Dataset:")
print(df)

# Approximate missing values using Mode for 'Course Completed'
mode_course_completed = df['Course Completed'].mode()[0]
df['Course Completed'].fillna(mode_course_completed, inplace=True)

# Approximate missing values using Median for 'Age'
median_age = median(df['Age'].dropna())
df['Age'].fillna(median_age, inplace=True)

# Remove rows with missing values
df.dropna(inplace=True)

# Display the cleaned dataset
print("\nCleaned Dataset:")
print(df)
```

Original Dataset:

	Student No	Age	Course Completed	Score
0	1	20.0	Yes	89
1	2	22.0	No	73
2	3	19.0	Yes	92
3	4	21.0	NaN	78
4	5	NaN	No	62

Cleaned Dataset:

Student No	Age	Course Completed	Score
------------	-----	------------------	-------

0	1	20.0	Yes	89
1	2	22.0	No	73
2	3	19.0	Yes	92
3	4	21.0	No	78
4	5	20.5	No	62

In this example, we used the mode to approximate missing values in the "Course Completed" column and the median to approximate missing values in the "Age" column. We also demonstrated the code for last resort approach of removing rows with missing values. It is up to the data analyst's discretion to decide which approach would be suitable based on feature variables.

Note: For the sake of better visualization, smaller datasets are being considered.

So far, we have discussed examples of dataset with numerical features only. However, in real world scenarios, it is very rare to get such a dataset. We usually deal with varied variety of data types and text being the most prominent. Natural Language Processing(NLP) relies heavily of feature engineering techniques based on text data.

13.5. Dealing with Text Data

Text data is abundant in various domains, including social media, customer reviews, news articles, and more. However, they are not structure like numerical features or categorical features. For example, consider the following movie review of the Batman.

"The Batman is an exhilarating and visually stunning film that redefines the superhero genre. Robert Pattinson delivers a captivating performance as the enigmatic Dark Knight. The cinematography and action sequences are top-notch, immersing the audience in the gritty atmosphere of Gotham City. The plot keeps you engaged from start to finish, with unexpected twists and turns. Director Matt Reeves has crafted a modern and intense take on the iconic character, making The Batman a must-watch for both fans and newcomers."

This is one of the sample reviews posted by a user on a web portal. Say, we are building a sentiment analysis model to determine whether a movie review is positive or negative based on the text of the review. How are we supposed to do that? We do not have any categorical features or numerical features like action to be score on 1-10 or screenplay to be selected out of 'captivating', 'funny' or 'boring'.

Challenge with text data is that it is not immediately clear how to transform them to a format suitable for Machine Learning Algorithms for further analysis. Each word in a review could potentially be a feature. Longer reviews will result in more features, leading to high-dimensional data. Consider the word "bad." In one context, it might refer to poor quality (negative sentiment), while in another context, it might describe a villainous character (neutral sentiment). Reviews might contain typos, colloquialisms, or misspellings that could confuse the model.

To mitigate the challenges posted by textual data, we start with simplest of techniques, bag-of-words, which is a list of word counts.

13.5.1. Bag-Of-Words

The Bag of Words (BoW) representation is a fundamental technique in natural language processing that serves as a foundation for various text-related tasks. BoW allows us to convert text documents into a structured format that machine learning algorithms can understand. This representation is particularly useful

Objects, Data & AI: Build thought process to develop Enterprise Applications

for simple tasks like document classification and information retrieval, where word count statistics play a significant role.

Each document is represented as a vector of word frequencies. The vocabulary consists of all unique words from the entire corpus. The value in each dimension of the vector corresponds to the frequency of a particular word in the document. The order of words is disregarded, and only the presence or absence of words matters.

Let us perform this technique of the sample movie review we just discussed.

```
import spacy
from collections import Counter

# Load the English tokenizer, tagger, parser, and NER
nlp = spacy.load("en_core_web_sm")

# Sample movie review text for "The Batman"
movie_review = (
    "The Batman is an exhilarating and visually stunning film that "
    "redefines the superhero genre. Robert Pattinson delivers a "
    "captivating performance as the enigmatic Dark Knight. The "
    "cinematography and action sequences are top-notch, immersing "
    "the audience in the gritty atmosphere of Gotham City. The "
    "plot keeps you engaged from start to finish, with unexpected "
    "twists and turns. Director Matt Reeves has crafted a "
    "modern and intense take on the iconic character, making "
    "The Batman a must-watch for both fans and newcomers."
)

# Tokenize the text using spaCy
doc = nlp(movie_review)

# Count the frequency of each token
token_freq = Counter()
for token in doc:
    token_freq[token.text] += 1

# Print the token frequencies
print("Token Frequencies:")
for token, freq in token_freq.items():
    print(f"'{token}': {freq}")

# Create a simple Bag-of-Words representation
bag_of_words = [token.text for token in doc]
print("\nBag-of-Words Representation:")
print(bag_of_words)
```

Token Frequencies:

'The': 4

```
'Batman': 2
... [output truncated for sake of simplicity]
'for': 1
'both': 1
'fans': 1
'newcomers': 1
```

Bag-of-Words Representation:

```
['The', 'Batman', 'is', 'an', 'exhilarating', 'and', 'visually', 'stunning',
'film', 'that', 'redefines', 'the', 'superhero', 'genre', '.', 'Robert',
'Pattinson', 'delivers', 'a', 'captivating', 'performance', 'as', 'the',
'enigmatic', 'Dark', 'Knight', '.', 'The', 'cinematography', 'and',
... [output truncated for sake of simplicity]
'intense', 'take', 'on', 'the', 'iconic', 'character', ',', 'making', 'The',
'Batman', 'a', 'must', '-', 'watch', 'for', 'both', 'fans', 'and',
'newcomers', '.']
```

The above code tokenizes the text, counts the frequency of each token, and creates a Bag-of-Words representation. It uses a widely popular Spacy library to tokenize the content.

The BoW representation of a positive review might have high frequencies of words like "exhilarating," "stunning," and "captivating," while a negative review could have frequent words like "dull," "disappointing," and "weak." The BoW approach would convert each review into a vector based on word frequencies, enabling classification or retrieval tasks.

While BoW is powerful for its simplicity, it has limitations. BoW does not consider the order of words or capture semantic meaning. It loses the original sequence of words. It treats all words as independent features, leading to high-dimensional sparse vectors. Additionally, stop words (common words like "the," "and," "is," etc.) might dominate the frequency count and overshadow important content words. We can use libraries like Spacy or NLTK to remove stop words from the text document while parsing.

13.5.2. Bag-Of-N-Grams

Bag-of-N-Grams is an extension of BoW technique and is used to retain information about the frequency of word sequences.

N-grams are contiguous sequences of N items from a given text. In the context of natural language processing, items are typically words, but they can also be characters. For example:

- Unigrams (1-grams): Single words in the text (e.g., "the," "quick," "brown").
- Bigrams (2-grams): Pairs of consecutive words (e.g., "the quick," "quick brown").
- Trigrams (3-grams): Triplets of consecutive words (e.g., "the quick brown," "quick brown fox").

The Bag of N-Grams treats a document as a "bag" of N-grams, disregarding their order. It represents the text by creating a vector where each element corresponds to a specific N-gram, and the value represents the frequency (or sometimes a binary presence/absence) of that N-gram in the document. The model captures the distribution of N-grams across the entire text corpus.

Let's consider a sentiment analysis task where we want to classify movie reviews as positive or negative.

Positive Review: "The movie was absolutely fantastic, and the acting was superb!"

Negative Review: "The movie was terrible, and the acting was dreadful."

Following would be resultant bigrams (2-grams).

Positive Review N-grams: ["The movie", "movie was", "was absolutely", "absolutely fantastic", "fantastic and", "and the", "the acting", "acting was", "was superb"]

Negative Review N-grams: ["The movie", "movie was", "was terrible", "terrible and", "and the", "the acting", "acting was", "was dreadful"]

Later we can create vector for each review based on presence or absence of certain bigrams. Let us look at a sample code which utilises Scikit-learn and Spacy library to create bi-grams from a set of movie reviews to train the model and later predict using test data.

```
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
import spacy

# Sample movie reviews and their corresponding sentiments
reviews = [
    ("This movie was absolutely amazing! The plot, the acting, everything was perfect.", "positive"),
    ("I can't believe how terrible this movie was. It was a waste of time.", "negative"),
    ("The cinematography was stunning, and the story kept me engaged throughout.", "positive"),
    ("I regret watching this movie. The dialogue was cheesy, and the characters were one-dimensional.", "negative"),
    ("An incredible film that left me speechless. A must-watch for all movie enthusiasts.", "positive"),
    ("I was highly disappointed by this movie. It did not live up to the hype.", "negative"),
    ("A heart-warming and touching story that brought tears to my eyes. A true masterpiece.", "positive"),
    ("Avoid this movie at all costs. I couldn't wait for it to end.", "negative"),
    ("A rollercoaster of emotions! The acting was phenomenal, and the twists were unexpected.", "positive"),
    ("I've never been so bored during a movie. The pacing was slow, and the plot was dull.", "negative")
]

# Separate reviews and sentiments
corpus = [review for review, _ in reviews]
sentiments = [sentiment for _, sentiment in reviews]

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(corpus, sentiments, test_size=0.2, random_state=42)

# Load spaCy stop words
nlp = spacy.load("en_core_web_sm")
stopwords = spacy.lang.en.stop_words.STOP_WORDS

# Create a custom tokenizer function that removes stop words
def custom_tokenizer(text):
```

```
doc = nlp(text)
tokens = [token.text for token in doc if token.text.lower() not in stopwords]
return tokens

# Create a pipeline with CountVectorizer and a Support Vector Classifier (SVC)
pipeline = Pipeline([
    ('vectorizer', CountVectorizer(ngram_range=(1, 2), tokenizer=custom_tokenizer)),
    ('classifier', SVC(kernel='linear'))
])

# Train the pipeline on the training data
pipeline.fit(X_train, y_train)

# Make predictions on the test set
y_pred = pipeline.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

Accuracy: 1.00
```

The above code performs sentiment analysis on a set of movie reviews. It uses the scikit-learn library to create a pipeline that consists of a CountVectorizer and a Support Vector Classifier (SVC). The pipeline is trained on a training set of movie reviews and their corresponding sentiments. After training, the pipeline is used to make predictions on a test set of movie reviews. The accuracy of the predictions is then calculated and printed. It is very important to observe that n-grams are expensive to compute, store, and model.

Here is a step-by-step explanation of the code:

1. Import necessary libraries:

- numpy: for numerical operations
- sklearn.feature_extraction.text.CountVectorizer: for converting text data into numerical features
- sklearn.model_selection.train_test_split: for splitting the data into training and testing sets
- sklearn.pipeline.Pipeline: for creating a pipeline of data transformation and model training
- sklearn.svm.SVC: for implementing the Support Vector Classifier
- sklearn.metrics.accuracy_score: for calculating the accuracy of predictions
- spacy: for natural language processing tasks

2. Define a list of movie reviews and their corresponding sentiments.

3. Separate the reviews and sentiments into two separate lists.

4. Split the data into training and testing sets using the *train_test_split* function. The training set contains 80% of the data, and the testing set contains 20% of the data. The *random_state* parameter is set to 42 for reproducibility.

5. Load the Spacy stop words and store them in the *stopwords* variable.

6. Define a *custom_tokenizer* function that uses Spacy to tokenize the text and remove stop words. It takes a text as input and returns a list of tokens.

7. Create a pipeline using the Pipeline class. The pipeline consists of two steps:

- *CountVectorizer*: It converts the text data into numerical features by creating a matrix of token counts. The *ngram_range* parameter is set to (1, 2) to consider both unigrams and bigrams. The *tokenizer* parameter is set to the *custom_tokenizer* function defined earlier.
- SVC: It is a Support Vector Classifier with a linear kernel, a type of supervised machine learning algorithm used for classification tasks. It is a member of the broader family of Support Vector Machines (SVMs) that can be applied to both binary and multiclass classification problems. The main idea behind SVMs, including SVC, is to find a hyperplane that best separates different classes in the feature space. This hyperplane is chosen in such a way that it maximizes the margin between the classes, which helps improve the model's generalization performance on unseen data. Detailed Discussion on SVC is out of scope for this book.

8. Train the pipeline on the training data using the *fit* method. The *X_train* and *y_train* variables contain the training reviews and sentiments, respectively.

9. Use the trained pipeline to make predictions on the test set of reviews (*X_test*) using the *predict* method. The predicted sentiments are stored in the *y_pred* variable.

10. Calculate the accuracy of the predictions by comparing the predicted sentiments (*y_pred*) with the actual sentiments (*y_test*) using the *accuracy_score* function. The accuracy is stored in the *accuracy* variable.

11. Print the accuracy of the predictions.

Note: The above program achieves a score of 1.00 for accuracy. However, in ideal world this is far from the actual results we get from our models. An accuracy of 1.0 (or 100%) suggests that the model's predictions perfectly match the true labels in the dataset. In other words, the model has achieved flawless performance and has correctly classified every instance in the test dataset.

While a perfect accuracy score might seem desirable, it can sometimes indicate potential issues or limitations with the modelling process:

- Data Imbalance: If the dataset is imbalanced (i.e., one class has significantly more samples than the other), a high accuracy might be achieved by simply predicting the majority class for all instances. This might not be a meaningful or useful result.
- Overfitting: The model might have overfitted the training data, effectively memorizing it and making perfect predictions on it. But this does not necessarily mean it will generalize well to new, unseen data.
- Data Leakage: There might be data leakage, where information from the test set has unintentionally leaked into the training process, leading to unrealistically high accuracy.
- Small or Simplified Dataset: In some cases, a small or highly simplified dataset might allow the model to achieve a perfect score, even if it's not a true representation of real-world complexity.
- Lack of Generalization: Achieving perfect accuracy on the test set doesn't guarantee that the model will perform well on new, unseen data.

We can consider other evaluation metrics, such as precision, recall, F1-score, and ROC-AUC, depending on the specific problem and dataset. Careful consideration of these factors helps avoid making overconfident conclusions based solely on accuracy.

In the code example above, we looked at text filter techniques using stop words. Let us briefly discuss some key filtering techniques used in text feature engineering.

13.5.3. Filtering Techniques

Stopwords: Stopwords are common words that are often removed from text during pre-processing because they do not carry significant meaning and can appear frequently across documents. Examples of stopwords include "the," "and," "is," "in," "of," etc. Removing stopwords can help reduce noise in the text data and improve the efficiency of text processing and analysis.

Frequency based filtering: It is a more statistical way of filtering out corpus specific common words as well as general purpose stopwords. High-frequency words are often stopwords or common terms that might not provide much discriminatory power for classification tasks. Low-frequency words might be typos, proper nouns, or domain-specific terms that could introduce noise. In practice, this technique is often combined with stopwords list.

Rare Words filtering: Rare words are words that appear very infrequently in the text corpus. They need to be filtered out as they may tend towards heavy tailed distribution. These words might be specific to individual documents or not common across the entire dataset. While rare words can sometimes carry important information or domain-specific knowledge, they can also add noise to the dataset, especially if they are typographical errors or specific to only a few instances. They incur a large computational and storage cost for not much additional gain.

Rare words can be identified using word count statistics. At times, their count is aggregated into special garbage bin and used as an additional feature. Filtering out rare words can help improve the model's generalization and reduce overfitting.

Stemming: Stemming is the process of reducing words to their base or root form. It involves removing suffixes and prefixes to obtain the core meaning of a word. For example, the stem of "running" is "run", "zeroes" is "zero" and "jumps" is "jump". Stemming is useful for reducing inflected or derived words to their common base, which can help in situations where variations of a word occur but convey similar meanings. Stemming can improve feature extraction by reducing the dimensionality of the data and making related words more comparable. Stemming is a more aggressive approach that might generate stems that are not valid words but can still help with tasks like text classification or search.

Lemmatization: It is a more sophisticated approach that reduces words to their base or dictionary form, known as the **lemma**. Unlike stemming, lemmatization takes into account the word's context and part of speech to accurately produce valid lemmas. Lemmatization aims to transform words to their meaningful and grammatically correct base forms. For example:

Original sentence: The cats are jumping over the fences.

Lemmatized sentence: the cat be jump over the fence.

Lemmatization is generally slower and more computationally intensive compared to stemming, but it provides more accurate and linguistically meaningful results. Lemmatization considers the context and part of speech of a word, making it a language-specific and context-aware process. Stemming is more language-agnostic and relies on heuristic rules.

13.5.4. Parsing and Tokenization

When we deal with textual data, they contain not only unstructured text but also semi structured text in the form of JSON, XML or HTML blobs. **Parsing** is utilized for analysing the grammatical structure of a sentence to understand its syntactic components and relationships. It involves determining the role of each word in a sentence and identifying how words relate to one another. Parsing provides insights into the

grammatical structure of the text, which can be useful for various NLP tasks, including syntactic analysis, semantic understanding, and information extraction. For example, in an email, headings like “To”, “From”, etc. need to be parsed and prevented from being coupled with normal words in statistical tasks.

After parsing a text document, it is subjected to process of **tokenization**, which is breaking a text into individual linguistic units, or tokens. Tokens can be words, phrases, or even characters, depending on the level of granularity required for analysis. It is one of the fundamental steps of text processing in NLP algorithms. Tokenizer algorithms utilize various strategy to divide the text into tokens, for example, they can look for punctuations, hash marks for social media posts, etc.

Once the text is broken into tokens, NLP algorithms try to identify meaningful combination of words that occur frequently. This is called **Collocation Extraction**. Collocations are word combinations that frequently co-occur in a language. These combinations are more meaningful than sum of their parts. For example, “strong tea” is a collocation of words “strong” and “tea”, however, individually both words have different meaning but when used together they impart a totally different meaning. While “strong” alone might refer to great physical strength and “tea” to the beverage, “strong tea” conveys a specific type of tea that is potent or intensely flavored. This combination yields a unique interpretation that cannot be derived from the meanings of the individual words.

It is important to note that collocations don't have to be sequence of words. In contrast, non-collocations like “cute girl” exhibit compositional meaning. The word “cute” independently conveys a sense of charm or attractiveness, and “girl” refers to a young girl. When combined, “cute girl” maintains the additive meaning of an adorable young girl. The words contribute their individual meanings without any significant shift in interpretation.

NOTE: Some collocations may coincide with n-grams (sequences of n words), the two concepts are not always synonymous. Not every n-gram constitutes a meaningful collocation. N-grams encompass a broader range of sequential word combinations, whereas collocations specifically emphasize the meaningfulness and non-compositionality of certain word pairings.

Collocations are useful features but extracting them can be a tricky process. Before advent of NLP algorithm, one approach which could be thought of was to predefine them, but it is computationally expensive. NLP algorithms like bag-of-n-grams have made the life easier for data scientists. One hack can be to look at frequently occurring n-grams, however, this may lead to over-populating the probable candidate of collocations as most of n-grams might not make sense. Other approach can be to use a Hypothesis test, where key idea is to understand, whether two words that appear together more often than they would by chance. Data scientists also use ‘**Chunking**’ as a replacement for n-gram approach. Python open source libraries use Part of Speech tagging for words mapping and have models for different languages.

The general idea is to turn a text into tokens and extract a disconnected set of counts. Since, sets have less structure than sequences, extraction of feature vectors is supported.

13.5.5. TF-IDF Approach

In the quest to extract meaningful information from text data, which captures the main entity and overall sentiments of the text, data scientists innovated a twisted approach over bag-of-words, known as TF-IDF approach.

The **term frequency (TF)** of a word in a document measures how often that word appears in the document. It's calculated by dividing the number of times the word appears in the document by the total number of words in that document.

For example, if the word "apple" appears 5 times in a document with a total of 100 words, the TF for "apple" would be 0.05 (5/100).

The **inverse document frequency (IDF)** of a word assesses how unique or rare that word is across a collection of documents. It's calculated by taking the logarithm of the total number of documents divided by the number of documents containing the word, and then adding 1 to avoid division by zero. If the word "apple" appears in 100 out of 1,000 documents, the IDF for "apple" might be around 1.61.

While discussing Log Transforms, we observed how logarithms turn 1 into 0 and makes large numbers small. A word which appears in every single document will be zeroed out and word which appear in fewer documents will have a larger count than before. TF-IDF scores help us identify significant terms that contribute to the overall content and theme of each document within the context of the entire collection.

Suppose we have a collection of movie plot summaries and we want to use TF-IDF to analyse the importance of words in each summary.

Movie 1 - "The adventurous boy and his dog set out on a quest to find a hidden treasure deep in the jungle."

Movie 2 - "A young scientist accidentally discovers a time machine and travels to different historical eras."

Movie 3 - "In a futuristic world, a group of rebels fights against a powerful AI that has taken control of society."

Let's focus on the word "adventurous" and calculate its TF-IDF score for each movie:

Term Frequency (TF):

Movie 1: "adventurous" appears once in 15 words. $TF = 1/15 \approx 0.067$

Movie 2: "adventurous" does not appear. $TF = 0$

Movie 3: "adventurous" does not appear. $TF = 0$

Inverse Document Frequency (IDF):

Total number of documents (movies): 3

Number of documents containing "adventurous": 1 (Movie 1)

$IDF = \log(3/1) + 1 \approx 1.477$

TF-IDF Score for "adventurous" in each movie:

*Movie 1: $(0.067) * 1.477 \approx 0.099$*

*Movie 2: $(0) * 1.477 = 0$*

*Movie 3: $(0) * 1.477 = 0$*

In this example, "adventurous" has a higher TF-IDF score in Movie 1 compared to the other movies. This indicates that the word is relatively more important in conveying the theme of adventure in Movie 1. Similar process can be repeated for other words to gauge their importance or relevance with respect to movie plots.

TF-IDF is a transformation that adjusts the word counts in a way that reflects their importance in the corpus. It can stretch the counts of important words while compressing the counts of less informative words. This has the effect of giving more weight to words that are distinctive and indicative of the document's content, potentially leading to more effective feature representations. By compressing the counts of uninformative words close to zero, TF-IDF effectively filters out words that do not contribute meaningfully to the document's content. This reduction in dimensionality can enhance the model's ability to focus on the significant features, potentially leading to improved accuracy.

13.6. Dealing with Categorical Variables

Datasets used in Machine Learning models are comprised of wide range of information in structured, unstructured, semi-structured, numerical or text format. For example, if a restaurant seeks to maintain a dataset of its customer based on their cuisine preference, along with other information of the customer, the "cuisine" feature variable in the dataset will have values limited to the range of cuisines the restaurant has got to offer. Let us observe a sample dataset for better understanding.

Customer	Age	Monthly Visit Count	Cuisine
Rita	34	5	Italian
Rahul	40	2	Thai
Jack	35	4	Indian
Seema	25	7	Chinese
Thomas	40	4	Indian
Mike	28	3	Thai

In the above dataset, Age and Monthly Count Visit are continuous variables, while Customer and Cuisine are categorical variables.

Categorical variables are variables that can take on a limited number of values such as the color of a car (red, blue, green, etc.), the day of the week (Monday, Tuesday, Wednesday, etc.), or the gender of a person (male, female). These variables do not have a natural numerical order or mathematical meaning.

Hence, while working with machine learning algorithms, categorical variables need to be encoded into numerical values in order for the algorithms to understand them. Encoding techniques are used to convert categorical variables into numerical values while preserving their inherent relationships or characteristics.

One-hot encoding: It is one of the most common encoding techniques for categorical variables. In one-hot encoding, a new column is created for each unique category in the variable. The value in each column is 1 if the observation belongs to that category and 0 otherwise. For example, in our restaurant dataset, if "cuisine" variable has four categories (Italian, Thai, Indian and Chinese), then four new columns will be created. An observation with the value "Indian" in the original variable will have the value 0 in the first and second column, 1 in the third column, and 0 in the fourth column.

Customer	Cuisine_Italian	Cuisine_Thai	Cuisine_Indian	Cuisine_Chinese
Rita	1	0	0	0
Rahul	0	1	0	0
Jack	0	0	1	0
Seema	1	0	0	1
Thomas	0	0	1	0

Mike	0	1	0	0
------	---	---	---	---

One hot encoding can be performed using explicit programming as well as using ‘panda’ library. Let us look at the sample code.

```
import pandas as pd

# Create the dataset
data = {
    'Customer': ['Rita', 'Rahul', 'Jack', 'Seema', 'Thomas', 'Mike'],
    'Cuisine': ['Italian', 'Thai', 'Indian', 'Chinese', 'Indian', 'Thai']
}

# Convert the dataset to a DataFrame
df = pd.DataFrame(data)

# Perform one-hot encoding using pandas
one_hot_encoded = pd.get_dummies(df, columns=['Cuisine'], prefix=['Cuisine'])

# Print the one-hot encoded DataFrame
print(one_hot_encoded)
```

In the code above, we use the `pd.get_dummies()` function to perform one-hot encoding on the "Cuisine" column. The prefix parameter adds a prefix to the new columns.

Label encoding: It is another common encoding technique for categorical variables. In label encoding, each category is assigned a unique integer value. For example, the categories “Italian”, “Indian”, “Thai” and “Chinese” could be assigned the values 0, 1, 2, and 3, respectively. Label encoding is a simpler and less computationally expensive technique than one-hot encoding, but it does not preserve the order of the categories. This can be a problem if the order of the categories is meaningful, such as if the categories represent different levels of severity.

Current restaurant dataset example was a very simple one with one feature variable having only one value. However, in actual scenarios, a customer may well have more than one cuisine as preference. In this case, we perform **Multi-label binarization**, which extends the concept of one-hot encoding to handle multiple categories. Consider the modified dataset:

Customer	Age	Monthly Visit Count	Cuisine
Rita	34	5	Italian, Indian
Rahul	40	2	Thai, Chinese
Jack	35	4	Indian, Chinese
Seema	25	7	Chinese, Thai
Thomas	40	4	Indian, Italian
Mike	28	3	Thai, Indian

Now, let us look at the sample code performing Multi-label binarization using scikit-learn library in Python.

```
import pandas as pd
from sklearn.preprocessing import MultiLabelBinarizer
```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
# Create the dataset
data = {
    "Customer": ["Rita", "Rahul", "Jack", "Seema", "Thomas", "Mike"],
    "Age": [34, 40, 35, 25, 40, 28],
    "Monthly Visit Count": [5, 2, 4, 7, 4, 3],
    "Cuisine": [[["Italian", "Indian"], ["Thai", "Chinese"], ["Indian", "Chinese"],
                 ["Chinese", "Thai"], ["Indian", "Italian"], ["Thai", "Indian"]]]
}
df = pd.DataFrame(data)

# Instantiate the MultiLabelBinarizer
mlb = MultiLabelBinarizer()

# Apply multi-label binarization to the Cuisine column
cuisine_encoded = mlb.fit_transform(df["Cuisine"])

# Create a DataFrame with the binarized cuisine columns
cuisine_df = pd.DataFrame(cuisine_encoded, columns=mlb.classes_)

# Concatenate the original DataFrame and the cuisine DataFrame
encoded_df = pd.concat([df, cuisine_df], axis=1)

# Drop the original Cuisine column
encoded_df.drop(columns="Cuisine", inplace=True)

# Display the resulting encoded DataFrame
print(encoded_df)
```

Following would be the output:

Customer	Age	Monthly Visit Count	Chinese	Indian	Italian	Thai
Rita	34	5	0	1	1	0
Rahul	40	2	1	0	0	1
Jack	35	4	1	1	0	0
Seema	25	7	1	0	0	1
Thomas	40	4	0	1	1	0
Mike	28	3	0	1	0	1

Some other key encoding techniques are Target encoding and Frequency encoding.

Target encoding is a more sophisticated encoding technique that takes into account the relationship between the categorical variable and the target variable. In target encoding, the mean value of the target variable is calculated for each category. The value of the categorical variable in the new encoded column is then set to the mean value of the target variable for that category. Target encoding can be a very effective way to encode categorical variables, but it can also be computationally expensive, especially if the variable has a large number of categories.

Frequency encoding is a simple and efficient encoding technique that counts the number of times each category appears in the dataset. The value of the categorical variable in the new encoded column is then set to the frequency of that category. Frequency encoding is a good choice for categorical variables with a large number of categories, as it does not create a large number of new columns. However, it does not take into account the order of the categories or the relationship between the categorical variable and the target variable.

The datasets which we have used till now had very limited number of categorical variables, but in real world datasets, categorical variables can be in a very large number. Suppose a telecom company maintaining the dataset of its consumers contains a categorical variable “Location”. A telecom company operating nation wide will have customers across cities and if we perform traditional encoding techniques on it, the dataset will get polluted with large number of new features. To over come this challenge, we consider an approach called Feature Hashing.

Feature Hashing is a technique that converts categorical variables into a fixed number of binary features. This is done by using a hash function to map each category to a unique integer value. The value of the categorical variable in the new encoded column is then set to 1 if the hash function returns the value 0, and 0 otherwise.

For example, if a categorical variable has 100 cities, then feature hashing will create 100 new binary features. An observation with the value "New Delhi" in the original variable will have the value 1 in the feature that corresponds to the hash value of "New Delhi", and 0 in all the other features.

Let us look at a comparatively smaller dataset though, for the sake of clarity in observation:

Customer	City
Anil	New Delhi
Raj	Kolkata
Ram	Ayodhya
Hemanth	Bengaluru
Nikhil	Ranchi
Jasmine	New Delhi

In the above dataset, we have 5 different cities. To represent them using feature hashing, we should be using a Uniform Hashing Function, which would ensure roughly same number of bins are created. In Feature Hashing, collisions can occur, and different categories might be mapped to the same hash bucket. Hence, we have to experiment with different hashing functions or use large number of buckets to avoid collision. Collisions can impact the model's ability to distinguish between different categories, potentially leading to decreased model performance. Scikit-learn library provides the functionality of “FeatureHasher” to perform this operation.

Customer	City_Hash_1	City_Hash_2	City_Hash_3
Anil	1	0	0
Raj	0	0	1
Ram	1	0	0
Hemanth	0	1	0
Nikhil	0	0	1
Jasmine	1	0	0

Note: Feature Hashing introduces the possibility of collisions, as observed in this example where different cities might have been mapped to the same hash bucket. This potential loss of information is a trade-off

when using Feature Hashing to handle high-cardinality categorical variables. It is a technique used for dimensionality reduction and efficient storage of high dimensional feature vectors.

Since, one hot encoding or binary encoding techniques result in High dimensionality and Hashing technique outputs are difficult to interpret, the need of the hour is to use an alternative that allows easy inspection of the model to directly associate each attribute with likelihoods. It requires aggregating past data in a set of count table, where each table associates an attribute or combination with its historical counts.

Bin Counting converts a categorical variable into statistics about value. Let us imagine a dataset which contains information about a customer visiting a restaurant on a particular day of the week. The day of the week variable has seven categories (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, and Sunday). Bin counting can be used to divide the categories into three bins:

- Weekdays: This bin will include the observations for Monday, Tuesday, and Wednesday.
- Thursday and Friday: This bin will include the observations for Thursday and Friday.
- Weekend: This bin will include the observations for Saturday and Sunday.

An observation with the value "Monday" in the day of the week variable would be counted in the "Weekdays" bin. An observation with the value "Tuesday" in the day of the week variable would also be counted in the "Weekdays" bin. An observation with the value "Wednesday" in the day of the week variable would also be counted in the "Weekdays" bin.

If there is a total of 3 observations that were counted in the "Weekdays" bin. Therefore, the value of the categorical variable in the new encoded column would be set to 3 for each observation that was counted in the "Weekdays" bin.

In other words, the new encoded column would have the value 3 for each observation that visited the restaurant on a Monday, Tuesday, or Wednesday. This would help the machine learning algorithm to understand that these days are more likely to be weekdays than weekends.

The best way to determine the number of bins to use is to experiment with different values and see which one results in the best performance for the machine learning algorithm.

13.7. Dimensionality Reduction using PCA

Machine Learning or real life, we try to prune away unnecessary information and retain the important bits to focus on the bigger picture. As datasets become more complex, it becomes increasingly difficult to analyse them using traditional methods. Not all the features in a dataset are informative or useful. For example, consider image processing, where pixel measurements form a high dimensional vector space. However, most of the images are highly compressed, which basically means that the relevant information is captured in much lower dimensional space. If a dataset has 'n' number of features, we call the dataset n-dimensional. It is easy to visualize in 2-D or 3-D, but beyond that visualization will get blurry. In real life, data is multi-dimensional and sometimes features of data can be correlated with each other.

The techniques discussed so far have been focused towards removing unnecessary bit of information from datasets based on methods without referencing the data. For instance, frequency-based filtering is a rule-based technique that removes features with counts below a certain threshold. This procedure can be determined solely based on the properties of the features themselves and doesn't require specific insights from the data they represent. Removing features based solely on frequency thresholds or manual feature selection or other measures without referencing data, might discard potentially useful information.

In order to make sense of data, we try to reduce dimensions using model-based techniques which reference the data and try to gain insight from it. **Principal Component Analysis (PCA)** is one such technique which makes use of applied Linear Algebra and focuses on dimensionality reduction of a dataset containing a large number of interrelated variables, while retaining as much as possible variation present in the dataset.

PCA aims to capture and extract the most significant patterns and variations present in the original data. By transforming the data into a new set of orthogonal features (**principal components**), PCA identifies the directions along which the data exhibits the most variability. These principal components represent the essential information that contributes to the observed variations in the dataset. These combinations are chosen in a way that the first principal component explains the maximum possible variance, the second principal component explains the maximum variance remaining after the first component, and so on. Each principal component is orthogonal to the others, ensuring that they are uncorrelated.

By reducing the number of features, less relevant and noisy features are discarded. This reduction in noise can mitigate the risk of overfitting, as the model becomes less likely to learn from irrelevant variations and focuses on capturing the most significant patterns in the data. It also helps in speeding up the training of a ML algorithm. High-dimensional data can lead to increased computational complexity during model training. By reducing the number of features through PCA, the training process becomes more efficient and faster. Fewer features require fewer computations and less memory, resulting in quicker training times and more efficient use of computing resources. Moreover, this reduction in dimensions simplifies data visualization, enabling the creation of scatter plots, heatmaps, and other visualizations that are easier to interpret and communicate. Such simplified visualizations can aid in understanding patterns, clusters, and relationships in the data.

13.7.1. Mathematical explanation of PCA

Principal Component Analysis (PCA) can be interpreted as a statistical application of the Singular Value Decomposition (SVD) on the covariance matrix of the data. SVD is a matrix factorization technique that decomposes a matrix into three component matrices, which can be leveraged to reveal the underlying structure of the data. It provides a systematic way to capture low dimensional approximation of high dimensional data in terms of relevant or dominant patterns present in the dataset.

Suppose we want to analyse a dataset X , which can be represented in a matrix form, as below.

$$X = \begin{bmatrix} \vdots & \vdots & \vdots & \vdots \\ X_1 & X_2 & \dots & X_n \\ \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

Here, different columns represent features or measurements from datasets or experiments.

The SVD is a unique matrix decomposition that exists for every complex valued matrix X .

$$X = U\Sigma V^T$$

Here, X is the original matrix that we want to decompose. It could represent data, a transformation, or any other type of matrix.

U is an orthogonal matrix (i.e., its columns are mutually perpendicular unit vectors). It represents the left singular vectors of X and captures the relationships between the rows of X .

Σ is a diagonal matrix containing the singular values of X . Singular values are non-negative numbers that indicate the importance of each singular vector in U and the corresponding singular vector in V^T . They are usually arranged in decreasing order along the diagonal of Σ .

V^T is the transpose of an orthogonal matrix V . It represents the right singular vectors of X and captures the relationships between the columns of X .

PCA can be seen as a special case of SVD, where the data matrix X is centred (mean-subtracted) and standardized before applying SVD. **Standardization** is done to ensure that all features have the same scale and prevents any feature from dominating the analysis. The resulting orthogonal matrix V in SVD corresponds to the principal components in PCA. The matrix U Contains the left singular vectors of X , which correspond to the relative alignment of the original data points with the principal components. The projected data in the principal component space can be obtained by matrix $U\Sigma$. The rows of $U\Sigma$ represent each data point in the new coordinate system defined by the principal components. Each principal component is a linear combination of the original features and is orthogonal (uncorrelated) to the other components. Ultimately, we get a hierarchical representation of the data in terms of a new coordinate system defined by dominant correlations within the data.

Although, it provides a robust and stable decomposition, even for matrices that might not be perfectly symmetric, SVD can be computationally expensive, especially for large datasets, as it involves calculating all eigenvalues and eigenvectors. This is because the eigenvalues and eigenvectors of a matrix are the roots of a polynomial equation, which can be a very difficult problem to solve.

Hence, a more computationally efficient approach is to perform PCA using only eigen vectors and calculating covariance matrices. Eigenvector-based PCA is often computationally more efficient than SVD, especially when the number of features is much larger than the number of observations. It may require less memory and computational resources compared to SVD. However, we must always keep in mind the limitation of this approach as it is only applicable with covariance matrix. SVD based approach is generally more numerically stable and less prone to precision errors.

13.7.2. PCA Implementation

Let us now try to implement PCA on a dataset using eigen-based approach. Although, libraries like scikit-learn have built-in models to perform PCA on a given dataset, we would take the route of explicit programming to understand the internal details of the implementation. Following is the implementation of PCA on *iris dataset*, widely available on internet. It contains information about various attributes of different species of iris flowers.

The Iris dataset was introduced by the British biologist and statistician Ronald A. Fisher in 1936. It consists of measurements of four features (attributes) of iris flowers from three different species: Iris setosa, Iris versicolor, and Iris virginica. The four features are:

- Sepal Length: The length of the sepal (a leaf-like structure at the base of the flower).
- Sepal Width: The width of the sepal.
- Petal Length: The length of the petal (a colourful structure that attracts pollinators).
- Petal Width: The width of the petal.

Each species of iris has its own distinct characteristics, and the goal is often to use the feature measurements to classify the flowers into their respective species. By applying PCA to the Iris dataset, we can gain a better understanding of the data's structure, relationships, and patterns. Visualizing the data in a reduced-dimensional space can reveal clusters, separations, and potential overlaps between different species of iris flowers. By projecting the data onto the principal components, one can compare the distribution of different iris species in the reduced-dimensional space. This might reveal how well-separated the species are or if there's overlap between them.

Let us have a glance on the dataset.

Species_No	Petal_width	Petal_length	Sepal_width	Sepal_length	Species_name
1	0.2	1.4	3.5	5.1	Setosa
1	0.2	1.4	3	4.9	Setosa
1	0.2	1.3	3.2	4.7	Setosa
1	0.2	1.5	3.1	4.6	Setosa
2	1.4	4.7	2.9	6.1	Versicolor
2	1.3	3.6	2.9	5.6	Versicolor
2	1.4	4.4	3.1	6.7	Versicolor
3	2.2	6.7	3.8	7.7	Verginica
3	2.3	6.9	2.6	7.7	Verginica
3	1.5	5	2.2	6	Verginica

13.7.2.1. Steps to perform in PCA

The assumption of linear correlations between features implies that PCA is most effective when relationships between features are linear or can be approximated as such. PCA aims to identify common factors in input dataset, seeking to capture underlying structure or patterns in a more compact representation.

PCA does not directly interpret correlation between features. It identifies the direction along which data varies the most. When performing PCA, correlation or covariance matrix is used as input. They are used to compute eigen vectors and eigen values.

Using correlation matrix can be advantageous when scales of features differ significantly, or emphasis is on relationship between features rather than their individual variances. If scales are similar and focus is on variability, co-variance matrix is used. The correlation matrix is a standardized covariance matrix, which means that the variances of the variables have been normalized.

Standardize the data: Given a dataset with n observations and p features, the first step is to standardize the data by subtracting the mean and dividing by the standard deviation for each feature. This ensures that all features have the same scale and prevents any feature from dominating the analysis.

$$\text{Standardized feature, } z_{ij} = (x_{ij} - \mu_j)/\sigma_j$$

Where x_{ij} is the j^{th} feature value of the i^{th} observation, μ_j is the mean of the j^{th} feature, and σ_j is the standard deviation of the j^{th} feature.

Calculate the Covariance Matrix: The covariance matrix is computed from the standardized data to capture the relationships between features.

$$\text{Covariance between Features } j \text{ and } k, \text{ cov}(X_j, X_k) = \frac{1}{n-1} \sum_{i=1}^n (z_{ij} - \bar{z}_j)(z_{ik} - \bar{z}_k)$$

Where z_{ij} and z_{ik} the standardized values of features j and k for observation i ; and $(\bar{z}_j \bar{z}_k)$ are respective means.

$$\text{Covariance matrix, } C = \begin{bmatrix} cov(x_1, x_1) & cov(x_1, x_2) & \dots & cov(x_1, x_p) \\ cov(x_2, x_1) & cov(x_2, x_2) & \dots & cov(x_2, x_p) \\ \vdots & \vdots & \ddots & \vdots \\ cov(x_p, x_1) & cov(x_p, x_2) & \dots & cov(x_p, x_p) \end{bmatrix}$$

Here, we try to understand how the variables of the input data set are varying from the mean with respect to each other, or in other words, to see if there is any relationship between them.

Since the covariance of a variable with itself is its variance ($cov(x_1, x_1) = var(a)$), in the main diagonal (Top left to bottom right) we actually have the variances of each initial variable. And since the covariance is commutative ($cov(x_1, x_2) = cov(x_2, x_1)$), the entries of the covariance matrix are symmetric with respect to the main diagonal, which means that the upper and the lower triangular portions are equal.

Covariances value which are present as entries of the matrix tell us about correlation of two variables. If the value is positive, the two variables are correlated, which means they increase or decrease together. If the value is negative, this implies that one increases when the other decreases (inverse correlation).

Compute Eigenvectors and Eigenvalues: The next step is to compute the eigenvectors and eigenvalues of the covariance matrix. Eigenvectors represent the directions (principal components) along which the data has the most variance, and eigenvalues quantify the amount of variance along each eigenvector.

$$C \cdot v = \lambda \cdot v$$

Where C is the covariance matrix, v is the eigenvector, and λ is the eigenvalue.

Eigen vectors represent the principal components and eigen values indicate the amount of variance explained by each principal component. Eigen vector with higher eigen value captures the direction of maximum variability in the data.

Principal components are new variables that we create by combining the original variables in a linear manner. Imagine we have several attributes (variables) that describe an object, like its height, weight, age, and so on. PCA allows us to construct new attributes (principal components) that are mixtures of these original attributes. PCA is designed to create these new principal components in a way that they are uncorrelated with each other. In other words, they are independent and don't carry redundant information. This is valuable because it simplifies the data representation and makes it easier to analyse.

If a principal component has high loadings (absolute value) for a set of features, it suggests that these features are strongly correlated or have similar patterns of variation.

Selection of Principal Components: PCA aims to maximize the amount of information packed into the first principal component. This component captures the most significant patterns or trends in the data. The second principal component captures the remaining patterns that are uncorrelated with the first component, and so on. This hierarchical arrangement ensures that we don't lose much information as we move to subsequent components. We achieve dimensionality reduction by discarding the components with low information and considering the remaining components as our new variables.

Transform the Data: The final step is to project the standardized data onto the chosen principal components to obtain the reduced-dimensional representation.

$$Y = X \cdot V$$

Where Y is the transformed data matrix, X is the standardized data matrix, and V contains the eigenvectors (principal components) as columns.

In summarized form, PCA involves the computation of eigenvectors and eigenvalues of the covariance matrix to identify the principal components, which represent the most significant directions of variance in the data. These principal components can then be used to transform the data into a lower-dimensional space, achieving dimensionality reduction while retaining the essential information in the dataset.

13.7.2.2. Program to perform PCA

Let us look at a sample JavaScript program to perform PCA on given dataset. The full code for program below can be found at this link:

https://github.com/reeshabh90/Data_Cluster_Tutorial/tree/main/Program/PCA

```
import pkg from 'xlsx';
const { readFile, utils } = pkg;
import { multiply, transpose, divide, eigs, matrix } from "mathjs";
import { calculateSDev, calculateMeanofColumns, getStandardizedData, transposeDatamatrix } from "./util.js";

// Read Excel file with training data
const workbook = readFile('PCA.xlsx');
const sheetName = workbook.SheetNames[0];
const worksheet = workbook.Sheets[sheetName];
const trainingData = utils.sheet_to_json(worksheet, { header: 1, defval: null });

/**
 * Calculate the mean and standard deviation
 */
// Extract the numeric data
const numericData = trainingData.slice(1)
  .map(row => row.slice(0, row.length - 1))
  .filter(cell => cell !== null).filter(row => row.length != 0);

//console.log('Numeric Data', numericData)
/**
 * Transpose training data
 */
const transposedData = transposeDatamatrix(numericData);

/**
 * Calculate mean of each column
 */
const means = calculateMeanofColumns(transposedData);
//console.log('means', means);
/**
 * Calculate standard deviation of each column
 */
const sdevs = calculateSDev(transposedData);
//console.log('sdevs', sdevs);
/**
 * Standardize the data using mean and standard deviation
 */

```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
const standardizedData = getStandardizedData(numericData, means, sdevs);

//console.log('Std Data', standardizedData);

// Calculate the covariance matrix of the standardized data
const X = matrix(standardizedData);
const Xt = transpose(X); // Transpose X
const n = standardizedData.length;
const covarianceMatrix = divide(multiply(Xt, X), n - 1);

// Calculate the eigenvalues and eigenvectors of the covariance matrix
const { values: eigenvalues, vectors: eigenvectors } = eigs(covarianceMatrix);
// Function to find the sorted index of Eigen Value array
const findIndexSeq = a => [...a.keys()].sort((b, c) => a[c] - a[b])
// Sort the eigenvalues and eigenvectors in descending order of eigenvalue
const sortedIndices = findIndexSeq(eigenvalues.toArray())
// Use the index array to sort the eigenvectors array
const sortedEigenvectors = sortedIndices.map(i => eigenvectors.toArray()[i])
const k = 1; // No of principal components
//console.log('Sorted Eigen Vector', sortedEigenvectors)
// Calculation of Principal Components:
/**  
The reason for transposing standardized data is because the shape of the  
standardized data is (n x m),  
where n is the number of samples and m is the number of features.  
However, the eigen vectors obtained from the covariance matrix have a shape of (m x m).  
Therefore, in order to perform the multiplication,  
we need to transpose the standardized data to match the shape of the eigen vectors.  
  
The multiplication between the eigen vector matrix and the transposed standardized data results  
in a new matrix where each row represents a principal component and each column represents a sample.  
This is because each principal component is a linear combination of the original features,  
and the weightings for each feature are contained within the rows of the eigen vector matrix.  
Therefore, each row in the new matrix represents the scores of each sample  
along a particular principal component.  
*/  
const principalComponents = sortedEigenvectors.slice(0, k).map((vector) => {  
  return multiply(vector, transpose(standardizedData));  
});  
console.log(principalComponents)
```

13.7.3. PCA Visualization

Having explicitly programmed our way to PCA implementation would definitely build up some insight, but there is nothing like a good visualization to grasp this intricate concept. Let us start with some EDA on iris dataset.

```
import numpy as np
import pandas as pd
```

```

from sklearn.model_selection import train_test_split
from sklearn.decomposition import PCA
import seaborn as sns
import matplotlib.pyplot as plt

url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
features = ['sepal length', 'sepal width', 'petal length', 'petal width', 'target']
# load dataset into Pandas DataFrame
df = pd.read_csv(url, names = features)
df.head()

```

	sepal length	sepal width	petal length	petal width	target
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Figure 12-0-8 Data Sample

```

sns.countplot(
    x='target',
    data=df)
plt.title('Iris targets value count')
plt.show()

```

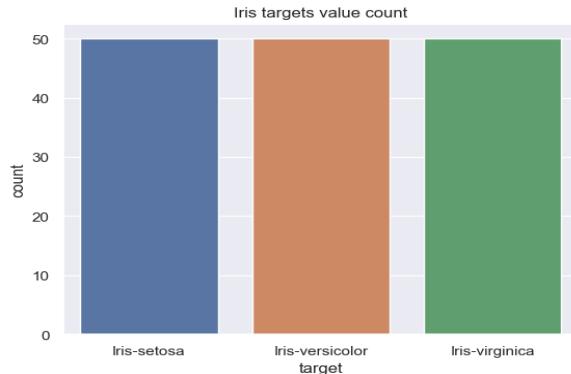


Figure 12-0-9 Plot showing Count of each species

Observation: There are 3 species of flowers and each have 50 records.

We can perform further EDA feature wise on the three species and observe the variance feature wise.

Objects, Data & AI: Build thought process to develop Enterprise Applications

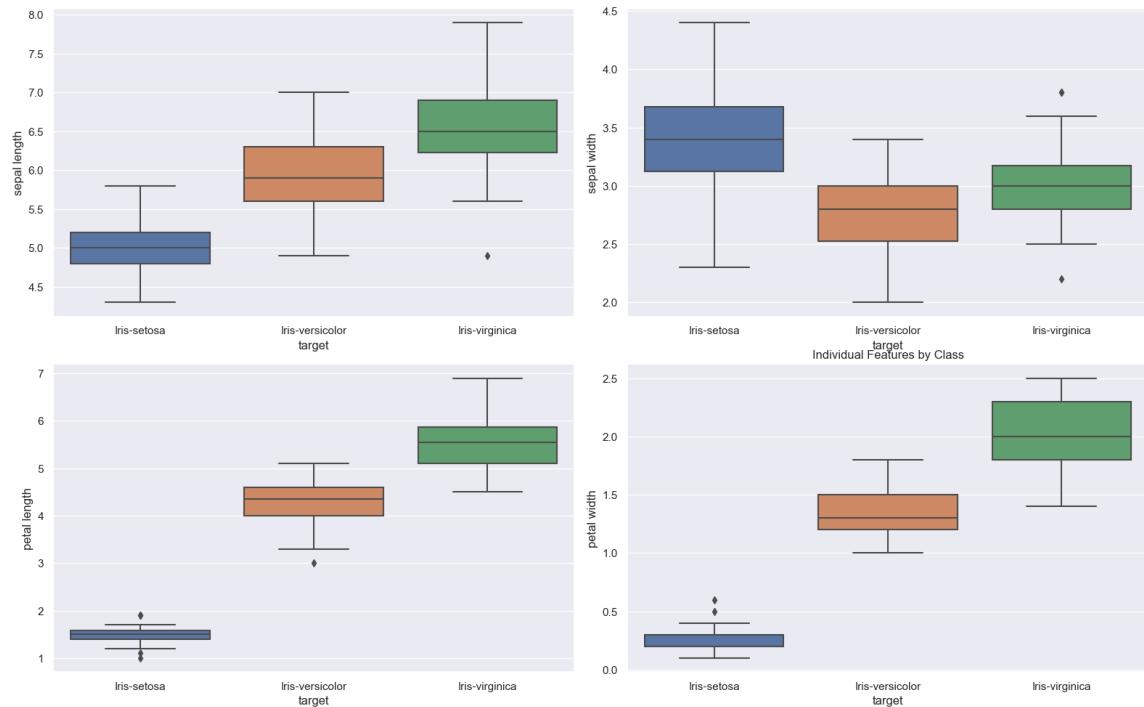


Figure 12-0-10 Plot showing variation of features of each species

```

from sklearn.preprocessing import StandardScaler
X = df[features]
X.drop('target', axis=1, inplace=True)
y = df['target']
X

# data scaling
x_scaled = StandardScaler().fit_transform(X)
from sklearn.decomposition import PCA
pca = PCA(n_components=3)
pca_features = pca.fit_transform(x_scaled)
pca_df = pd.DataFrame(
    data=pca_features,
    columns=['PC1', 'PC2', 'PC3'])
pca_df['target'] = y
plt.bar(
    range(1,len(pca.explained_variance_)+1),
    pca.explained_variance_
)

plt.ylabel('Explained variance')
plt.title('Feature Explained Variance')
plt.show()

```



Figure 12-0-11 Plot showing variance captured by Principal Components

Plot the explained variance (Eigen Value) to see the variance of each principal component feature after performing PCA using scikit-learn library. When using PCA, one must address the question of how many principal components to use. This decision can be taken using heuristics based on variance or spectrum of data matrix. Clearly, we can see the first two components cover the maximum variance of the given dataset, so we can discard the third component.

Variance is a statistical measure that describes the spread or dispersion of a set of values. In the context of dataset, it indicates how much the data points deviate from the mean. High variance implies that the data points are widely spread out, while low variance suggests that the data points are closely clustered around the mean. The **explained variance ratio** is the proportion of the total variance that each principal component explains.

When we reduce the dimensionality of dataset using PCA, we project it onto a lower-dimensional space. This projection inevitably results in some loss of information because we are capturing only the most important aspects of the dataset. The goal of PCA is to retain as much variance (and thus information) as possible while reducing the dimensionality.

From the plot above it is evident that first component contains majority of the variance captured from the dataset. And by combining the first two components, we reduce the 4-dimensional dataset to 2-D features using the two principal components. We can get the variance per component value as well.

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

sns.lmplot(
    x='PC1',
    y='PC2',
    data=pca_df,
    hue='target',
    fit_reg=False,
    legend=True
)
```

```
plt.title('2D PCA Graph')
```

```
plt.show()
```

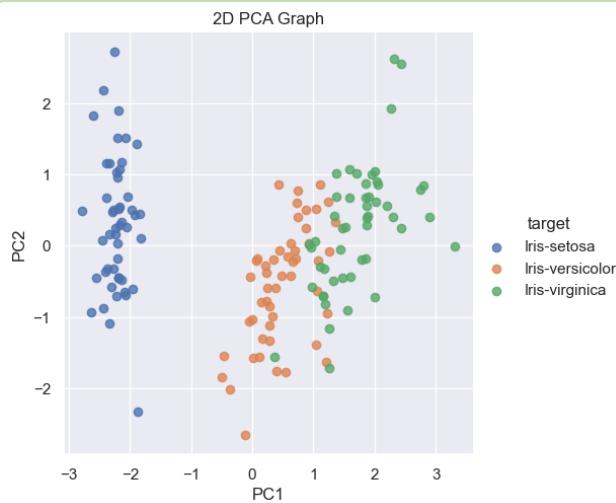


Figure 12-0-12 Principal Components mapped with each other

We have plotted the first two principal components with each other. We can clearly see the three species are separated from each other in the plot above.

13.7.4. PCA Limitations

There is a trade-off between reducing dimensionality and retaining information. By selecting a certain number of principal components, we can strike a balance between reducing the data's dimensionality and preserving a satisfactory amount of information. The explained variance helps us understand how much of the original information is retained by each principal component.

While PCA operations are powerful for reducing dimensionality and uncovering underlying structures, they often lack a direct and intuitive human-readable interpretation. Principal components and the vectors projected onto them can take positive or negative values. This means that the interpretation of each component isn't as straightforward as saying "higher values mean this" and "lower values mean that". Moreover, principal components have a directional aspect – a positive value might represent a certain pattern, while a negative value might represent the opposite pattern.

The interpretability of a model is crucial, especially in contexts like finance for applications like stock recommenders, where decisions have significant real-world consequences. If analysts cannot provide a clear and understandable explanation for the factors learned through PCA, it can erode trust in the model's recommendations. If the results of a model are too complex to be translated into understandable and justifiable decisions, its utility diminishes. Models that are not transparent and interpretable may not be adopted by decision-makers who need to be able to confidently explain and defend their choices.

13.8. Working with Images

Machine Learning - 101

Figure 12-0-13 A sample image

Since, we have now learned how to make computers understand numbers and texts, our computer should be able to read the above text (comprising of letters, symbols, and numbers) from the given image, No?

Well, modern computers indeed can perform image recognition or classification, but to achieve this progress, the subject of Machine Learning and Computer Vision required decades of research by numerous data scientists and programmers. Let us try to understand, why tasks like image classification present a far greater challenge for computers than text classifications.

Unlike languages like English, where individual words carry inherent semantic meaning, images are composed of pixels that lack the same level of inherent semantic context. This distinction underscores the complexity of feature extraction and engineering in the domain of images, as highlighted in the preceding text. In an image, the basic unit of information is a pixel, which essentially represents a color value at a specific location. While pixels are crucial for forming the visual content of an image, they do not inherently convey higher-level semantic meaning on their own. Consequently, translating these pixels into meaningful features that capture the essence of an object, scene, or pattern requires advanced techniques.

One of the primary challenges in image feature extraction lies in the vast variability of visual data. Images can contain intricate details, textures, shapes, and structures, making it difficult to discern which visual elements are relevant for a particular task. Moreover, the same object or concept can be depicted in numerous ways across different images, further complicating the extraction of consistent and informative features.

Below is the pixelated form of the sample image in Figure 12-13. If you look closely to the picture below, you can notice the small boxes or pixels, which are serving as building blocks of this image.



Figure 12-0-14 Pixelated Image

However, the way a human sees an image is completely different than how computers understand an image. Machines(computers) struggles to differentiate images based on their features. They store image in form of numbers. Look at the image in Figure 12-15.

```
((576, 3906),
 array([[0.94901961, 0.94901961, 0.94901961, ..., 0.94901961, 0.94901961,
 0.94901961],
 [0.94901961, 0.94901961, 0.94901961, ..., 0.94901961, 0.94901961,
 0.94901961],
 [0.94901961, 0.94901961, 0.94901961, ..., 0.94901961, 0.94901961,
 0.94901961],
 ...,
 [0.94901961, 0.94901961, 0.94901961, ..., 0.94901961, 0.94901961,
 0.94901961],
 [0.94901961, 0.94901961, 0.94901961, ..., 0.94901961, 0.94901961,
 0.94901961],
 [0.94901961, 0.94901961, 0.94901961, ..., 0.94901961, 0.94901961,
 0.94901961]]))
```

Figure 12-0-15 A small part of Sample Image in form of numbers

In digital image representation, images are stored as matrices of numbers, where each number represents the pixel intensity or color value at a specific position in the image. The size of this matrix depends on the resolution of the image, which is determined by the number of pixels along the width and height of the image. For grayscale images, each element typically holds a single intensity value, while for color images, each element can represent color channels (e.g., red, green, blue).

Individual pixels, while forming the basic building blocks of an image, often lack the necessary context and semantic understanding to fully capture the meaning and content of the image. This limitation is one of the key challenges in image analysis and computer vision. Instead of analysing individual pixels, algorithms extract meaningful features that capture important patterns and structures in the image. These features can include edges, textures, corners, and other higher-level attributes. A technique called Image segmentation divides an image into meaningful regions or objects, allowing for analysis at a higher semantic level. Many algorithms consider the context and relationships between neighbouring pixels to improve analysis accuracy. This might involve considering neighbouring pixels in various directions or incorporating spatial constraints.

When processing or analysing images using algorithms, neural networks, or any other techniques, this matrix representation of pixels is crucial. It forms the foundation for operations like filtering, transformation, and feature extraction, allowing algorithms to operate on the raw pixel data to extract meaningful information and make predictions.

*Computers extract pixels from images through a process known as **digitization or sampling**. It involves capturing discrete samples of an image at regular intervals. In the context of digital images, sampling involves selecting specific points from the continuous image to represent the pixel values. Each sample corresponds to a pixel in the digital representation. Once the samples are taken, quantization is applied to map the continuous range of intensity values to a finite set of discrete values. For grayscale images, this is typically done by assigning each sample to a specific intensity level (e.g., using 8 bits to represent values from 0 to 255). For color images, quantization is applied to each color channel (e.g., red, green, blue) to represent color values. The samples obtained through sampling and quantization are organized into a matrix, where each element of the matrix represents a pixel's value.*

Digitization involves converting continuous visual information into discrete numerical values that can be processed and stored digitally. Once the image is digitized and stored as a matrix of pixel values, computers can access and manipulate individual pixels or groups of pixels using programming techniques. This process enables computers to represent images using a matrix of pixel values, which forms the basis for image processing, analysis, and various computer vision tasks.

Traditional methods for image feature extraction often relied on handcrafted techniques designed to capture specific characteristics of the image data. They focused on extracting image details like edges of an image, corner detection or studying pixel distribution of an image, analysing pixel distribution of two images for classification, etc. Each object has a distinct overall shape that contributes to its identification. For instance, a car typically has a rectangular shape with wheels, while a cat has a more organic and elongated shape. The color of an object can provide important cues for recognition. For example, the brown fur of a dog, the metallic body of a car, and the fur and eyes of a cat have distinct colours. The scene and the arrangement of objects within the image can offer additional clues. For instance, seeing a car on a road or a cat near a window can help identify the objects. When distinguishing between objects like a dog, a car, and a cat, features do play an important role.

Let us briefly discuss some of the traditional methods before moving towards modern approach to Computer Vision.

Edge detection methods, such as Sobel and Canny operators, aimed to identify abrupt changes in intensity within an image. Edges typically correspond to object boundaries, which are important for object recognition and scene understanding. These methods work by convolving the image with specific filter kernels to highlight regions with significant intensity gradients. The Sobel operator, for example, calculates gradients in the horizontal and vertical directions, and the gradient magnitude represents the edge strength. High gradients indicate areas where there's a rapid transition from one color or intensity to another. The Canny edge detector is another widely used method that involves multiple steps, including gradient calculation, non-maximum suppression to thin edges, and hysteresis thresholding to identify strong edges. By detecting edges, the machine can identify important boundaries and shapes within the image. This edge information, along with other features like color and texture, can be used as input for machine learning models, including neural networks, to recognize and differentiate objects.

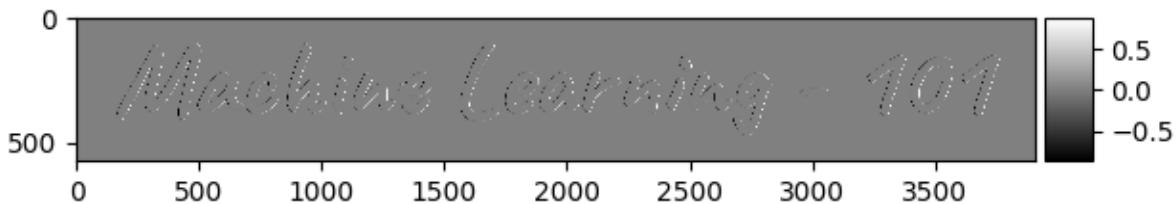


Figure 12-0-16 Edge Detection of Sample Image in greyscale format

Corner detection methods, like the Harris corner detector, identified points in an image where there are significant variations in intensity in multiple directions. Corners are unique and distinctive features often found at object junctions or intersections. The Harris corner detector calculates a corner response function based on the gradient of pixel intensities, and points with high responses are considered corners. These corners play a crucial role in applications like object tracking, image stitching, and camera calibration.

Some other key traditional feature extraction techniques for images are Texture Analysis of an image, study of pixel intensity distribution, template matching, etc. However, these techniques were good in patches but not good enough to perform image classification on a broad set of data.

13.8.1. SIFT

The introduction of the **Scale Invariant Feature Transform (SIFT)** in 1999 marked a significant advancement in the field of computer vision, revolutionizing the representation of images and paving the way for enhanced object recognition capabilities. SIFT, as devised by Lowe in his seminal work, addressed the challenges of image representation and object recognition in a novel manner, as outlined in the given text.

Originally conceptualized for the purpose of object recognition, SIFT addressed a critical aspect beyond mere object tagging – it enabled the precise localization of objects within images. This capability was a fundamental departure from earlier methods, which primarily focused on object identification without specifying its precise spatial location.

The methodology of SIFT involves a comprehensive process that unfolds across multiple stages. The process commences with a meticulous analysis of the image, performed over a pyramid of diverse scales. This approach allows SIFT to capture distinctive features regardless of variations in object size, ensuring that objects are accurately identified across different parts of the image.

A crucial facet of SIFT is the detection of interest points, which are key areas that potentially signify the presence of an object. These interest points serve as anchor points for subsequent analysis and form the basis for feature extraction. The process of feature extraction involves creating descriptors that encapsulate essential attributes of the identified interest points. These descriptors, often referred to as **image descriptors** in the world of computer vision, encode valuable information about the visual characteristics of the object. In the Figure 12-16 below, SIFT feature detector is being used to detect feature key points and descriptors in sample image Figure 12-13 and match the descriptors using **Brute Force** matcher. Following is the sample program using Python OpenCV library to achieve this:

```
import cv2
img1 = cv2.imread('ImageClassificationBasic.jpg')
img2 = cv2.imread('img2.jpg',0)
sift = cv2.SIFT_create()
# detect and compute the keypoints and descriptors with SIFT
keypoints_1, descriptors_1 = sift.detectAndCompute(img1,None)
kp2, des2 = sift.detectAndCompute(img2,None)
# create BFMatcher object
bf = cv2.BFMatcher()
# Match descriptors.
matches = bf.match(descriptors_1,des2)
# sort the matches based on distance
matches = sorted(matches, key=lambda val: val.distance)
# Draw first 50 matches.
out = cv2.drawMatches(img1, keypoints_1, img2, kp2, matches[:50], None, flags=2)
plt.imshow(out), plt.show()
```

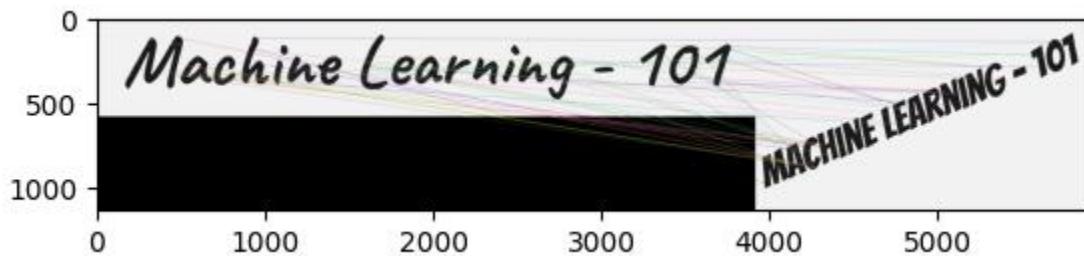


Figure 12-0-17 SIFT feature detector being used for Feature Matching

Perhaps one of the most remarkable achievements of SIFT is its ability to determine the pose of an object. By analysing the accumulated information from interest points and their corresponding descriptors, SIFT facilitates the accurate estimation of an object's orientation and spatial arrangement within the image. This profound capability greatly enhances the utility of SIFT for tasks like robotic navigation, augmented reality, and image matching.

However, SIFT involves complex calculations, such as convolutions, gradient computations, and histogram generation, which can be computationally expensive, especially when applied to large datasets or real-time scenarios. This can hinder its practicality for applications that require high-speed processing, such as video analysis and real-time object detection. Also, it relies on handcrafted features and descriptors, which are manually designed to capture relevant information from images. While SIFT's feature extraction process is effective, it might not fully capture the intricate and high-level patterns in data, limiting its ability to adapt to diverse and complex visual tasks.

13.8.2. CNN

The limitations of SIFT prompted researchers to explore alternative approaches, leading to the rise of neural network-based techniques. Neural networks, particularly **convolutional neural networks (CNNs)**, address many of these limitations by automatically learning hierarchical features from data, adapting to various transformations, and enabling end-to-end training. CNNs automatically learn features from raw pixel data during the training process. Unlike SIFT, where features are manually designed, CNNs have the ability to discover relevant features and patterns directly from the input images. They use multiple layers of convolutional and pooling operations to create a hierarchy of features. Lower layers capture basic features like edges and textures, while higher layers capture more complex and abstract features. Convolutional layers in CNNs apply filters to local regions of the input image, enabling them to capture spatial hierarchies of features that are invariant to translation. This property ensures that CNNs can detect features and objects regardless of their position within the image. This hierarchical representation enables CNNs to learn and represent intricate patterns and structures in the data, making them more effective at recognizing objects under varying conditions.

CNNs allow for end-to-end learning, where the entire model is trained jointly for a specific task. This means that the entire architecture, including feature extraction and decision-making components, is optimized together. This contrasts with SIFT's fixed pipeline, which might require manual adjustments and fine-tuning for different tasks. They can effectively learn from large amounts of data, capturing intricate patterns and nuances that might be difficult for handcrafted methods like SIFT to capture. The ability to process massive datasets contributes to the CNN's robustness and generalization to diverse scenarios. More importantly, CNNs can capture complex spatial relationships between features due to their hierarchical architecture. Features learned in early layers can be combined and recombined in later layers to capture high-level semantic information, enabling CNNs to understand context and relationships between objects and parts.

While early CNN architectures might still be computationally intensive, advancements in hardware and model design have enabled the development of more efficient networks that can perform real-time processing, even on resource-constrained devices. CNNs have demonstrated remarkable performance in a wide range of computer vision tasks, including object detection, image classification, and image generation.

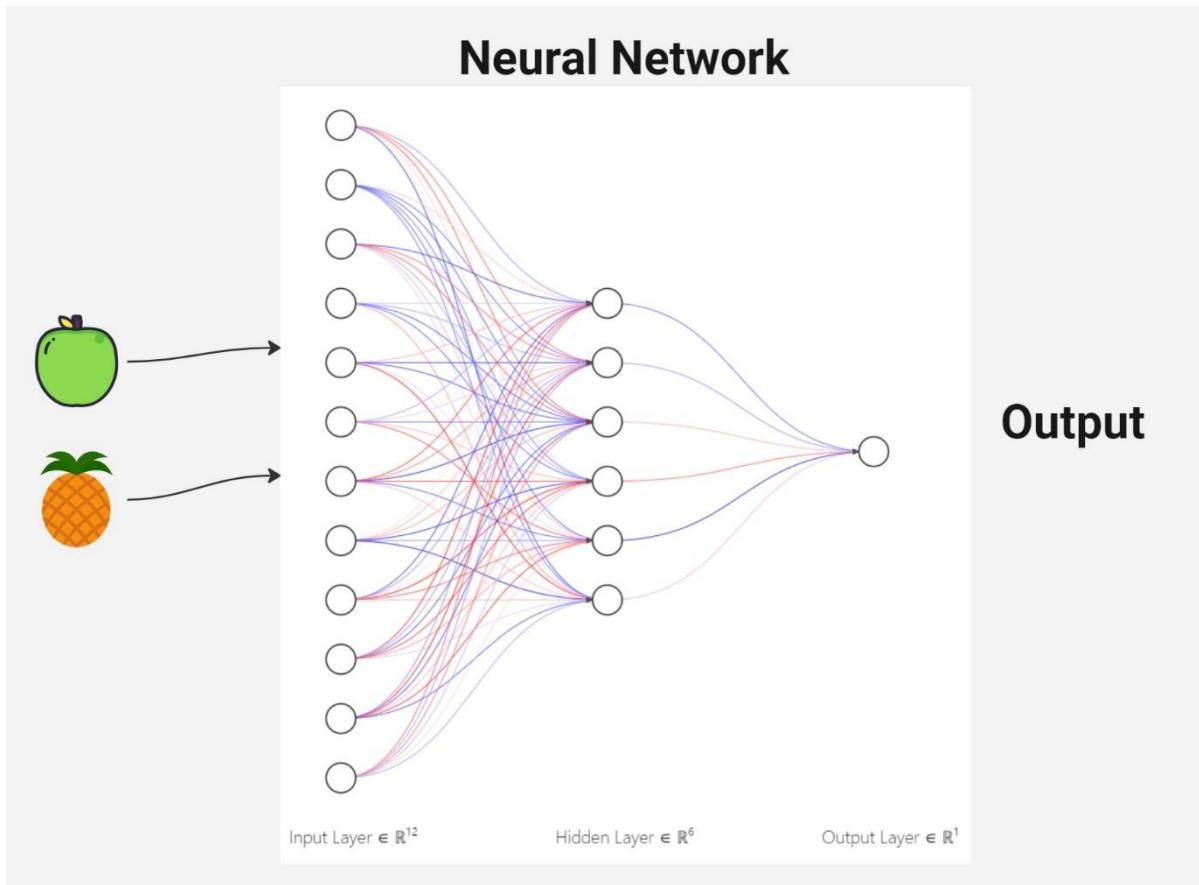


Figure 12-0-18 Neural network architecture diagram

We shall be studying about Neural Networks and CNNs in upcoming chapter. So more on this later.

13.9. Summary

In this chapter, we discussed the process of feature extraction from various types of data, including numerical, categorical, text, and image data. By leveraging distinct methodologies tailored to each data type, we gained insights into how to effectively transform raw data into meaningful features that power our machine learning models.

When dealing with numerical data, we explored techniques such as scaling, normalization, and dimensionality reduction. These methods enabled us to standardize feature ranges, mitigate the impact of varying scales, and enhance computational efficiency. Principal Component Analysis (PCA) emerged as a valuable tool for reducing dimensionality while retaining essential variance, thereby simplifying complex numerical datasets.

Our exploration of categorical data involved techniques like one-hot encoding, label encoding, and feature hashing. These methods enabled us to convert categorical variables into numerical representations suitable for machine learning algorithms. By capturing categorical relationships and preserving information, we ensured our models could effectively leverage this vital type of data.

Feature extraction from text data unveiled the power of techniques like BoW, Parsing, Tokenization, TF-IDF (Term Frequency-Inverse Document Frequency) and word embeddings. These methods enabled us to transform text into numerical vectors, capturing semantic meaning and relationships between words.

While discussing images, we explored the Scale Invariant Feature Transform (SIFT) and its limitations, paving the way for Convolutional Neural Networks (CNNs). CNNs revolutionized image feature extraction by automatically learning hierarchical features and spatial relationships. This approach helps to overcome the challenges of representing complex visual patterns and enabled robust image analysis and recognition.

Even though, numerous open source libraries are present to perform the operations discussed, yet it requires a sound intuition to use correct techniques at the time required. Now it is time to discuss Neural networks in detail.

Objects, Data & AI: Build thought process to develop Enterprise Applications

Deep Learning & Neural Network

“

“Any man could, if he were so inclined, be the sculptor of his own brain.” - Santiago R.y Cajal

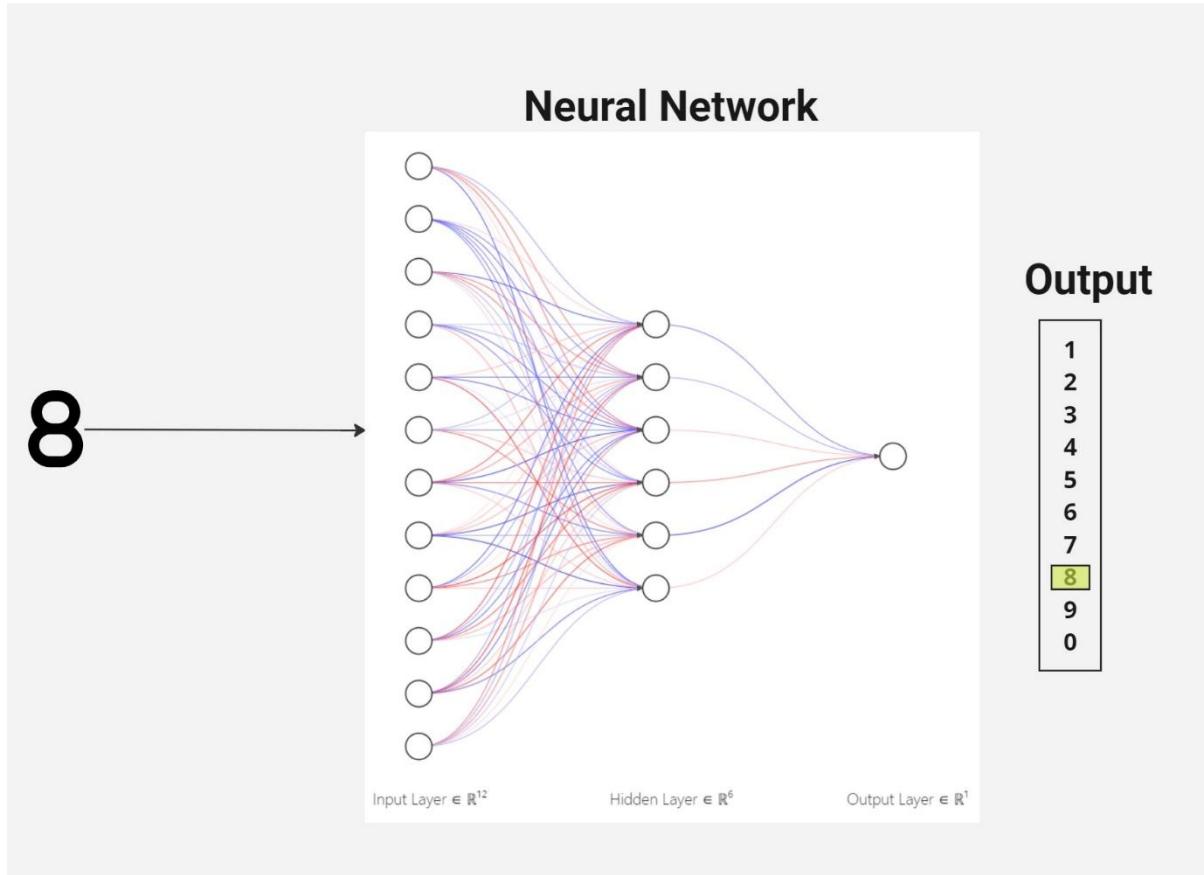


Figure 13-0-1 Neural Network - Image Classification

14.1. Human Brain as Inspiration

If we are presented with two slightly or completely different images of dog, we would be clearly able to classify the image object as dog. Our brain is able to make this distinction by learning the patterns of dogs over the years. Machine learning is based on similar concept and differs from traditional explicit programming (hard coded algorithm) due to its ability to adapt to and learn from similar but not identical situations. ML models learn from data and recognize patterns in order to generalize for data that they have never seen.

Human brain has been the subject of neuroscientists, who have always been interested in understanding the working structure of our brain. Human species is the most dangerous and evolved species on this planet. Our ability to evolve, thought process to navigate through a situation, channelling of emotion, etc. make a case of compelling study.

The study of the human brain and its intricate neural network architecture has played a significant role in inspiring the evolution of neural networks in machine learning. Researchers and scientists have looked to

the brain's structure and information processing mechanisms to develop artificial neural networks that can simulate some aspects of the brain's functionality. The idea of simulating the behavior of neurons and their interactions led to the creation of artificial neurons that mimic the behavior of their biological counterparts.

*The brain is composed of billions of nerve cells called neurons. **Neurons** are interconnected through synapses, which are specialized junctions that allow communication between neurons. These connections form a complex network that enables the brain to process information and perform various functions. Our senses (such as sight, hearing, touch, taste, and smell) receive information from the external environment. This information is transmitted as sensory signals to the brain for further processing. Neurons process and transmit information through a combination of electrical impulses and chemical signals. When a neuron receives a signal, an electrical impulse called an action potential travels along the neuron's axon. At the synapses, chemical neurotransmitters are released to transmit the signal from one neuron to another. The transmission of information between neurons occurs through synapses. Neurotransmitters released from the presynaptic neuron bind to receptors on the postsynaptic neuron, allowing the signal to be transferred from one neuron to another. Neurons integrate signals from multiple sources, making complex decisions based on the combined input. This process of integration enables the brain to perform various cognitive functions, including memory, learning, reasoning, and emotion. The nervous system consists of the central nervous system (CNS), which includes the brain and spinal cord, and the peripheral nervous system (PNS), which connects the CNS to the rest of the body. The PNS includes sensory and motor neurons that relay information between the CNS and different parts of the body. After processing and integrating information, the brain sends motor commands through the nervous system to control various actions in the body. Motor neurons transmit these commands to muscles and glands, enabling movement, response to stimuli, and other bodily functions. The entire process operates as a feedback loop. Sensory input triggers neural processing, which leads to motor output and action. The results of these actions are then sensed by the body, providing feedback that influences future responses.*

14.2. Perceptron

The concept of neural networks draws inspiration from the intricate information processing of human neurons. Each **neuron** in our body functions by receiving input, processing it, and producing an output. This notion gave rise to **Perceptrons**, mathematical models proposed by Frank Rosenblatt in 1958, which aimed to simulate the information processing capabilities of neurons.

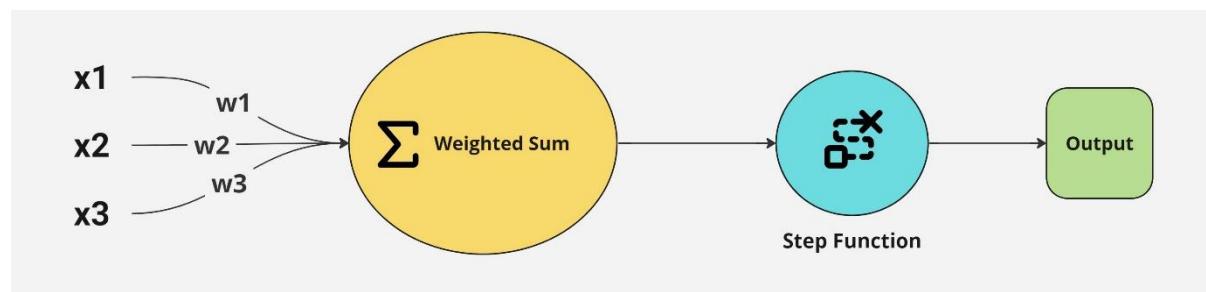


Figure 13-0-2 Perceptron Mathematical Model

A perceptron is a fundamental building block of artificial neural networks and serves as a simple model of a biological neuron. A perceptron receives multiple binary inputs, denoted as x_1, x_2, x_3, \dots , where each can be either 0 or 1 and produces a single binary output. Additionally, there are corresponding weights w_1, w_2, w_3, \dots associated with each input. The weights represent the importance or significance of each input in influencing the perceptron's output. The perceptron computes a weighted sum of the inputs and weights. This is done by multiplying each input by its corresponding weight and summing up these weighted values.

The perceptron then compares the weighted sum to a threshold value. The threshold is a parameter of the perceptron that determines the point at which the perceptron will "fire" and produce an output of 1. If the weighted sum is greater than or equal to the threshold, the perceptron outputs 1; otherwise, its output is 0.

$$f(x) = \begin{cases} 1, & \text{if } w \cdot x + b > 0, \\ 0, & \text{otherwise} \end{cases}$$

$$w \cdot x = \sum_{i=1}^m w_i x_i$$

Here, w denotes a vector of real valued weights, m is the number of inputs to the perceptron, and b is the bias. In addition to the weights and inputs, the perceptron model includes a bias term. The bias serves as an offset or a threshold that adjusts the decision boundary of the perceptron. It allows the perceptron to learn and generalize patterns that may not pass through the origin (0,0) in the input space. When combined with the weighted sum of inputs, the bias influences the point at which the perceptron transitions from producing output 0 to output 1. A positive bias makes it easier for the perceptron to output 1, while a negative bias makes it harder.

Imagine you're trying to decide whether to watch a movie at a cinema. Your decision could be influenced by the following factors:

1. Is it a weekend?
2. Is the movie a genre you enjoy?
3. Is the movie highly rated?
4. Is the cinema close to your home?

We can represent these factors using binary variables x_1 , x_2 , x_3 , and x_4 , where $x_i = 1$ indicates "yes" and $x_i = 0$ indicates "no." Let's create a perceptron model for this decision:

Suppose you have different levels of importance for each factor:

You prioritize watching movies on weekends, so you assign a weight $w_1 = 3$ to the weekend factor.

The genre of the movie matters a lot to you, so you assign $w_2 = 5$ to the genre factor.

The movie's rating is moderately important, so you assign $w_3 = 2$ to the rating factor.

The proximity of the cinema is somewhat important, so you assign $w_4 = 1$ to the distance factor.

You set a threshold of 6 for the perceptron.

With these choices, the perceptron's output will be 1 whenever the weighted sum of the factors (including the bias term) is greater than or equal to the threshold (6).

Resultant equation can be given as:

$$f(x) = \begin{cases} 1, & \text{if } w \cdot x + b > 6, \\ 0, & \text{otherwise} \end{cases}$$

In this example, if it's a weekend ($x_1 = 1$), the genre is enjoyable ($x_2 = 1$), and the movie is either highly rated ($x_3 = 1$) or the cinema is close to your home ($x_4 = 1$), then the perceptron would output 1, indicating

that you should watch the movie at the cinema. Otherwise, the output would be 0, suggesting you might not want to watch the movie.

The foundational principle of a neural network mirrors the interconnectedness and information processing observed in human neurons. Instead of electrical impulses, each neuron in an artificial neural network stores a numerical value representing its strength and capacity to transmit information to neighbouring neurons. The architecture of a neural network is organized into layers. The initial layer receives input data, while the final layer produces output or predictions. Neurons in each layer are often fully connected to neurons in the previous and subsequent layers. Information flows sequentially from the input layer through intermediate layers to the output layer. Notably, the connections between neurons are distinct, with varying strengths. Some connections may significantly impact the final outcome, while others play a more minor role.

14.3. Feed Forward Neural Network

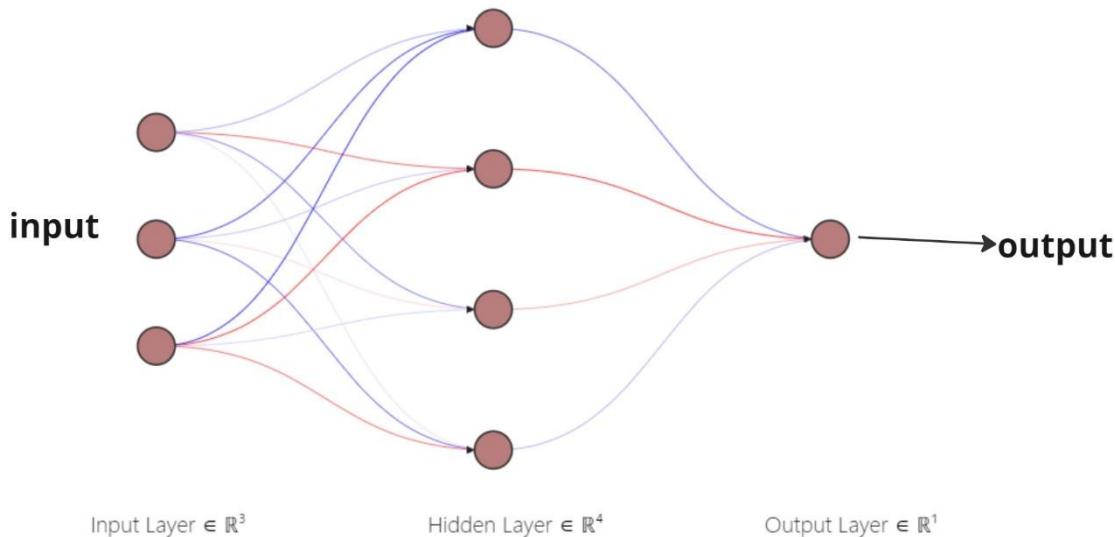


Figure 13-0-3 A multi-layer perceptron based Neural Network

In the example above, the first column of perceptrons, layer, by weighing input evidence, and the results produced by first layer is being passed as an input to second layer and after weighing up evidence at second layer, output is transmitted to final layer for final decision making result as an output. The layers between the input and the output layers are referred to as “hidden layers”. In this example, we just have 1 hidden layer.

The transmission of information occurs layer by layer, with data progressing from the input to the output. The training of neural networks is facilitated by a process known as backpropagation, a fundamental concept that will be explored in more detail in subsequent sections. **Backpropagation** involves adjusting the network's connection weights based on the discrepancy between predicted outputs and actual target values, iteratively refining the network's ability to make accurate predictions.

As a practical use case of perceptron model of neural network, we can think of Figure 13-1, where, the inputs to the network might be the raw pixel data from a scanned, handwritten image of a digit. And we'd

like the network to learn weights and biases so that the output from the network correctly classifies the digit.

The development of the backpropagation algorithm allowed for the training of **multi-layer perceptrons (MLPs)**. This enabled deeper architectures, allowing neural networks to learn complex features from data through successive layers of transformations. In 1980s, the concept of convolutional layers in neural networks, inspired by the visual processing in the human brain's visual cortex, revolutionized image analysis by automatically learning hierarchical features from images, leading to breakthroughs in image recognition tasks.

Since, in the neural network models based off perceptron, one layer transmits output to the other layer but does not receive any feedback, these models are also known as **feed-forward neural network**.

If we want to **train a feed forward neural network**, we can follow the steps as follows:

Step 1: Initialize random weights and biases:

At the beginning of training, the weights and biases of the neural network's individual neurons are set to random values. These weights determine the strength of connections between neurons and play a crucial role in the network's ability to learn and make predictions.

Step 2: Calculate an initial prediction by feed-forward:

Input data is fed into the network through the input layer. The data passes through the hidden layers, where weighted sums are computed and transformed by **activation functions**. These transformations generate an output prediction, which is compared to the desired target output.

Activation Functions: In a neural network, a single neuron's output calculation is a linear equation, however, in real life, data is non-linear. Activation functions inject the necessary nonlinearity into the network's computations, addressing a significant limitation in the original linear equations used for individual neurons.

Activation functions are mathematical functions applied to the output of each neuron. They introduce nonlinearity by introducing a curve or threshold that determines whether the neuron "fires" or becomes activated. This nonlinearity allows neural networks to approximate a wide range of functions, enabling them to model complex and nonlinear relationships within data.

The **sigmoid activation function** is often used to squash the output of a neuron into a bounded range, specifically between 0 and 1. This property makes it well-suited for tasks where the network's output needs to represent probabilities or binary classifications. By confining the output values within a limited range, the sigmoid function ensures that predictions remain within a meaningful and interpretable range.

While the sigmoid activation function does introduce some level of nonlinearity, it is less effective at combating the vanishing gradient. The **vanishing gradient problem** occurs when gradients become very small during backpropagation, leading to negligible updates to the network's parameters. This issue is particularly problematic for deep networks with many layers, as the gradients can diminish exponentially as they are propagated backward. When gradients are small, the network updates its parameters at a sluggish pace, effectively impeding the model's ability to learn from the data. This can result in networks that fail to capture complex patterns and fail to generalize well to new data.

Due to the vanishing gradient problem, using the sigmoid activation function between hidden layers is generally not recommended, especially for deep networks. Networks with many layers are more susceptible

to vanishing gradients, and this can result in poor convergence and learning. As a result, more modern activation functions, such as **ReLU** and its variants (Leaky ReLU, Parametric ReLU), have become more popular choices for hidden layers. These functions address the vanishing gradient issue more effectively and enable faster convergence.

ReLU, or Rectified Linear Unit, is a widely used activation function in neural networks. Its fundamental purpose is to introduce nonlinearity into the network's computations. Despite its simplicity, ReLU's effectiveness in adding nonlinearity makes it a crucial component for enabling neural networks to model and capture complex patterns and relationships within data. ReLU helps mitigate vanishing gradients problem by allowing positive gradients to flow through unaffected.

While ReLU does prevent vanishing gradients, it can contribute to the exploding gradient problem for certain situations. When the network's weights are initialized with large values, ReLU can cause activations to explode, leading to large gradients during backpropagation.

Batch normalization is a technique that normalizes the activations within a layer by adjusting the mean and variance. This normalization can help prevent activations from becoming too large and contribute to more stable training.

When compared to other activation functions, such as sigmoid or tanh, ReLU requires less computational resources. This efficiency is valuable, especially in scenarios involving large datasets and deep networks, where computational demands can be significant.

Step 3: Calculate the network's error based on a differentiable metric:

A loss or error metric is chosen to quantify the difference between the predicted output and the actual target. **Loss functions** used in deep learning are chosen to be differentiable throughout their domains. This differentiability is a vital property that allows us to calculate the derivative of the loss function with respect to the network's parameters. The gradient of the loss function with respect to the parameters indicates how the loss changes as the parameters are adjusted.

The gradients obtained from these derivatives are utilized in **gradient descent**, an optimization technique that systematically adjusts the parameters to minimize the loss function. This process facilitates efficient and organized exploration of the loss landscape, enabling the network to gradually converge to a set of parameter values that result in improved predictions.

A loss function takes two inputs: the model's prediction and the actual ground-truth value. It computes a measure of the discrepancy between the prediction and the true value, quantifying how well the model's output aligns with reality. This measure serves as an indicator of the quality of the model's predictions, guiding the optimization process.

Common loss functions include **mean squared error (MSE)** for regression tasks and **cross-entropy** for classification tasks. The goal is to minimize this error during training.

Step 4: Adjust the values of weights and biases using backpropagation:

Backpropagation is a process that calculates the gradients (derivatives) of the loss function with respect to the network's parameters. These gradients are computed by propagating the error backward through the network, layer by layer. A gradient indicates the direction and magnitude of adjustments needed to minimize the error. The obtained gradients indicate the direction and magnitude of changes needed in the parameters to reduce the loss. During each iteration of the optimization process, the parameters are updated in the direction opposite to the gradient, gradually steering the model towards more accurate predictions.

Mathematically, for a function $f(x_1, x_2, \dots, x_n)$, the gradient is denoted as :

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right)$$

Each component of the gradient vector represents the partial derivative of the function with respect to the corresponding input variable. The gradient provides the direction of the fastest increase in the function's value from a specific point in the input space.

In the context of optimization, such as in gradient descent, the negative gradient (opposite direction of the gradient) is used to move towards the minimum of a function. By iteratively adjusting the parameters in the direction opposite to the gradient, the optimization algorithm aims to find the minimum (or maximum) value of the function.

Optimization algorithms, such as **stochastic gradient descent (SGD)** or its variants, use these gradients to update the weights and biases iteratively, nudging the network parameters toward values that reduce the error.

Step 5: Repeat until the desired accuracy is reached:

Steps 2 to 4 are repeated iteratively for a predetermined number of epochs or until a specific accuracy threshold is achieved. The network continues to refine its weights and biases, gradually improving its performance on the training data. It's essential to monitor the validation performance to avoid overfitting (excessive adaptation to the training data) and achieve a balance between training and generalization. A validation set, separate from the training set, is used to monitor the model's performance during training. This helps prevent overfitting. Training can be stopped early if the model's performance on the validation set starts to degrade, indicating that it's no longer learning useful patterns.

Later on, a specialized type of neural network came into evolution which took into consideration the concept of time and sequence. These were called **recurrent neural networks (RNN)**. Unlike feedforward networks, RNNs allow feedback loops, enabling them to process sequences of data, time-series, and dynamic temporal patterns. Even though, the architecture of RNN is much closer to our brain functioning, they have not had much success and Feed-forward neural networks remained the popular choice of data scientists.

Later in 2000s, LSTM networks improved upon present neural networks by addressing the vanishing gradient problem and better capturing long-range dependencies in sequences. This architecture was inspired by the brain's ability to store and retrieve information over extended periods.

Just as the brain processes information hierarchically, artificial neural networks have evolved to automatically learn hierarchical features from raw data. This capability enables neural networks to capture meaningful patterns and representations, similar to how the brain recognizes complex features in sensory data. Neural networks aim to achieve generalization and abstraction ability like Human brain by learning underlying patterns and relationships from data, enabling them to perform tasks beyond the training data. It enables the neural networks to excel in tasks like image classification, speech recognition, etc.

14.4. Neural Network Use Case

Let us apply the theory we have covered over a sample use case, inspired from Kaggle challenges.

Problem Statement:

The quality of concrete is determined by its compressive strength, which is measured using a conventional crushing test on a concrete cylinder. The strength of the concrete is also a vital aspect in achieving the

Objects, Data & AI: Build thought process to develop Enterprise Applications

requisite longevity. It will take 28 days to test strength, which is a long period. So, what will we do now? We can save a lot of time and effort by using Data Science to estimate how much quantity of which raw material we need for acceptable compressive strength.

Solution:

We have to build a solution that should able to predict the compressive strength of the concrete.

Let us approach this problem via EDA and then build a neural network model, which will try to understand underlying relationship between features and ultimately predict cement's compressive strength.

```
import pandas as pd
df = pd.read_csv('concrete.csv')
df.head()
```

Cement	BlastFurnaceSlag	FlyAsh	Water	Superplasticizer	CoarseAggregate	FineAggregate	Age	CompressiveStrength
540.0	0.0	0.0	162.0	2.5	1040.0	676.0	28	79.99
540.0	0.0	0.0	162.0	2.5	1055.0	676.0	28	61.89
332.5	142.5	0.0	228.0	0.0	932.0	594.0	270	40.27
332.5	142.5	0.0	228.0	0.0	932.0	594.0	365	41.05
198.6	132.4	0.0	192.0	0.0	978.4	825.5	360	44.30

Figure 13-0-4 Cement Dataset - first 5 rows

```
# Segregating feature and target variables
x_org = df.drop('CompressiveStrength',axis=1).values
y_org = df['CompressiveStrength'].values

# Plotting correlation heatmap
corr = df.corr()
sns.heatmap(corr,xticklabels=True,yticklabels=True,annot = True,cmap = 'coolwarm')
plt.title("Correlation Between Variables")

# pair Plot
sns.pairplot(df,palette = "husl",diag_kind = "kde")
```

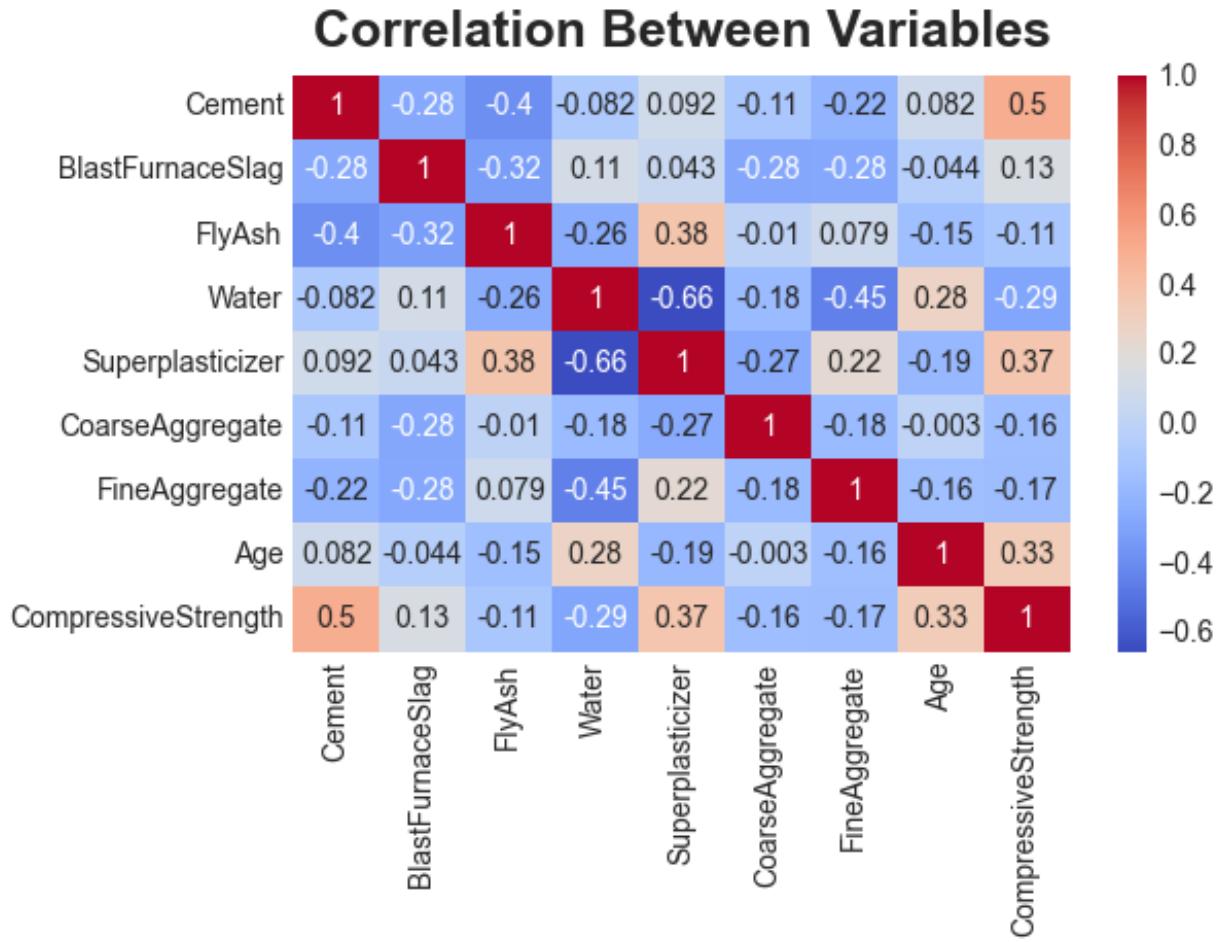


Figure 13-0-5 Correlation Matrix of Feature variables

The pair plot below in Figure 13-6 gives a visual representation of each feature's variance with respect to each other. We build neural networks to understand the underlying relations of these features by self-learning, as it is not comprehensible for humans to learn through them.

Objects, Data & AI: Build thought process to develop Enterprise Applications

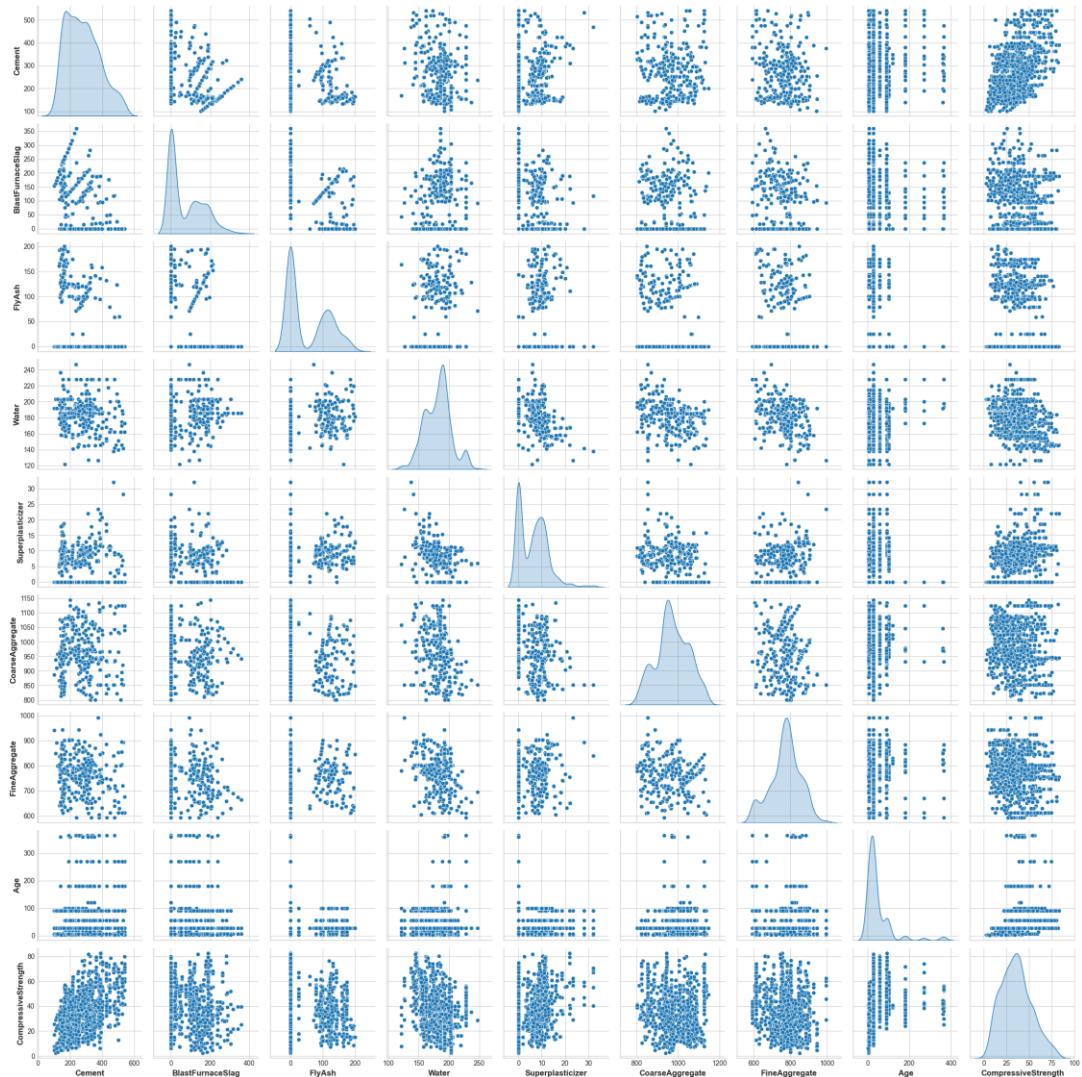


Figure 13-0-6 Pair Plot of features w.r.t correlation

```
from tensorflow import keras
from keras import layers
model = keras.Sequential([
    layers.Dense(units=512, activation='relu', input_shape=[8]),
    layers.Dense(units=512, activation='relu'),
    layers.Dense(units=512, activation='relu'),
    layers.BatchNormalization(),
    # the linear output layer
    layers.Dense(units=1),
])
```

The above code creates a neural network model using the Keras library in TensorFlow. The model consists of several layers of densely connected neurons. The input layer has 8 units, and each subsequent hidden layer has 512 units with the ReLU activation function. After the hidden layers, a Batch Normalization layer is added to normalize the inputs, improving the stability and speed of the training. Finally, there is a linear output layer with 1 unit. It has a linear activation function, which means it outputs a continuous value.

```
from keras import backend
#Defining Root Mean Square Error As our Metric Function
def rmse(y_true, y_pred):
    return backend.sqrt(backend.mean(backend.square(y_pred - y_true), axis=-1))
```

Root Mean Square Error (RMSE) is a commonly used metric for evaluating the performance of predictive models, particularly in regression tasks. It measures the average magnitude of the errors between predicted values and actual (observed) values. RMSE provides a comprehensive measure of how well the model's predictions align with the true outcomes.

```
# Optimize , Compile And Train The Model
opt = keras.optimizers.Adam(lr=0.0015)

model.compile(optimizer=opt, loss='mean_squared_error', metrics=[rmse])
history = model.fit(X_train, y_train, epochs = 35, batch_size=32, validation_split=0.1)

print(model.summary())
```

The code is optimizing, compiling, and training a model using the Adam optimizer with a learning rate of 0.0015. The model is compiled with the mean squared error loss function and the root mean squared error metric. It is then trained on the training data for 35 epochs with a batch size of 32 and a validation split of 0.1. Training history is stored in a variable *history*. Finally, the summary of the model is printed.

Adam Optimizer: Adam (short for Adaptive Moment Estimation) is an optimization algorithm commonly used for training neural networks. It is an extension of the stochastic gradient descent (SGD) algorithm that adapts the learning rate for each parameter based on the estimates of the first and second moments of the gradients. The Adam optimizer helps improve the convergence speed and performance of the model during training. While Adam is a popular optimizer, it might not always be the best choice for every problem. Depending on the specific task, architecture, and dataset, other optimizers like SGD with momentum, RMSProp, or more specialized optimizers such as Nadam or AdamW might perform better.

Epoch: An epoch refers to one complete pass through the entire training dataset during the training process. In the code, the model is trained for 35 epochs, which means it will iterate over the entire training dataset 35 times.

During each epoch, the neural network updates its internal parameters, such as weights and biases, based on the training data. The goal is to minimize the difference between the network's predicted outputs and the actual target outputs for the training examples. This process involves forward propagation (computing predictions), calculating the loss (a measure of prediction error), and backward propagation (updating weights using gradients of the loss with respect to the network's parameters).

Training neural networks over multiple epochs is essential for improving the network's performance. As the network sees the data repeatedly, it refines its internal representations and gradually learns to generalize better to new, unseen data. Early epochs help the network quickly adjust its initial weights, while later epochs allow it to fine-tune its parameters to capture more intricate patterns in the data.

Modern training techniques, such as learning rate schedules and early stopping, help manage the number of epochs effectively. Learning rate schedules adjust the step size of weight updates during training, allowing faster convergence in the beginning and more cautious updates as training progresses. Early stopping monitors the validation error and stops training when the error starts to increase, preventing overfitting.

Learning Rate: It is a hyperparameter in the training process of neural networks and other machine learning models. It controls the step size at which the model's parameters (such as weights and biases) are adjusted during optimization, based on the calculated gradients of the loss function with respect to these parameters. In simpler terms, the learning rate determines how quickly or slowly a model learns from the training data.

```
y_predict = model.predict(X_test)
from sklearn.metrics import r2_score
print(r2_score(y_test,y_predict))
```

0.8735449342594467

The above code calculates the R-squared score for the predictions made by our model. The predicted score displayed suggests that model has a pretty high level of accuracy i.e. 87.35% of the variability in target variable can be explained by the model's prediction. However, interpretation of the score may vary depending upon specific problem and domain.

```
# Plotting Loss And Root Mean Square Error For both Training And Test Sets
plt.plot(history.history['rmse'])
plt.plot(history.history['val_rmse'])
plt.title('Root Mean Squared Error')
plt.ylabel('rmse')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.savefig('4.png')
plt.show()
```

Visualizing the loss during training is a powerful way to gain insights into how well a model is learning and converging. The loss curve provides a visual representation of how the model's performance changes over epochs, helping you make informed decisions about training parameters and potential improvements. The loss curve can guide decisions on when to stop training. If the training loss continues to decrease while the validation loss starts to increase, it could be a sign of overfitting. We can observe if the learning rate is appropriate by analysing the speed and smoothness of loss reduction. Sudden drops or erratic behavior might indicate that the learning rate is too high. If the validation loss plateaus or starts to increase, it might be an indicator that the model has reached its optimal performance. If we are experimenting with different architectures, activation functions, or other hyperparameters, the loss curve helps compare their effects on model training.

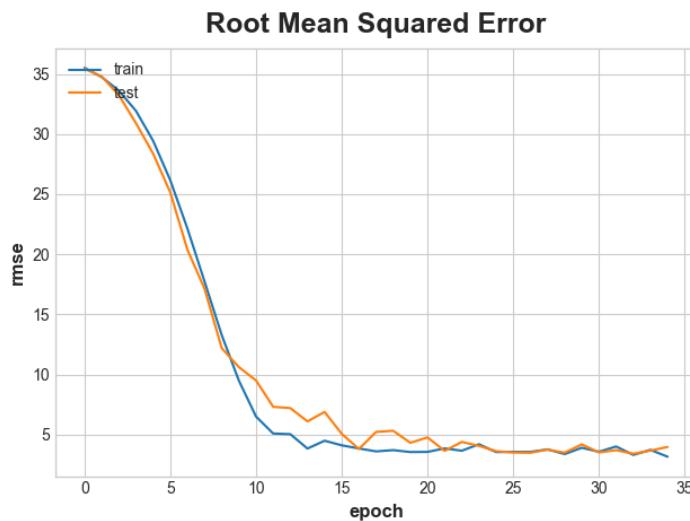


Figure 13-0-7

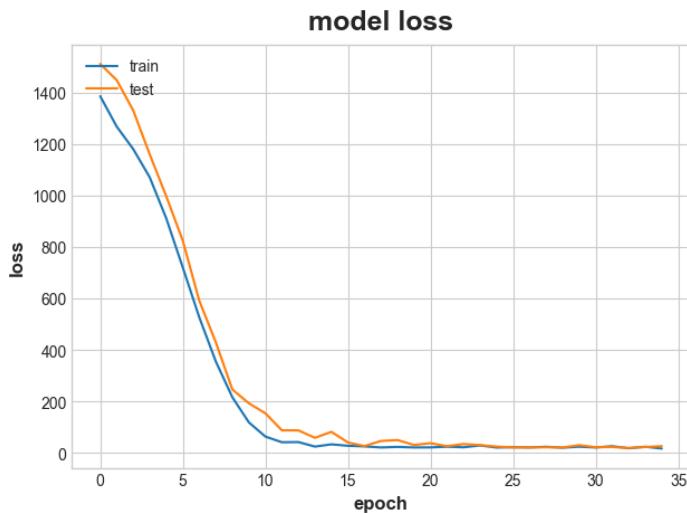


Figure 13-0-8

14.5. Deep Learning

The resurgence of interest in neural networks, often referred to as the "deep learning revolution," was driven by advances in computational power, the availability of large datasets, and improved optimization techniques.

Deep learning is a specialized branch of machine learning that focuses on the automatic extraction and learning of increasingly complex and meaningful representations of data. This approach is achieved via the use of neural networks, which are structured models consisting of successive layers stacked on top of each other. The term "deep" in deep learning refers to the idea of learning multiple layers of representations, with each layer capturing progressively abstract features from the input data. In essence, deep learning aims to enable computers to learn and understand intricate patterns and features from raw data without explicit programming.

Unlike traditional machine learning approaches that may only involve one or two layers of representations (often referred to as shallow learning), deep learning models may involve tens or even hundreds of such layers. Each layer transforms the input data using a combination of learned weights and activation functions, gradually distilling complex information into more compact and meaningful representations.

The process of learning in deep learning is driven by exposure to large amounts of labelled training data. As the model processes the data and computes predictions, it adjusts its internal parameters (weights) to minimize the discrepancy between its predictions and the actual target values. This optimization process, typically achieved through techniques like gradient descent, fine-tunes the model's ability to capture relevant patterns in the data.

14.6. Computer Vision model using CNN

In the last chapter, we discussed about Convolutional Neural Networks in context with image feature extraction. **Convolutional neural networks (CNNs)** are a specialized class of deep learning models designed specifically for image analysis tasks. CNNs have proven highly effective at extracting relevant features from images and learning complex representations that enable them to excel in tasks such as image classification and object recognition.

The **convolutional** operation is a fundamental mathematical operation in CNNs. It involves sliding a small filter (also known as a **kernel**) over the input image and computing the dot product between the filter's weights and the corresponding pixels in the image. This operation allows the network to capture spatial hierarchies and patterns in the input data.

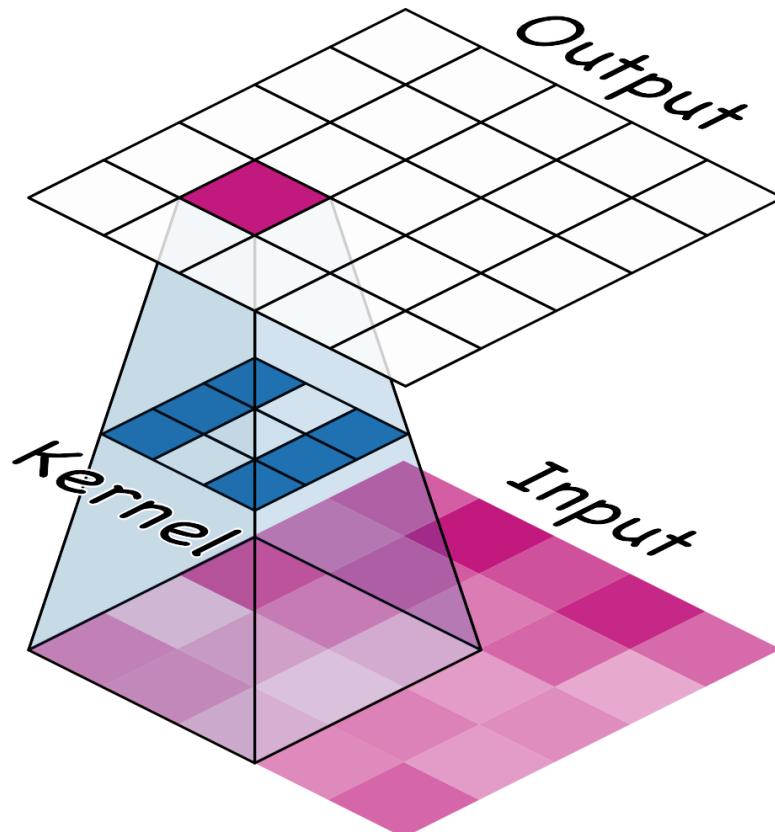


Figure 13-0-9 A representation of kernel acting as lens, source: Kaggle

A typical CNN used for image classification consists of two main parts: the convolutional base and the dense head.

Convolutional Base: The convolutional base is responsible for feature extraction. It comprises multiple layers that perform convolutions, pooling, and other operations to capture local and global features of the input image. These layers progressively transform the raw pixel data into a more abstract representation that encodes relevant patterns.

Dense Head: The dense head is the final part of the CNN, responsible for making predictions or classifications based on the features extracted by the convolutional base. It often consists of densely connected layers that take the high-level features and output the final class probabilities.

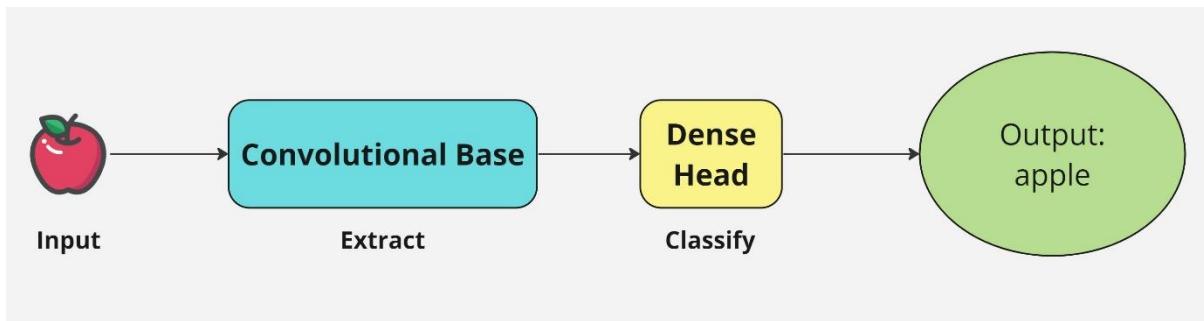


Figure 13-0-10 A Convolutional Classifier

The convolutional base is adept at extracting features such as edges, textures, and shapes from the input image. These features are learned through the training process and are then used by the dense head to make decisions about the image's class. The dense layers in the head combine these learned features and perform the final classification.

After the convolution operation, the ReLU activation function is typically applied element-wise to the output. ReLU replaces all negative values in the feature map with zeros and leaves positive values unchanged. As discussed earlier, it introduces non-linearity to the network, which is crucial for enabling the model to learn complex relationships and representations. It also helps to address the vanishing gradient problem, allowing gradients to flow more effectively during backpropagation and improving training stability.

After the features are detected, pooling operation is performed to reduce the spatial dimensions of the feature maps while retaining their important information. Maximum pooling is a common pooling technique where a small window (pooling kernel) slides over the feature map, and at each position, it selects the maximum value within the window. It also reduces the computational burden by decreasing the number of parameters in the network and controlling overfitting.

Let us now look at a sample implementation of a CNN model for our understanding.

For our example, we take this car image:



Figure 13-0-11 Sample car image

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

plt.rc('figure', autolayout=True)
plt.rc('axes', labelweight='bold', labelsize='large',
      titleweight='bold', titlesize=18, titlepad=10)
plt.rc('image', cmap='magma')
image_path = 'car_1.jpg'
image = tf.io.read_file(image_path)
image = tf.io.decode_jpeg(image, channels=1)
image = tf.image.resize(image, size=[400, 400])
img = tf.squeeze(image).numpy()
plt.figure(figsize=(6, 6))
plt.imshow(img, cmap='gray')
plt.axis('off')
plt.show()
```

The code above reads the image file, resizes it and displays it in grayscale using matplotlib.



Figure 13-0-12 Grayscale car image

```
# Define a kernel with 3 rows and 3 columns.
# Following kernel is used in Edge detection
kernel = tf.constant([
    [-1, -1, -1],
    [-1,  8, -1],
    [-1, -1, -1],
])

# Reformat for batch compatibility.
image = tf.image.convert_image_dtype(image, dtype=tf.float32)
image = tf.expand_dims(image, axis=0)
kernel = tf.reshape(kernel, [*kernel.shape, 1, 1])
kernel = tf.cast(kernel, dtype=tf.float32)
```

The above code defined a 3x3 kernel that is commonly used in Edge detection. It then reformats the image and the kernel to be compatible for batch processing. We can use different kernel to extract different features like sharpened image, etc.

Convolution involves sliding the kernel over the image and performing element-wise multiplication and summation to produce an output known as the feature map. Sum of the numbers in kernel matrix determines how bright the image is. Kernels are designed to highlight specific features or patterns within an image. If the sum of the kernel values is too high, the output image can become excessively bright, while a low sum

can lead to an overly dark image. By adjusting the values within the kernel, we can ensure that the overall effect of the convolution does not dramatically change the brightness or intensity of the image. This process is called **Kernel normalization**.

Batch processing, also known as batch computation or batch execution, refers to the practice of processing multiple data inputs or tasks together as a group, rather than individually or sequentially. In the context of machine learning or deep learning, batch processing is commonly used to train models or perform inference on large datasets.

Instead of processing one data input at a time, batch processing allows for parallelization and efficient computation by processing multiple inputs simultaneously. This can significantly speed up the training or inference process, especially when dealing with large datasets or complex models.

In batch processing, a batch refers to a subset of the entire dataset that is processed together. The size of the batch, known as the batch size, can vary depending on the specific task and available computational resources. Common batch sizes are powers of 2, such as 32, 64, or 128.

Batch processing is widely used in machine learning frameworks like TensorFlow and PyTorch, where data is divided into batches and processed in parallel across multiple processing units, such as GPUs or distributed systems. It helps to optimize memory usage, improve computational efficiency, and achieve better performance in training or inference tasks.

```
conv_fn = tf.nn.conv2d
image_filter = conv_fn(
    input=image,
    filters=kernel,
    strides=1, # or (1, 1)
    padding='SAME',
)

plt.imshow(
    # Reformat for plotting
    tf.squeeze(image_filter)
)
plt.axis('off')
plt.show()
```

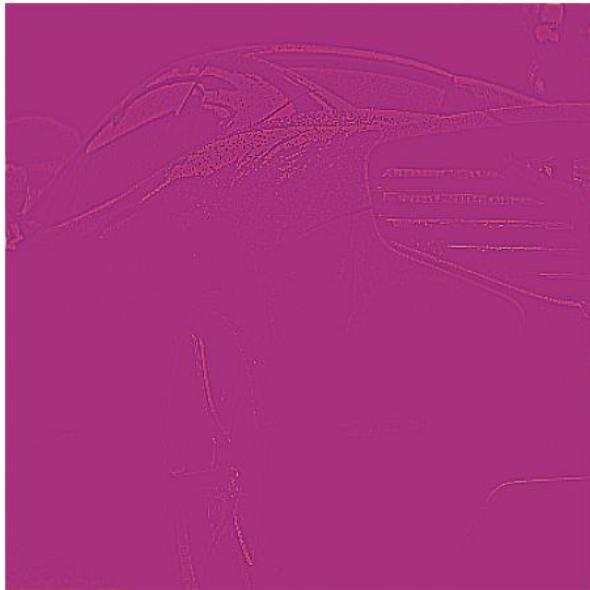


Figure 13-0-13 Edge Detection using kernel

The code above performs a 2D convolutional operation on the input image using the kernel for image detection and visualizes this result, see Figure 13-13.

```
relu_fn = tf.nn.relu
image_detect = relu_fn(image_filter)

plt.imshow(
    # Reformat for plotting
    tf.squeeze(image_detect)
)
plt.axis('off')
plt.show();
```

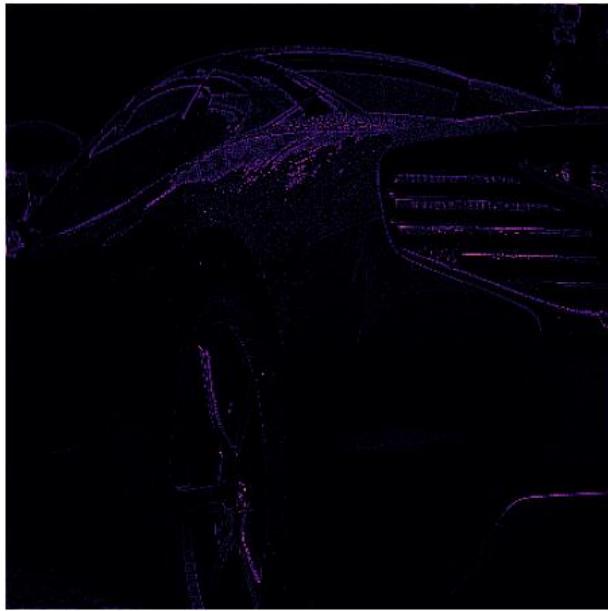


Figure 13-0-14 ReLU feature map detection

So, we have a feature map now. Convolutional head uses this feature set to solve its classification problem. However, after applying the ReLU function (Detect) the feature map ends up with a lot of dead space that is, large areas containing only 0's (the black areas in the image), see Figure 13-14. Having to carry these 0 activations through the entire network would increase the size of the model without adding much useful information. Instead, we would like to condense the feature map to retain only the most useful part -- the feature itself.

To address the issue of dead space and reduce the dimensionality of the feature maps, **max pooling** is used. The operation involves dividing the feature map into non-overlapping patches (also known as pooling regions or pooling windows) and replacing the values within each patch with the maximum value. It selects the maximum value within a small window (pooling kernel) and discards the rest. By doing so, max pooling retains the most important and salient features while discarding less relevant information. However, in some cases, it might result in information loss. The trade-off between down sampling and information loss is a crucial consideration when designing the architecture of a CNN.

```
image_condense = tf.nn.pool(  
    input=image_detect, # image in the Detect step above  
    window_shape=(2, 2),  
    pooling_type='MAX',  
    # we'll see what these do in the next lesson!  
    strides=(2, 2),  
    padding='SAME',  
)  
  
plt.figure(figsize=(6, 6))  
plt.imshow(tf.squeeze(image_condense))  
plt.axis('off')  
plt.show()
```

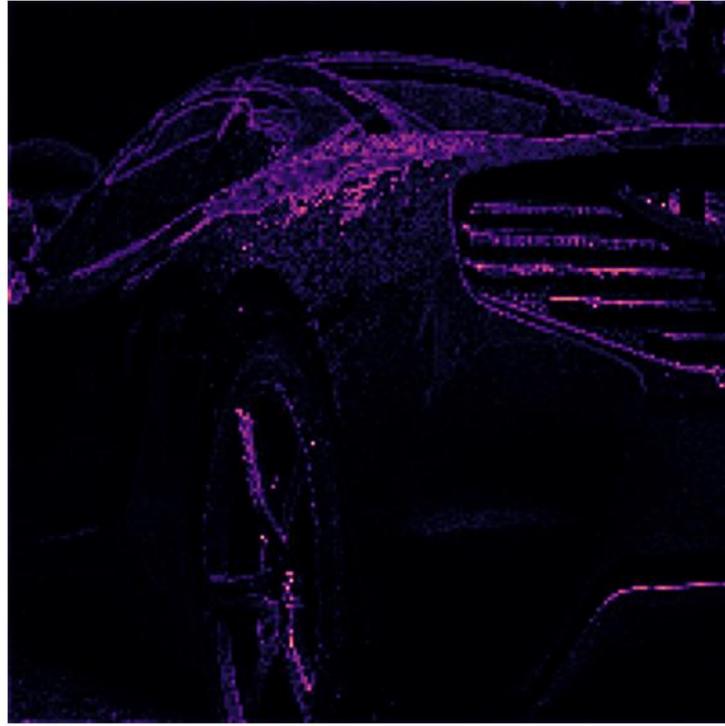


Figure 13-0-15 Intensified feature set after Max Pooling

By pooling around the most active pixels, our CNN model focuses on the most discriminative information, enhances the model's ability to recognize patterns at different spatial positions, as the maximum activation within a pooling region indicates the presence of a relevant feature, regardless of its exact location and making it more likely to capture features that are relevant for classification or other tasks. This requirement is called **Translation invariance**. It is a desirable property in image analysis tasks because it allows the network to focus on the essence of a feature rather than its precise location. We can observe in Figure 13-15 that the regions containing strong activations become more enhanced, while less important areas are downplayed or suppressed.

When pooling is performed, the network is effectively encoding the presence of a feature into a single activation value, making it more robust to small shifts or translations of that feature within the input data. As a result, the network becomes less sensitive to minor variations in the position of features and is better equipped to generalize and recognize patterns across different spatial locations.

Although, it's important to strike a balance between down sampling and preserving information. Aggressive pooling with large strides or pooling sizes could lead to loss of fine-grained details. The choice of pooling hyperparameters should align with the specific task and architecture to ensure that the network retains the necessary level of detail for effective feature extraction.

In this section, we observed how CNNs can be built to do feature extraction over an image. On a given dataset, CNN models can perform classification task with proper training. CNNs will try to extract different feature sets from the training data over the course of defined epochs and use an optimizer such as Adam to improve the convergence speed and performance of the model during training. Process will be similar to what we did in Feedforward model's [example](#), with change in configuration units in input and dense layers,

however the final output function will not be a linear function but a sigmoid function, as it is not producing continuous output. CNNs have become the cornerstone of modern computer vision, powering applications like image classification, object detection, facial recognition, and more.

Kaggle website has good amount of resources where you can play with such examples and tutorials. Make sure to check them out. Kaggle tutorials can be a good place to start your hands-on journey with Machine learning and Neural networks. They have a good community support and challenging competitions on various problem statements, which can help in honing your skill as a data scientist.

14.7. Summary

In this chapter, we discuss about neural networks, their architecture, mathematical interpretation, and discussed about image feature extraction using a CNN. In essence, neural networks provide the foundation for deep learning, which, in turn, has led to remarkable advancements in computer vision through the specialized architecture of CNNs. These networks have proven to be highly effective at extracting features, recognizing patterns, and understanding visual data, making them a pivotal technology in the field of artificial intelligence, and opening the door to numerous real-world applications. Libraries like Keras are designed to run on top of many low-level deep learning packages such as TensorFlow, Cognitive toolkit, etc. Python and R remain the first choice of programming languages for designing ML models and neural networks.

In the next chapter, we shall be discussing the latest revolution in the field of artificial intelligence: Generative-AI. And thus, we shall be heading towards the final convergence of the agenda of this book. If you have covered this book so far, do not miss out the last few pages, as we discuss how Enterprise applications can now directly leverage artificial intelligence in some use cases and discuss its game changing impact.

Final Convergence: Objects, Data and AI

Rise of Gen-AI

“

“A year spent in artificial intelligence is enough to make one believe in God.” - Alan Perlis

While in the period 2012-2020, artificial neural networks were getting a lot of traction for their ability to understand images, audio, texts, number, etc. and find underlying relation in the feature of input data to produce an output for tasks like image classification, prediction, etc., computer scientists were working to overcome the limitations of present neural networks, i.e. ability to generate new content.

15.1. Evolution of Deep Learning: from predicting to generating data

In 2013, Diederik P. Kingma and Max Welling published the ground-breaking paper titled “*Auto-Encoding Variational Bayes*”, which introduced the concept of **Variational Autoencoders (VAEs)**. VAEs are a type of generative model designed to learn compact representations of data by mapping it to a lower-dimensional latent space using an encoder, and then reconstructing the data using a decoder. VAEs focused on probabilistic interpretation to the latent space, which was a new concept that time.

An encoder neural network takes in the input data (e.g., images, text, audio) and maps it to a lower-dimensional space known as the **latent space**. The latent space is a probabilistic representation, meaning that instead of directly producing a single point in the space, the encoder outputs a probability distribution over possible point. The encoder outputs two vectors for each input: a mean vector (μ) and a standard deviation vector (σ) that define the parameters of a Gaussian distribution in the latent space. However, directly sampling from this distribution and then backpropagating through it during training can be problematic, as it involves non-differentiable operations (sampling) that prevent the gradient signal from flowing backward effectively.

To address this issue, the **reparameterization trick** is used. Instead of directly sampling from the Gaussian distribution defined by μ and σ , a separate noise vector (ϵ) is sampled from a standard Gaussian distribution (mean = 0 and standard deviation = 1). This noise vector is then scaled by σ and added to μ , resulting in a sampled latent vector that preserves the probabilistic nature of the distribution while allowing for gradient-based optimization:

Sampled Latent Vector, $z = \mu + \sigma * \epsilon$

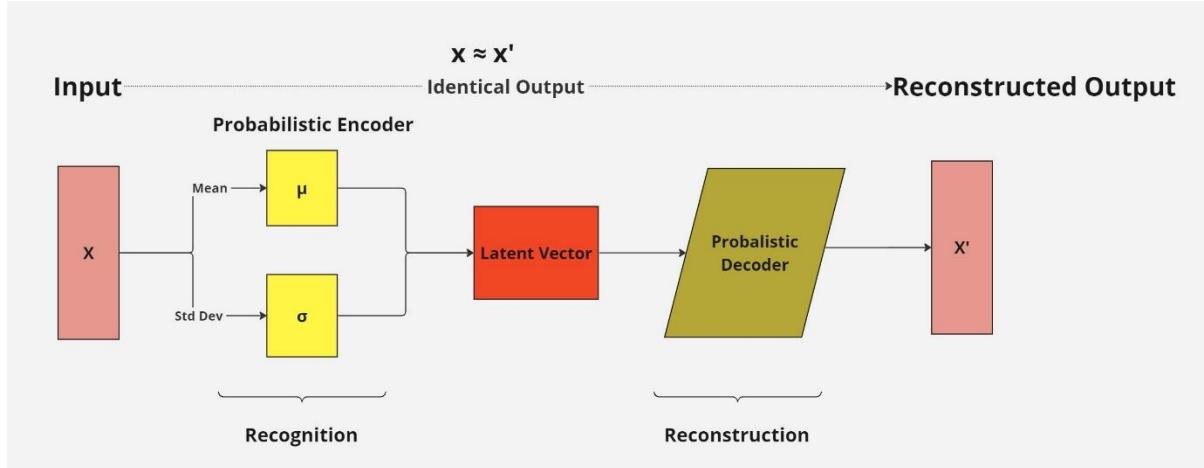


Figure 14-0-1 VAE representation

The reparameterization trick enables the VAE to compute gradients with respect to the encoder's parameters (μ and σ) and the decoder's parameters (weights of the decoder neural network). This is crucial for updating the model's weights through gradient descent or any other optimization algorithm. The gradients indicate how changes in these parameters affect the reconstruction quality and the regularization terms, guiding the optimization process.

By forcing the encoder to output probability distributions instead of deterministic values, VAEs encourage the latent space to have certain desirable properties, such as continuity and smoothness. This regularity in the latent space can lead to better generalization and interpolation capabilities. The decoder, often implemented as a neural network, takes the sampled latent vector, and generates a reconstruction of the original data by mapping these samples back to the data space, while introducing variations due to the probabilistic nature of the model. The decoder's objective is to generate data that resembles the input as closely as possible.

*Suppose we want to train a VAE to create new pictures of animals similar to original ones. To do this, the VAE first takes in a picture of an animal, say a cat or a dog and compresses it down into a smaller set of numbers into the latent space, which represent the most important features of the picture. These numbers are called **latent variables**. Now, the VAE takes these latent variables and uses them to reconstruct a new picture that looks like it could be a real cat or dog picture. The reconstructed picture may differ slightly from the original pictures, but it should still look like it belongs in the same group of pictures of animals. It gets better at creating realistic pictures over time by comparing generated pictures to the original pictures and adjusting its latent variables via backpropagation to make the generated pictures look more like the original ones.*

Short after the release of VAE paper, in 2014, Ian Goodfellow and his colleagues introduced the concept of **Generative Adversarial Networks (GANs)** through the paper “*Generative Adversarial Nets*”. GANs operate on a fundamentally different principle than VAEs, involving two competing neural networks, a generator and a discriminator – which work against each other in a zero-sum game. The generator aims to create data samples that resemble real data, while the discriminator tries to distinguish between real and fake data. During training, the generator and discriminator are updated alternately. The generator aims to improve its ability to generate realistic data by creating samples that deceive the discriminator, while the discriminator aims to enhance its ability to distinguish between real and fake data. This iterative training

process leads to the point of **Nash equilibrium**, where the generator produces data that is indistinguishable from real data, and the discriminator can no longer differentiate between real and fake samples. This equilibrium represents the point at which the generator has learned to generate high-quality data. Early GAN training faced challenges such as mode collapse, where the generator only produces a limited variety of samples. However, subsequent improvements in network architectures, loss functions, and training strategies helped mitigate these issues.

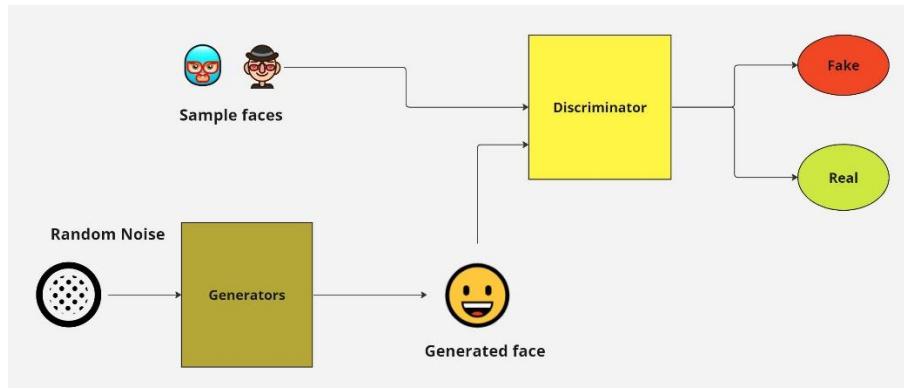


Figure 14-0-2 GNN model

15.2. Encoder-Decoder

Alongside development of VAE and GAN to generate new data from sample data, in parallel, data scientists were also evolving the existing NLP models for machine translation. They used a **sequence to sequence (Seq2Seq)** model, based on **recurrent neural network (RNN)**, which were designed to perform tasks involving transformation of one sequence to another.

RNNs were special kind of neural networks, designed to work with sequential data, where the order of the element matters. RNNs are particularly well-suited for tasks involving sequences such as time series analysis, natural language processing, speech recognition, and more. RNN has the ability to maintain an internal state or memory that captures information from previous time steps in the sequence. This memory is then used to influence the processing of current and future inputs. This concept of memory and recurrence allows RNNs to capture temporal dependencies and patterns in sequences, making them powerful tools for modelling sequential data.

Seq2Seq models were based on Encoder-Decoder architecture. The encoder takes in the input sequence and processes it step by step, producing a fixed-size context vector or hidden state that captures the information from the entire input sequence. RNNs, such as LSTM (Long Short-Term Memory) or GRU (Gated Recurrent Unit), are often used as the building blocks of the encoder. The last hidden state of the encoder serves as the initial hidden state for the decoder.

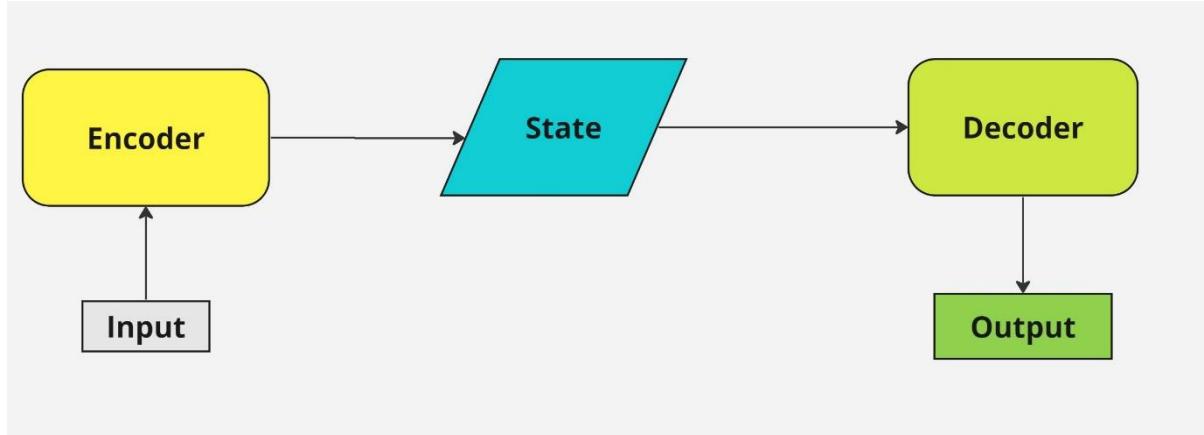


Figure 14-0-3 Encoder Decoder Architecture

The decoder generates the output sequence step by step based on the context vector and the previous generated tokens. Similar to the encoder, RNNs are used in the decoder. At each time step, the decoder generates a token and updates its hidden state using both the context vector and the previously generated token. This process continues until an end-of-sequence token is generated or a maximum sequence length is reached.

Say, an input sentence is encoded into a single context vector that is used as the initial hidden state for the decoder. These hidden states capture the context and information of the input sequence as a whole. However, as the input sentence becomes longer, compressing all the information into a single vector can lead to a loss of detailed information. Long sentences may have complex structures and distant associations that are challenging to capture effectively within this single context vector. Natural language sentences can have complex structures and dependencies that span long distances. These dependencies are crucial for proper translation or generation. Encoding such complex structures into a single vector can lead to inefficiencies and a lack of expressive power to capture all the nuances of the source sentence. Long sequences pose challenges due to the vanishing gradient problem. RNNs struggle to capture long-range dependencies, which is important for tasks like machine translation.

During the decoding phase, when generating the output sequence, the decoder can leverage these hidden states from the encoder. Each time step in the decoder's generation process can be influenced by different hidden states from the encoder. This allows the decoder to access and incorporate the relevant context information from the input sequence as needed.

To Address the limitations of Seq2Seq models, attention mechanisms were introduced.

An **attention mechanism** is a computational method inspired by the human visual attention system, which allows neural networks and models to focus on specific parts of input data while processing information. The core idea of an attention mechanism involves associating different weights or attention scores with various elements of the input. These scores determine the relative importance of different elements during the computation process. It helps models to focus on relevant information, enhancing their ability to capture important relationships and patterns. In tasks involving long sequences, attention mechanisms mitigate the vanishing gradient problem, allowing the model to capture dependencies across distant elements.

RNNs although powerful for their time, were limited by the amount of compute and memory needed to perform well at generative tasks. With significant increase in input data, RNNs required significant scaling of resources the model required. We must understand, for text predicting tasks in natural languages like

English, a model needs to have an understanding of the language, as in many languages, one word can have multiple meanings. Hence, models need to understand the context of the input before generating reasonable text, and to achieve this required significant training data, and in case of RNNs, this became quite a limitation.

15.3. Transformers: A step towards Language Models

In 2017, a revolutionary paper “*Attention Is All You Need*” was published by Google researchers, which proposed Transformers architecture and it revolutionized the way neural networks process sequential data, addressing some of the limitations of RNNs and LSTMs. Transformers can be scaled efficiently to use multi-core GPUs and it can parallelly process input data thus making use of a much larger training datasets. Most importantly, Transformers are able to learn to pay attention to the meaning of the words it's processing, as the title itself suggests: “*Attention Is All You Need*”.

The idea behind the Transformer is to handle the dependencies between input and output with attention and recurrence completely. The Transformer architecture relies heavily on a mechanism known as **self-attention**. The power of this architecture lies in its ability to learn the relevance and context of all of the words in a sentence. It allows the model to weigh the importance of different words in a sentence based on their relationships to other words within the same sentence. This is particularly powerful for capturing long-range dependencies and complex linguistic structures. Self-attention enables the model to consider all positions in a sequence simultaneously, enabling parallel processing, unlike traditional RNNs and LSTMs, which process sequential data sequentially (one-time step at a time).

15.3.1.1. Transformers Architecture

Look at the Transformer architecture diagram in Figure 14-4, taken from the original paper. A Transformer architecture can be divided in two main components: Encoder and Decoder. Each layer consists of two sub-layers: a multi-head self-attention mechanism and a feed-forward neural network. As the computer understand numbers not text, the first obvious step is to Tokenize the input, say the input has sentences then words in sentences are converted into numbers. Different tokenization techniques can be used here. However, it is very important to understand that the method of tokenization used while parsing the input, must only be used while generating the text

Since, the input is now ready to be fed to the model (machine/computer), the tokenized input is passed to the embedding layer. This layer is a trainable vector embedding space and represents each token as a high-dimensional vector within an embedding space. These vectors capture the semantic meaning and context of the tokens. The embedding vectors are learned during training, and the model adjusts them to optimize its performance on the task.

For e.g., if input sentence “I Love You” is to be translated in French, following would be its probable representation in Embedding space:

“I [TokenId - 001]” is represented by vector [0.3, 0.2, -0.2,..].

“Love [TokenId - 002]” is represented by vector [0.1, -0.8, 0.4,..].

“You [TokenId - 003]” is represented by vector [0.9, 0.2, -0.6,..].

Each token in the sentence gets its own embedding vector, creating a sequence of vectors. **Embedding** is technique to transform data into continuous vector representations in a multi-dimensional space. Words in a text corpus are transformed into dense vectors, where semantically similar words are closer to each other in the vector space. This transformation enables algorithms to work more effectively with the data by

capturing meaningful relationships, patterns, or similarities between the items being embedded, thus being useful for a wide range of tasks like language understanding, sentiment analysis, and machine translation. The goal of embedding is to preserve the semantic meaning of the object in the vector representation. Embeddings can be stored using a Vector DB, but more on this later.

Since the Transformer processes data in parallel, it needs a way to incorporate information about the order of words in a sequence. Each word in the sequence interacts with all other words through attention weights, regardless of their position. This parallelism is a key factor in ensuring the Transformer's efficiency and effectiveness. Although, this also means that without explicit positional information, the model wouldn't be able to differentiate between words based on their order.

To overcome this limitation, **Positional encodings** are added to the input embeddings to provide the model with positional information in **Self Attention Layer**. These encodings allow the model to differentiate between words based on their positions within the sequence, as they are designed to carry information about the relative positions of the words in the sequence. The positional encoding vector is typically a sine or cosine function of the position of the word. The self-attention weights that are learned during training and stored in these layers reflect the importance of each word in that input sequence to all other words in the sequence.

However, the Transformer architecture makes use of multiple sets of self-attention mechanisms in parallel, referred to as **heads**. This **multi-head attention mechanism** enables the model to capture different types of relationships and dependencies between words. The outputs of these heads are concatenated and linearly transformed to produce the final self-attention output. The count of attention heads in the attention layer varies from model to model, but numbers in the range of 12-100 are common. The idea here is that each self-attention head will learn a different aspect of language. Say, one set might learn the relationship of entities in the sentence while other set focuses upon the activity. In our example, the model learns to recognize that “*I*” is the subject, and “*You*” is the object of the sentence, influencing the overall translation. This representation is used to guide and influence the decoding process.

Now that the Transformer architecture has looked closely at different parts of the input data and figured out their importance and applied attention weights, the model would like to make use of this understanding to predict what comes next. It takes in all the information gathered so far and processes through a fully connected feed-forward network, which is good at finding patterns, as in the pieces of puzzle start coming together. The Encoder's feed-forward network consists of nonlinear activation functions, typically followed by linear transformations. This introduces complex transformations to the representations, allowing the model to capture intricate relationships and patterns in the data. One of the motivations behind the feed-forward network is to reduce the dimensionality of the representations while retaining relevant information. This is achieved by using intermediate layers with smaller dimensions, followed by an expansion back to the original dimensionality. The feed-forward network operates on each token's representation independently, which means it doesn't inherently capture positional information. This is why positional encodings are added to the embeddings before processing through the network.

The contextual representation from the encoder is used to initialize the decoder's hidden state or initial context. This step ensures that the decoder has access to the encoded information when generating the output sequence. It serves as a foundational step for Decoder's translation process. Before generating the actual output sequence, a special token known as the **start of sequence** token is added to the input of the decoder. This token serves as the initial trigger for the decoder to begin generating the output sequence. The decoder component is similar to the encoder but also has a cross-attention mechanism.

The decoder then works in a loop, iteratively predicting each token in the output sequence. During each iteration, the decoder's self-attention mechanisms and other components focus on the previously generated tokens and the contextual representation from the encoder. This allows the decoder to understand the context and generate tokens that are coherent and contextually appropriate.

The Decoder's feed forward network outputs a bunch of numbers called **logits**. These numbers represent how likely each word(token) in the vocabulary is to come next. Let's imagine, if each word(token) had a score showing how likely it is to be the next word in the sentence. This is where the puzzle gets really interesting. Logits are passed to the **Softmax** layer, which makes sure that all the scores add up to 1, like percentages. It's like turning those scores into a probability game which asks, “*Which word is most probable to come next?*”.

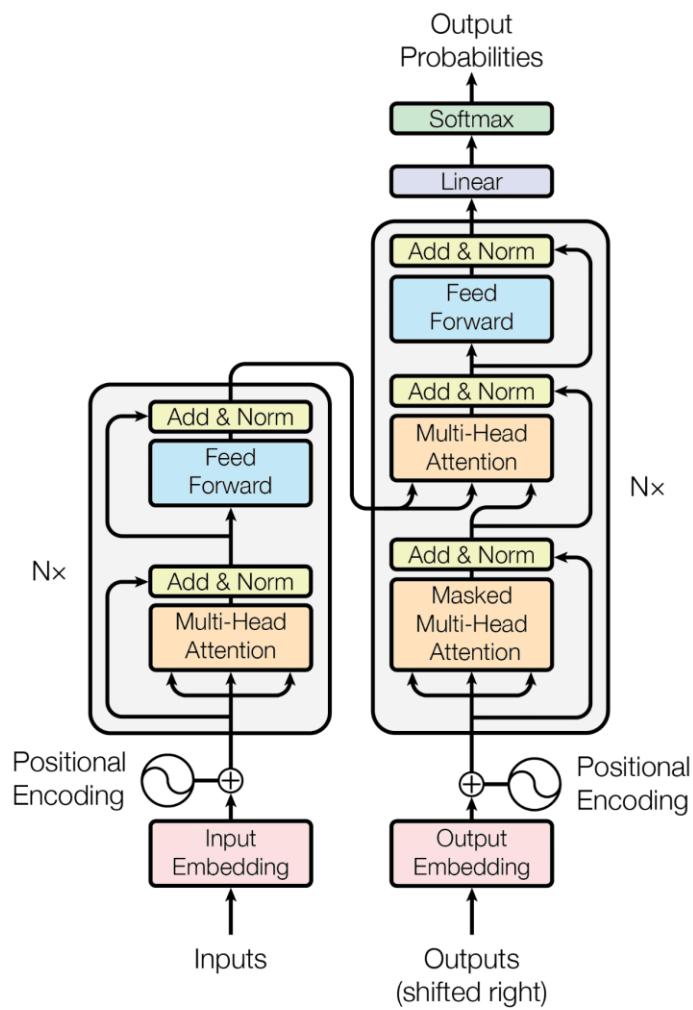


Figure 14-0-4 The Transformer – Model Architecture (Source: <https://arxiv.org/abs/1706.03762>)

The Softmax layer outputs probability score for each word(token). This tells us the likelihood of each word being the next word in the sentence. Among all these probabilities, one word(token) will have the highest probability (greedy decoding). This word is judged as the best guess for the next word in the sentence, based on everything the Decoder has learned from the input. This approach can work well for short generation tasks but is susceptible to repeated words or repeated sequences of words. To generate output

that seems more natural or creative and avoids repetition, **Random Sampling** is used to introduce the variability. However, it may be possible that output may become too creative or wander off the topic.

Say, in the first iteration, the model predicts the French word (token) “Je” (meaning “I” in English) with the highest probability. The predicted token “Je” is then passed back into the decoder loop. This token, along with the contextual representation from the encoder, guides the decoder to predict the next token. In the subsequent iteration, the decoder predicts the next word (token) “T'aime” (meaning “Love You” in English) based on the previously predicted “Je” and the contextual information. The decoder considers both the input context and the tokens generated so far to make informed predictions.

*The decoder continues this process, predicting tokens iteratively until it predicts an **end-of-sequence** token, which indicated that the translation is complete. The sequence of predicted tokens, which includes “Je” and “T'aime”, is detokenized to convert it into human-readable text. In this case, the translation output would be “Je T'aime”, which means “I Love You” in French.*

The original Transformer model had only 6 encoder and decoder layers. However, more recent models have had hundreds or even thousands of layers. This allows the models to learn more complex patterns in the data, but it also makes them more difficult to train. In fact, Transformers are the foundation behind Language models and present-day Large Language Models. The evolution of transformer architectures has led to significant improvements in the performance of language models. Today, transformers are the state-of-the-art approach for a wide range of NLP tasks, including machine translation, text summarization, question answering, and natural language inference. Understanding the context of words and generating coherent responses, makes the Transformers well-suited for the tasks mentioned.

A year later in 2018, Google introduced **BERT**, based on the transformer architecture, which brought a breakthrough in NLP by introducing bidirectional pretraining. Prior to BERT, most language models were trained to process text in one direction only, either left-to-right or right-to-left. This made it difficult for them to capture the full context of a sentence, as they could only see the words that came before or after the current word.

BERT considers both directions, capturing richer contextual information. This significantly improved the quality of contextual embeddings, benefiting tasks like language understanding, sentiment analysis, and more. For example, on the GLUE benchmark, which is a suite of natural language processing tasks, BERT achieved state-of-the-art results on 11 of the 12 tasks. This includes tasks such as question answering, natural language inference, and sentiment analysis.

BERT's bidirectional pretraining enables the model to learn the contextual relationships between words that appear in different parts of a sentence. This is essential for disambiguating words with multiple meanings and for understanding how words influence each other's interpretation within a given context. For example, the word "bank" can refer to a financial institution or the side of a river, and BERT can learn which meaning is more likely based on the surrounding words.

One of the limitations of traditional models that process text sequentially, such as RNNs, is their inability to capture long-range dependencies effectively. While RNNs can be used for generative AI tasks, they struggle with compute and memory, making it hard to keep context in longer texts. The transformers architecture is more parallelizable, and its dynamic attention mechanism helps to capture long-range dependencies in the input. BERT overcomes this limitation by considering words that are not adjacent to each other in the sentence. For instance, BERT can learn that words like "love" and "hate" are often associated, even if they appear in separate sentences. This ability to capture relationships between distant words is crucial for understanding the overall sentiment, tone, and meaning of a piece of text.

Bidirectional pretraining allows BERT to learn more complex representations of words. For example, BERT can learn that the word "bank" can have multiple meanings, depending on its context. BERT's contextual embeddings contribute to its effectiveness across a wide range of NLP tasks, making it a foundational model for modern natural language processing and Generative AI.

Transformers, with their attention mechanisms and parallel processing capabilities, opened the door to improved performance on a wide array of NLP tasks that had previously posed challenges for traditional RNN-based models. The key breakthrough lay in their remarkable ability to capture contextual relationships among words in a sentence. Unlike older models that processed text sequentially, Transformers could simultaneously weigh the significance of each word in relation to every other word in the input sequence. This holistic contextual understanding paved the way for more nuanced comprehension and generation of language.

In the context of Generative AI, the Transformers became a catalyst for the development of LLMs. Large Language models, based upon the Transformer architecture, put the generative capabilities in top gear. LLMs like **GPT (Generative Pre-trained Transformer)** have clearly demonstrated a proficiency in generating coherent and contextually relevant text across a multitude of topics.

15.4. Generative AI

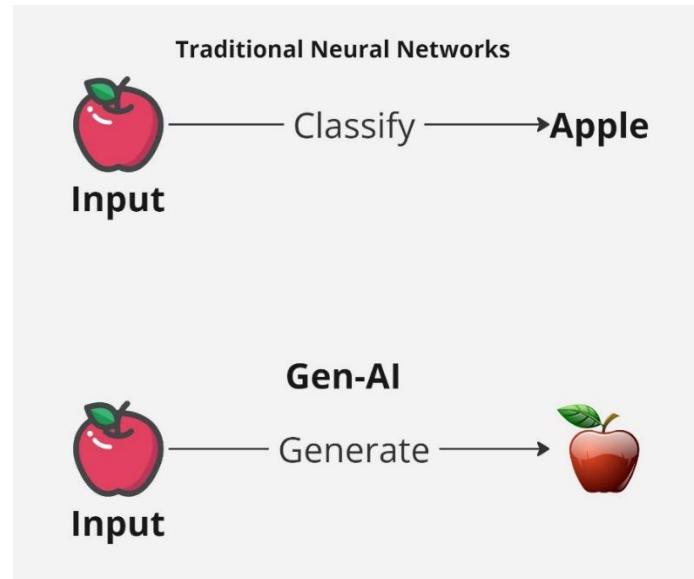


Figure 14-0-5 Gen-AI can generate text, images, etc.

Generative AI, which is a subfield of artificial intelligence and deep learning, focused on developing models and techniques that can generate creative, coherent, and contextually relevant content. Generative AI has gained significant attention in recent years due to the advancements in models like GANs (Generative Adversarial Networks), VAEs (Variational Autoencoders), transformers, and other creative AI approaches. These models have demonstrated the ability to generate realistic images, compose music, generate text, create art, and even translate text between languages.

Generative AI models, such as those based on the Transformer architecture, have made it possible for individuals and organizations to harness the power of AI-generated content and creativity without requiring an in-depth understanding of data science or machine learning. These models allow users to generate text,

images, music, and more, expanding the scope of creative expression and content generation. The outputs generated by these AI models are often remarkable in their creativity, accuracy, and coherence.

Generative AI models learn from a dataset, which could be anything from text to images to music. They analyse the patterns and features present in the data and create a statistical model that captures the essence of the data's distribution. These models generate new data based on the statistical probabilities they've learned from the training data. They create outputs that are likely to occur according to the learned patterns but may introduce variations or creativity.

Generative models encode a simplified version of the data's distribution in their parameters. This encoding enables the model to generate new samples that share characteristics with the training data but aren't just simple copies. One of the strengths of generative models is their ability to produce new and diverse outputs. They can generate variations of the original data, which can be valuable for tasks like data augmentation, content creation, and more.

As the capabilities of generative AI models become more apparent, individuals, organizations, and governments are faced with the need to adapt to this new reality. This adaptation involves recognizing the potential benefits and challenges posed by AI-generated content and determining how to regulate, manage, and use it responsibly. By 2022, various powerful AI models and tools became accessible to the public, contributing to the widespread adoption and interest in the field. Dall-E, Open AI's Chat GPT, text-to-3D, etc. were some of the prominent generative models, which surfaced in 2022.



Figure 14-0-6 ChatGPT Example

Above picture, is a perfect example of the powers of Gen AI. End user **prompted** the GPT model to write a review about a movie. GPT model came up with a review which accurately captures the essence of "Batman Begins," including its themes, characters, and the director's style. The model understands the context of the movie and its significance within the Batman franchise. The GPT models have been trained on a diverse range of internet text, books, articles, websites, and other sources. This extensive training data

allows the model to have access to a wide array of information, including plot summaries, analyses, reviews, and discussions about earlier Batman movies.

In the period of 2022-23, numerous Large Language models (LLMs) stamped their arrival. Microsoft took a stake in OpenAI's GPT model and brought it on Azure platform, thus making it accessible for enterprise use. Parallelly, Google has been working on Vertex AI, Amazon has been working on AWS Bedrock, Meta (previously Facebook) release an open source LLM, Llama.

With every passing day, as the book is being compiled in year 2023, Gen AI models' capabilities are growing exponentially, and enterprises are harnessing its power in numerous ways. Keeping aligned with our goal to understand the convergence of Object, Data and AI with enterprise applications, let us try to understand how Gen AI's capabilities can be utilized directly by software developers.

Before getting into the details, let us formally define what a Large Language Model is.

15.4.1. Large Language Models (LLMs)

A large language model (LLM) is a type of artificial intelligence (AI) model that has been trained on a massive dataset of text and code and is designed to understand, generate, and process human language. Large language models have been trained on trillions of words over many weeks and months, and with large amounts of compute power. Inspired from transformers-based architecture, LLMs are trained to learn patterns, grammar, syntax, and semantics of human languages. These models are capable of performing a variety of language-related tasks, including text generation, translation, summarization, sentiment analysis, and more. These **foundation models** with billions of parameters, exhibit emergent properties beyond language alone, and data scientists are unlocking their ability to break down complex tasks, reason, and problem solve.

LLMs are built with a significant number of parameters, often ranging from hundreds of millions to billions. The sheer scale of these models contributes to their ability to understand and generate human-like text. For example, the GPT-3 model has 175 billion parameters, which is more than 100 times the number of parameters in a typical machine learning model. Some other popular LLMs are BERT, BLOOM, LLaMa, PALM, etc. These foundation models, also referred as base models, differ in size on the basis of parameters. Parameters can be thought of as memory of a model, meaning, more parameters a model has, better would be its task performing ability.

These models undergo two main phases: **pretraining and fine-tuning**. During **pre-training**, models are trained on a diverse range of text data that empowers them with the deep statistical representation of language. During this phase, LLMs learn from huge amount of unstructured textual data, ranging range from gigabytes to petabytes, and consists of variety of sources including internet scrapes and specially curated corpora (a selected and organized collections of textual data that have been assembled and processed for specific purposes, such as training language models or conducting research in natural language processing). This self-supervised learning step is where the model internalizes the intricate patterns and structures inherent in language. We have already discussed about training of a neural network in earlier chapter, hence, I shall not be going into the details

While pre-training helps in predicting the next tokens or words, it is not same as following commands. **Fine-tuning** involves training the model on specific tasks or datasets to adapt it to perform those tasks effectively. The GPT-3 model was trained on a dataset of text and code that was over 500GB in size. By either using these models in their original form or by applying fine tuning techniques to adapt them to an enterprise specific use case, we can rapidly build customized solutions without the need to train a new model from scratch.

15.4.2. Challenges of training LLMs

There can be three major hurdles while training an LLM: Data, Hardware Resources and Legal Challenges. Pre-training (or training) an LLM is a compute-intensive process and requires significant processing power, which is fulfilled by utilizing specialized accelerators such as GPUs or Google's custom-developed Tensor Processing Units (TPUs).

Let us look at the scale of the problem mathematically:

A single parameter is represented by 32-bit float, which takes up 4 bytes of memory. Hence, to store one billion parameters we will require four bytes times one billion parameters, or four gigabytes of GPU RAM at 32-bit full precision. This is a lot of memory, yet it only accounts for storing the model weights so far. Training LLMs require additional components such as Adam optimizer states, gradients, activations, and temporary variables, which can easily account for roughly 20 extra bytes of memory per parameter, which means to account for all training overheads, we will end up consuming 20 times the amount of GPU RAM that the model weights alone would take up. Hence, to train a one billion parameter model at 32-bit full precision, we will be requiring roughly 80 gigabytes of GPU RAM, and that's huge. Definitely too large for common consumer hardware and even for the hardware used in data centres of big enterprises.

One technique used for reducing the memory footprints in LLM training is called **quantization**, which focuses to memory required by a single parameter from 32-bit floating point numbers to 16-bit floating point numbers, or eight-bit integer numbers. Quantization statistically projects the original 32-bit floating point numbers into a lower precision space, using scaling factors calculated based on the range of the original 32-bit floating point numbers. However, an elaborate discussion on this topic would be out of scope for the book.

As LLMs are scaling beyond a billion parameters, they require access to 100s of GPUs, which makes a point that pre-training a model from scratch for a particular use case is definitely a bad idea.

Training Google's PaLM (a large language model (LLM) trained on 100+ languages that can perform text processing, sentiment analysis, classification, and more) required 6144 TPU v4 chips made of TPU v4 pods, which were connected over a data center network. Meta's OPT (a language model with 175 billion parameters) was slightly efficient in utilizing resources yet consumed 992 80GB NVidia A100 GPUs.

Heavier the hardware is, more chances would be towards failure and it would require intervention either manual or automatic to restart the training process. Meta AI has publicly acknowledged that “*in total, hardware failures contributed to at least 35 manual restarts and the cycling of over 100 hosts over the course of two months. During manual restarts, the training run was paused, and a series of diagnostic tests were conducted to detect problematic nodes. Flagged nodes were then cordoned off and training was resumed from the last saved checkpoint. Given the difference between the number of hosts cycled out and the number of manual restarts, we estimate 70+ automatic restarts due to hardware failures*”.

Although LLM training is a resource intensive process, still it does not come with the benefit that LLMs can be trained quickly. Rather, training LLMs is a time-consuming process and requires thousands of compute days. To balance things out, enterprises utilize a mixture of parallelism techniques which partition the models into segments, process data in parallel, and efficiently utilize the available compute resources.

In **Model parallelism**, the model's parameters are partitioned across multiple devices (GPU or TPU hardware). Each device is responsible for computing a specific part of the model. Output of each GPU/TPU is consumed sequentially by the next device in the pipeline. Each GPU is like a compartment in the factory production line, it waits for the products from its previous compartment and sends its own products to the

next compartment. This is particularly useful for models that are too large to fit entirely in the memory of a single device. However, in a sequential process, when one device is computing, other devices are sitting idle and this can be resolved by switching towards an asynchronous style of GPU functioning.

Another approach, **Data parallelism** involves sharding the dataset across multiple devices and training each replica on a different subset of the training data (mini-batch). Each replica computes gradients based on its local mini-batch, and the gradients are then averaged across all replicas. This approach increases the overall training throughput, as multiple replicas can process data simultaneously.

Judging by just tip of the iceberg, it is safe to assume that training LLMs generates a high carbon footprint.

A significant research paper '*Carbon Emissions and Large Neural Network Training*', published in 2021, estimated that GPT-3 used 1,287 gigawatt-hours, generating 502 tons of carbon emissions, which is equivalent to 120 years' worth of a single U.S. family's electricity use. Actual numbers can be considerably higher. Meta AI claims to be much more efficient in terms of carbon footprints, as it claims to emit 1/7-th of GPT3 carbon emission. However, with increasing parameters size in new LLM models, carbon cost of LLM training would definitely be a factor with law makers taking this into cognizance.

A **petaflop** per second day represents a measurement of performing one quadrillion floating-point operations in one second, continuously for an entire day. When considering training transformer models, achieving this level of computational performance is roughly equivalent to using eight NVIDIA V100 GPUs running at maximum efficiency for a full day. If a more powerful processor is used that can handle more operations simultaneously, then fewer chips would be required to achieve the same level of computation. For instance, two NVIDIA A100 GPUs can provide the same compute capacity as the eight V100 GPUs.

From the above discussion, we can infer that decision to train an LLM efficiently hinges upon three parameters: Compute Budget, Dataset size and Parameter size. LLM makers ultimately put a definite limit on computer budget and look to increase either dataset size or parameter size or both, to get an overall improvement in performance of output. The Chinchilla paper, "*Train Computer Optimal Large Language Models*" carried out a detailed study of the performance of language models of various sizes and quantities of training data to find the optimal number of parameters and volume of training data for a given compute budget. This paper suggests that many large LLMs may actually be over parameterized and under trained. They hinted towards using low parameters and higher dataset for an LLM model. However, this composition can vary case to case, as these are just early days (year or writing 2023) for LLM research. The Chinchilla paper did indeed inspire the organisations to work on developing smaller compute optimal models that can achieve on par results, if not better, in comparison to the bigger models like GPT3. Meta's LLaMa is one good example, which is trained on a dataset size of 1.4 trillion tokens and has 65 billion parameters and gives a decent performance. Bloomberg GPT model trained on 50 billion parameters has been gaining traction for its performance as well.

It is important to understand that data quality plays a pivotal role in the process. While training data is often sourced from the internet, there is a need to curate and process the data to enhance its quality, address biases, and eliminate harmful content. This curation can result in a smaller subset of tokens, generally only 1-3% of the original data, being used for actual pre-training phase.

However, a significant bottleneck which can be very prominent is a realistic concern that LLM makers will run out of Internet data in the next few years. And it is a serious concern. Imagine, if a book contains 50,000 words or an average 68,000 tokens, then 1 trillion tokens would be equivalent to 15 million books. At this rate, data generation would be overrun by data being used in training process of LLMs. So, it leads to a

scenario where LLMs might go into securing massive amount of proprietary data – copyrighted books, translations, video/podcast transcriptions, contracts, medical records, genome sequences, user data, etc.

And last but not the least, we have legal troubles with LLMs. With Social Media giants like Meta, Google, etc. already under scanner in many countries for data privacy issues, LLMs just add to their burden significantly. Several artists and writers have even sued the LLM makers for ingesting their content into their model, without copyrights. Then there is the case of LLM giants eating up the competition. In short, training LLMs come with a very significant cost.

15.4.3. When to go for LLM Training?

The intention of discussion in previous section was not to deter the reader from taking the route of training LLMs, but to make the reader aware of the outlying challenges. However, there are certain use cases, where training LLMs is required.

Consider the following medical text, taken from the very famous book “What’s going on in There” by Lise Eliot.

The similarity between ontogeny and phylogeny shows that the strategy of early development has been highly conserved in evolution. This makes sense, if you think about the precise timing and series of events necessary to turn a single fertilized egg into many different complex organ systems; it’s simply much easier to add changes at the end of a common developmental sequence than to alter things from the outset. A slight change early in neurulation, for example, could invalidate all kinds of later, subtly timed cues, throwing off the whole process of brain formation. (Just such a problem occurs in spina bifida, a relatively frequent condition in which part of the spinal cord is not fully enclosed because of a defect in the early neural tube.) It has been much easier for evolution to take an existing structure, like a forelimb, and turn it into a wing, or a primate cerebral cortex, and enlarge it into the human cortex, than to start with a whole new game plan for each species. Evolution proceeds through the selection of random mutations, and the later in development such a change occurs, the likelier it is to produce a viable offspring than a horrible mistake. Indeed, this is why miscarriages are more common in early pregnancy, a topic that will be discussed more fully in the next chapter.

The above text (although presented in a very simplified manner) is clearly written in a language which don't use in our daily usage. Similarly, such medical language or even more complex may be common for healthcare organizations and medical applications for their internal operations, but not available in public data domain especially web scrapes and book texts. This means that the current LLMs trained over available data, will not have the specific information in their datasets. Since, models learn about a language and its vocabulary in their pre-training phase, in this case, pre-training a data model from scratch for a medical entity makes a valid case. Similarly, models can be pre-trained targeting other specific domains such as law, finance, etc. Bloomberg GPT model is one such example which is trained on 51% financial data. Hence, enterprises building applications for specific domains can ponder over training an LLM from the scratch for the sake of domain adaptation and better performance over generally available LLMs.

15.4.4. Prompt Engineering

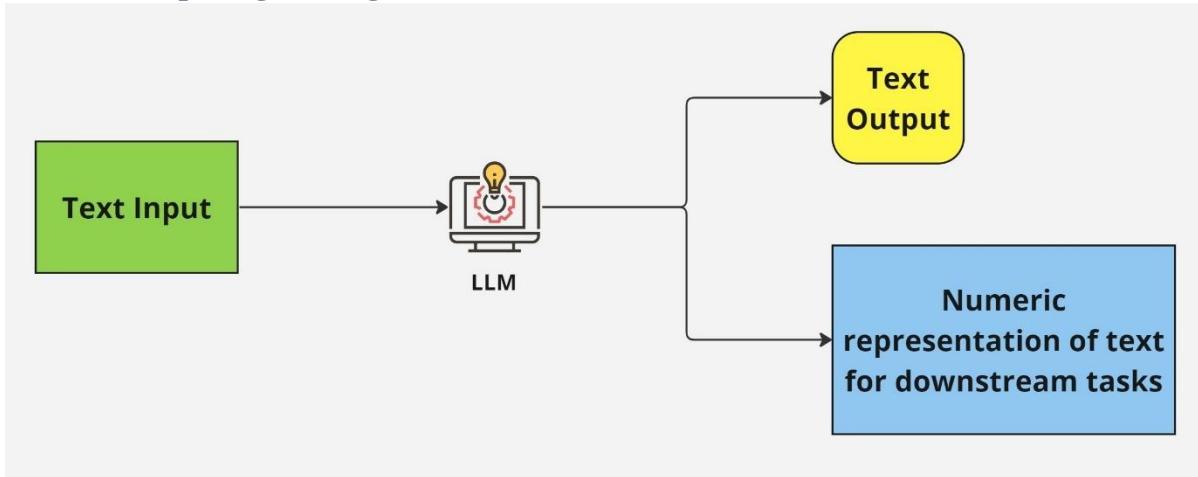


Figure 14-0-7 LLM working pattern

LLMs can understand context in text, allowing them to generate coherent and contextually relevant responses. This contextual understanding is achieved through mechanisms like attention mechanisms in transformers.

The text input provided to a LLM model is called **prompt**. A prompt can be a sentence, a question, a command, or any other form of input that you want the model to continue from or respond to. For example, if you want the model to generate a story about a detective solving a mystery, your prompt might start with *“Once upon a time, in a quiet town, Detective Smith was called to investigate...”*. Prompts set the context and topic for the generated text. Well-crafted prompts are key to obtaining relevant and coherent outputs.

LLMs process text in form of **tokens**. Tokens have been discussed in Chapter 12 under section ‘Dealing with text data’. Tokens are the fundamental units of text that a language model processes. A token can be as short as a single character or as long as a word. For example, “GPT is great!” is broken down into four tokens: [“GPT”, “is”, “great”, “!”]. Tokens determine the granularity of text that the model processes. They have a direct impact on how much computational power and memory a model uses, as well as how much it costs to generate or process text.

Context refers to the information that a language model uses to understand and generate text. It includes the prompt as well as preceding text that provides background and sets the stage for the model's response. Context is crucial for generating coherent and relevant text, as the model uses the context to understand the tone, style, and content of the text it's expected to produce. When we started this book, in the very first chapter, we set the context of the book, about what it is and what is to be expected. Context of LLM is pretty much on the same lines.

Model confidence refers to the level of certainty that a language model has in its own responses. It's an indicator of how likely the model believes its generated output is correct or accurate. In OpenAI's GPT models, you can use parameters like *“temperature”* and *“max_tokens”* to influence the model's confidence and creativity. A lower temperature value makes the output more focused and deterministic, while a higher value increases randomness and creativity.

The prompt is passed to a model and then it predicts the next words, and because our prompt contained a command to write a review of a movie, in Figure 12-6, this model generates an answer which has a detailed review of the said movie. The output of the model is called a **completion**, and the act of using the model to

generate text is known as **inference**. The completion is comprised of the text contained in the original prompt, followed by the generated text.

To extract the desired results from LLMs, we need to be very good at providing them with proper prompts and setting the context. However, as a new user to LLMs, it won't be possible to give appropriate prompts from the first go. After all we are humans, and we learn by trial and errors! This process of refining our prompts over time is called **Prompt Engineering**.

There are different ways to provide prompts to LLMs. You can either give instructions to it, such as "*write an email to my boss complaining about work pressure but use a subtle tone*" or you can give the model a role to play, for e.g. "Imagine you are Charles Dickens, write a short story of a boy on journey to Atlantic". Another prompting strategy could be to provide examples in prompts and then asking model to do the task. Look at the example below:

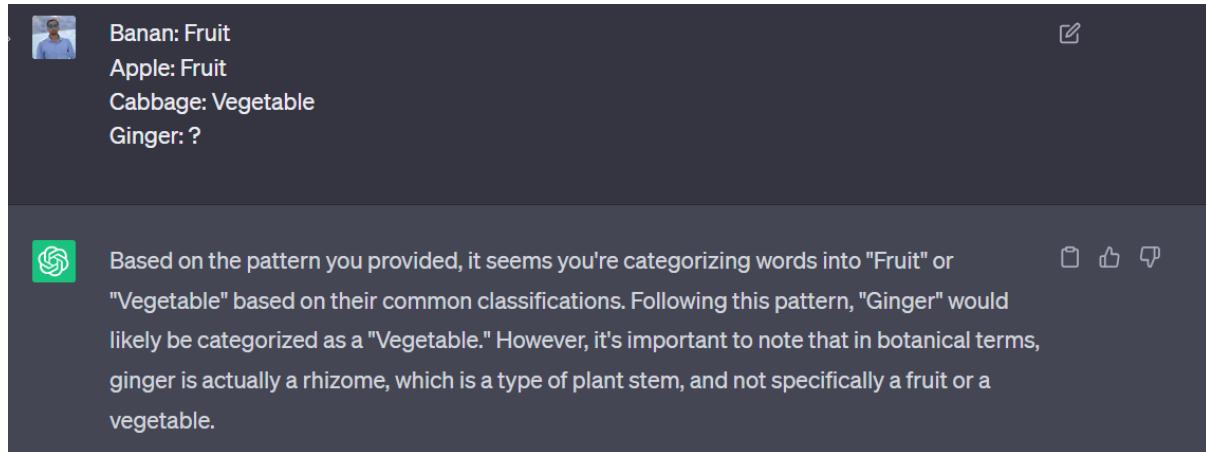


Figure 14-0-8 Few shot prompt engineering

Notice, how it is able to get the context and also the mistake in the spelling of "Banana" is deliberate. We can even combine the strategies discussed above or get innovative with prompts and observe the results and fine tune it. Prompt engineering is more intuition-based learning than theoretical learning. Practice will get you better at it.

LLMs with bigger parameters are able to perform classification or sentiment analysis tasks easily on inputs provided in the prompts (**Zero shot inference**, without example like in Figure 12-6), however, LLMs with smaller parameter size might fail to do so, hence, it is recommended to provide examples of the task to be performed, as shown in the example above (**Multi shot inference**, as multiple examples provided in context).

To enhance the performance of LLMs, examples are used in the context and this process is called **in-context learning**. When utilizing in-context learning, we include specific instances, sentences, or snippets of information related to the task we are asking the LLM to perform. These examples serve as contextual cues that guide the model's generation process in a targeted manner. The idea behind in-context learning is to guide the LLM's generation process by offering explicit cues, examples, or data that align with the desired task. This can help improve the relevance, accuracy, and coherence of the generated output. However, it's important to strike a balance, as excessive examples might overly constrain the model's creativity, while insufficient context might lead to vague or incorrect outputs.

We may have to try a few LLM models and test our examples with them to understand which are best suited for our task.

15.4.5. LLM Configuration

LLM models usually expose a set of configuration parameters which has the ability to influence a model's output during inference. These parameters are not to be confused with the training parameters, as they are invoked at inference time. Let us look at some of these configuration tokens:

Max new Tokens: This parameter, also known as **max length**, is used to limit the number of tokens in generated output.

Top-k Sampling: By setting a value for “ k ”, Top-k sampling limits the selection of next tokens to the $top-k$ most probable choices. It encourages the model to choose from a smaller set of tokens, which can lead to more controlled and coherent outputs. This method can help the model have some randomness while preventing the selection of highly improbable completion words.

Top-p Sampling (Nucleus Sampling): Alternatively, with the $top-p$ setting to limit random sampling, the model selects tokens that cumulatively exceed a specified threshold “ p ” in probability. This technique allows for variability while maintaining context. In *Top-k* approach we were specifying the number of tokens but in this approach, we specify the total probability.

Temperature: Temperature setting influences the shape of the probability distribution for the next token. The probability distribution represents the likelihood of each possible token being the next one in the sequence. When generating text using the Softmax layer, the model calculates the probabilities of all tokens in the vocabulary for the next token. These probabilities are influenced by the internal representations learned during training. The Softmax function then normalizes these probabilities, converting them into a distribution where the sum of all probabilities is equal to 1.

A higher value (> 1) increases randomness since probability distribution becomes more spread out, resulting in more diverse outputs. The distribution becomes flatter, with multiple tokens having relatively similar probabilities. This increases the chance of generating unexpected and novel outputs. A lower value (< 1) has the effect of sharpening the probability distribution and makes the model more deterministic and focused. It becomes more certain in its predictions, favouring tokens that it's more confident about. The distribution becomes more peaked, with one or a few tokens having significantly higher probabilities than the rest. As a result, the generated text is more deterministic and follows paths that the model learned with higher confidence during training.

Choosing the appropriate temperature value means balancing creativity and coherence. A lower temperature might be suitable for tasks where generating coherent, contextually consistent text is crucial, while a higher temperature could be used for generating more creative or diverse responses, even at the risk of occasional outputs which wander off the topic or might not make sense at all.

15.4.6. Gen AI Project Lifecycle

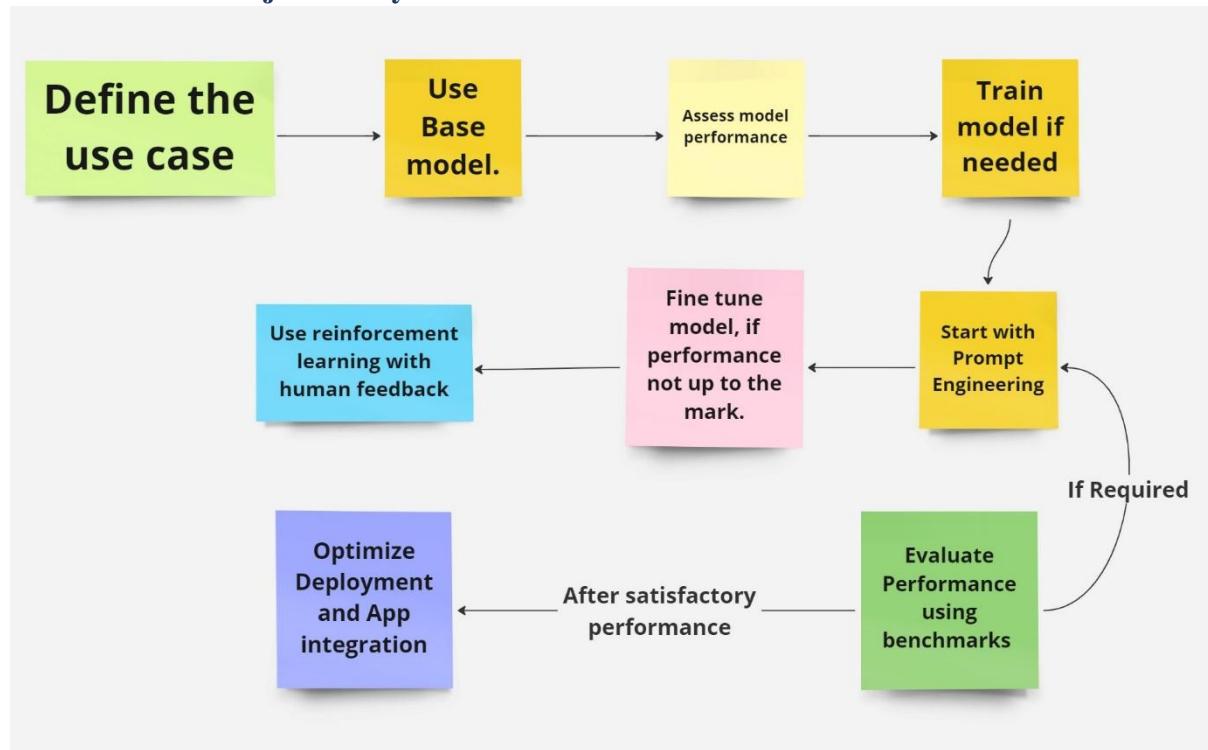


Figure 14-0-9 Gen-AI project Lifecycle

LLMs are hugely capable AI models with ability to perform varied tasks, however, their ability depends on size and architecture of the model. Certain LLMs are trained for a specific task only. Before utilizing an LLM, we must be doing a thorough research on their capabilities. Good amount of documentation is usually present about LLMs on their respective provider's website. Most of the LLM providers also provide a model playground on their website, so that users can try out LLM capabilities before initiating with project integration.

We should be thoroughly defining the use case and outline the scope of the project as narrowly as possible by specifying the desired outcomes and objectives. This step is crucial for cost evaluation of a model.

Usually, base models are equipped enough to perform user specific tasks, however, in peculiar cases, there can be a possibility to train a model from scratch.

We should be checking the performance of a model and adapting the model using prompt engineering to perform required tasks. Still, if we are not satisfied with the outcome, we can consider the scenario of finetuning the model or using reinforcement learning with human feedback to get the model working. Anthropic (an AI safety and research company based in San Francisco) states that: *“we expect human feedback (HF) to have the largest comparative advantage over other techniques when people have complex intuitions that are easy to elicit but difficult to formalize and automate”*. This is done to optimize performance and keeping a tab whether outcomes are aligned. Model performance should be evaluated using appropriate metrics. We should validate the model's outputs against ground truth or human judgment to assess its quality and accuracy.

The above process can be highly iterative, and developers may again have to adjust prompts or fine tune the models to derive the desired output. Once the model starts producing desired output in alignment with

requirements, model can be deployed and integrated with user application. We can choose to configure inference parameters like max length, sampling technique or temperature.

15.4.7. Fine Tuning LLMs

While discussing Prompt Engineering, we discussed extensively about different in-context learning techniques like Zero-Shot inference or Few-Shot inference while prompting LLM models. However, the performance of an LLM might hit the ceiling, even with the use of these techniques and leaving a scope for better performance. To offset this limitation, especially when fine tuning our prompts do not work well in producing optimal output, we can consider fine-tuning LLMs. It is also trivial to not the context space taken up during multi-shot inference, when we provide around 5-6 examples in the context window of an LLM and then include user command in the prompt. Especially with smaller LLMs (yet cheaper), in context learning strategies may hit limitations more often than the large ones. However, if a smaller, open-sourced, and cheap LLM can give better performance or at least on par performance with fine-tuning with respect to the industry leader LLMs, then it is worth giving it a shot.

So how is fine-tuning different from pre-training? Let us address this difference first. In pre-training phase, an LLM is trained over a vast pool of unstructured data via self-supervised learning, but in fine-tuning an LLM, a supervised learning process is used over a dataset of labelled examples to update the weights of the LLM. These labelled examples can be pairs of command and action or prompt and completion pair. The provided examples allow the model to learn to generate responses which are in adherence of the examples. Fine-tuning is done to extend the capabilities of a pre-trained LLM to give better performance and generate more suited output to the required use case.

15.4.7.1. Full Fine Tuning

The process of fine-tuning via instruction examples, where all of model's weight are updated is called as **full fine-tuning**. It requires enough memory and compute budget just like pre-training phase, to process and store gradients, optimizers or any other components being updated. And for the same reason, we can leverage memory optimization and parallel computing strategies discussed in earlier sections for improving the process of fine-tuning.

For fine-tuning, you can either work hard and create your own specific dataset, or else you can borrow from the available prompt template libraries for specific tasks. Once the dataset is ready, the obvious next step is to divide it into training, validation, and test sets. Prompts from training datasets are selected and passed to the pre-trained model which generates a response, which is compared against the actual response specified in training data. As we know, output of LLMs is a probability distribution across tokens, so metrics like cross entropy are used to compare the distribution of resultant response and actual response in training dataset, loss is calculated and model weights are updated using backpropagation technique, which has been discussed in the earlier chapter. This process is done for multiple batches of data from training set and several epochs and model weights are adjusted accordingly till the model's performance on the task improves. We can calculate validation accuracy over validation dataset and a final performance test of fine-tuned model can be performed over test dataset. The new data model which has been fine-tuned for a specific task is also generally called **instruct LLM**.

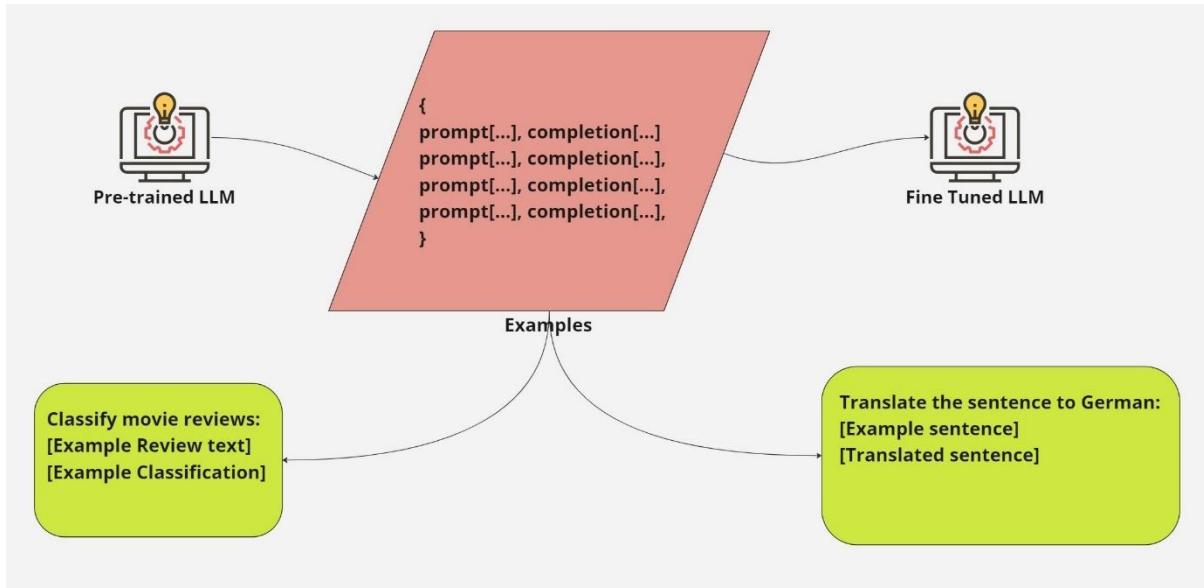


Figure 14-0-10 Instruction Fine Tuning

While LLMs can perform variety of tasks related to text generation, an enterprise use case may only require it to perform just one task. In that case, it is advisable to fine-tune LLMs on a specific task, and even with a smaller dataset of 500-1000 examples, instruct LLMs can achieve better performance at the specific task.

Caution: Although, fine-tuning may seem likely to be a go to approach for better performance, this path should be traded with caution. Fine-tuning involves weight modification of the original LLM, hence, the instruct LLM after fine-tuning may get better at a specific task but may lose its ability to perform well on other tasks. This phenomenon is called **Catastrophic forgetting**.

The decision of fine-tuning a model on specific task or multiple tasks could be a very tricky one. If we are sure about using the model for just one specific task, then only single task fine-tuning should be considered, or else, we should approach **multi-task fine-tuning**.

In multi-task fine-tuning, examples of input-output are provided for multiple tasks such as summarization, classification, review, code generation, etc. The LLM model is trained over many epochs of training and calculated loss across examples are used to update the model weights, which will likely improve the performance of the resultant instruction model. However, multi-task fine-tuning would definitely require higher number of dataset examples in comparison to single-task fine-tuning. In general, it is a good idea to provide similar ratio of dataset examples on multiple tasks. The dataset size would be definitely larger than the single task specific fine-tuning, ranging from 50-100,000s datasets. Assembling of such large datasets can be quite a challenge in itself. However, good new is that task specific datasets are also available on hubs like HuggingFace website. We can even leverage already fine-tuned LLMs like FLAN-T5, which has been fine-tuned over wide variety of datasets. Many open source LLMs are being fine-tuned on various tasks and then the update versions are being uploaded to hubs like HuggingFace, where other users can benefit from it. Again, it is worth mentioning that additional fine-tuning can be performed over an already fine-tuned model like FLAN-T5, if the performance of the model is not satisfactory for the required use cases.

Developers can use several performance metrics and benchmarks to compare the performance of one model over other, or even compare the original LLM and the instruct LLM. However, must take into cognizance, that in earlier machine learning models, evaluation of performance or accuracy of the model was easily

retrievable as the output of the model was deterministic. On the contrary, LLMs have non-deterministic output, and it makes the evaluation process challenging.

For example, “I love coffee very much” and “I love you very much”, are two different sentences with different meaning, which appear similar with just one-word difference. On other side, “I like to sip coffee” and “Sipping coffee is my favourite pass time” are two sentences, who meaning are quite similar, yet their structures are quite different.

15.4.7.2. Parameter Efficient Fine-Tuning

Yet another strategy that can be applied to fine-tune LLMs is **Parameter efficient fine-tuning (PEFT)**, which preserves the weights of the original LLM and trains only a small number of task specific adapter layers and parameters. Full blown fine tuning is an expensive process as model weights are updated and there are several other parameters like gradients, optimizer states, etc. which add to the cost of running the fine-tuning process. Most of the times, it would not be possible to run on consumer hardware as this much memory allocation will not be feasible. In contrast, PEFT targets only subset of model weights and is comparatively efficient to run on consumer hardware. As a matter of fact, data scientists are able to run the PEFT on a single GPU. It also helps avoid Catastrophic forgetting as well, since, the majority of original weights of the LLM remain intact. There are several methods we can use for parameter efficient fine-tuning, each with trade-offs on parameter efficiency, memory efficiency, training speed, model quality, and inference costs.

Methods for PEFT are broadly divided into three classes based on their approach. **Selective methods** involve fine-tuning only a subset of the original LLM parameters. This approach aims to reduce computational demands by focusing updates on specific parts of the model. Possible strategies include training only certain components (such as the encoder or decoder), specific layers, or individual parameter types (such as attention weights or feed-forward weights). While selective methods offer the potential for efficiency gains, researchers have observed mixed performance results, and there are trade-offs between parameter efficiency and compute efficiency.

Additive methods involve fine-tuning the LLM by introducing new trainable components while keeping the original weights frozen. There are two main approaches within this category: **Adapter Methods** and **Soft Prompt Methods**. Adapter methods add new trainable layers to the existing model architecture, typically inserted after attention or feed-forward layers within the encoder or decoder components. In Soft Prompt methods, the model architecture remains fixed and frozen, and the focus is on manipulating the input to improve performance. This can involve adding trainable parameters to the prompt embeddings to enhance task-specific signals or it can entail keeping the input fixed and retraining only the embedding weights to adapt the model to different tasks.

Reparameterization methods work with the original LLM parameters but aim to reduce the number of parameters that need to be trained. This is achieved by creating new low-rank transformations of the original network weights. Examples of this approach are LoRA (Low-Rank Adaptation) and Q-LoRA (Quantized-LoRA), which involve approximating the original parameters with a smaller set of parameters while retaining their essential characteristics. In fact, by the time of writing this book, Q-LoRA has gained significant popularity. Reparameterization methods attempt to strike a balance between reducing computation and maintaining model effectiveness.

Q-LoRA is an extension of LoRA that further introduces quantization to enhance parameter efficiency during fine-tuning. It builds upon the LoRA principles and introduces two key techniques: **NF4 Quantization** and **Double Quantization**.

NF4 quantization takes advantage of the underlying weight distribution of pretrained neural networks. It transforms weights into a fixed distribution that fits within the NF4 range (-1 to 1). By doing this, NF4 quantization effectively quantifies weights without requiring complex quantile estimation algorithms. This process simplifies quantization and avoids the need for expensive computations.

Double Quantization addresses the memory overhead associated with quantization constants. It reduces memory usage while maintaining performance by quantizing the quantization constants themselves. This is achieved using 8-bit Floats with a block size of 256 for the second quantization step. This approach significantly reduces memory consumption, making it more memory-efficient.

Despite its focus on parameter efficiency, Q-LoRA maintains high model quality. It demonstrates comparable or even superior performance compared to fully fine-tuned models across various downstream tasks while being memory efficient.

15.4.7.3. Evaluation and Benchmarks

It is easy for human eyes to observe the similarities and differences but obviously machines cannot observe. They need a more structured way to make measurements. ROGUE (Recall Oriented under study for jesting Evaluation) and BLEU (Bilingual Evaluation) are two prominent evaluation metrics used to gauge performance of the models. ROGUE is used to assess the quality of model generated text summaries by comparing with other human generated summary references and BLEU is used to evaluate the quality of translated text by comparing with human generated translations. Under the hood, they both run the evaluation by breaking the inputs and outputs into multiple size n-grams and then compare these n-grams using certain algorithms. The detailed discussion on this topic would extend the scope of this book. Moreover, these two are relatively simple metrics and are implemented as external libraries with providers like HuggingFace, where one can run the diagnostic evaluation of the LLMs performance after fine-tuning.

The above metrics help us judge the capability of a model (LLM), however, to gauge the performance of LLMs more holistically, we must use existing datasets and benchmarks which have been established by LLM researchers for the same purpose. Selection of dataset is based upon the specific model skills and we should carefully choose datasets which the model has not seen during training. Benchmarks such as GLUE, SuperGLUE or Helm make use of these specific evaluation datasets to test the specific aspect of the LLM. These benchmarks have their respective leader boards where they list the evaluated models and compare their performance to match human ability. These benchmarks are usually competing with the growing size of LLMs and keep updating their own set of parameters and algorithms to evaluate the performance.

Consider GLUE (General Language Understanding Evaluation) for instance, it was introduced in 2018 to evaluate model's capability to perform multiple tasks such as sentiment analysis or question answering, and was replaced by its successor SuperGLUE, introduced in 2019, to address the limitations (series of tasks not earlier included or more complex version of the earlier tasks) of the GLUE. These benchmarks cover variety of scenarios and tasks such as History, mathematics, computer science, etc. Massive Multitask Language Understanding (MMLU) and BIG-bench are latest benchmarks, designed specifically for evaluation of modern LLMs.

The HELM (Holistic Evaluation of Language Models) framework is designed to enhance the transparency of AI models and provide guidance on selecting models suitable for specific tasks. HELM adopts a multi metric approach, evaluating seven different metrics across 16 core scenarios. It also includes also includes metrics for fairness, bias, and toxicity, which are becoming increasingly important to assess as LLMs become more capable of human-like language generation, and in turn of exhibiting potentially harmful

behavior. This approach aims to highlight trade-offs between models and metrics, offering a comprehensive understanding of model performance.

15.4.8. Reinforcement Learning from Human Feedback (RLHF)

So far, we have covered how LLMs are trained and fine-tuned to provide optimal output, however, there is one aspect which we did not consider i.e. the alignment of LLM response as per human values. When we fine tune our model with instructions, we train our model to better understand human context (prompts) and generate more human like responses. However, this comes with its own set of challenges. Remember, LLMs are exposed to a huge bunch of text data sourced from internet, which frequently contains toxic language. As a result, LLMs may produce output which sound toxic or be termed bad behavior as per human standards. It may also happen that LLMs give misleading responses or even incorrect answers, as we are not hand picking the data LLMs are exposed to while training.

In the ideal scenario, the basic goal for LLMs is to maintain a standard into responses and at the very least, should not be producing offensive, discriminatory responses which further elicit criminal behavior. It should adhere to the human values: **helpfulness, honesty, and harmfulness**; in the responses produced. Hence, LLMs are further fine-tuned with human feedback to eliminate toxicity and reduce chances of producing incorrect information. One such technique is called **Reinforcement Learning from Human Feedback (RLHF)**.

It is a powerful technique used to fine-tune large language models (LLMs) by incorporating human feedback into the reinforcement learning framework (briefly discussed in section 12.3.1.5). By leveraging reinforcement learning, RLHF enables LLMs to learn from human-provided guidance, aligning model outputs with human preferences, maximizing usefulness, and minimizing potential harm.

The core principle of reinforcement learning is for an agent to learn to make decisions within an environment to achieve a specific goal by taking actions that maximize cumulative rewards. RLHF adapts the same principle to language models by using human feedback as a form of reward or guidance for the model. This feedback can range from explicit ratings or preferences to corrections and explanations, helping the model understand what outputs align best with human expectations.

In the earlier section 12.3.1.5, we had taken an example of training a model to play a game of chess. In this scenario, the **environment** is the chessboard, and the **state** at any given moment represents the current configuration of pieces on the board. **Actions** available to the agent are the legal moves it can make based on the current board state. The policy of reinforcement learning guides the agent's decisions. Here, the goal of the agent is to win the game or achieve a favourable outcome, such as checkmate. The reward or penalty system is based on the effectiveness of the agent's actions. Successful moves leading towards winning may yield positive rewards, while suboptimal moves or losing positions may incur negative rewards or penalties.

The learning process in RL for chess involves an iterative approach, where the agent explores various moves, receives feedback (rewards or penalties) based on the outcomes, and adjusts its strategy or policy to improve future decisions. Initially, the agent might make random moves or explore various possibilities to gather experience and learn from trial and error.

Extending this concept to LLMs, models acting as agents aim to produce texts that align with human preferences and (state) bound to the current context (prompt) provided to the model. Generating sequence of texts would be analogous to actions in Reinforcement Learning concept, while the reward model assesses the generated moves or text for alignment with human preferences, providing feedback for model updates (see fig 15-0-11).

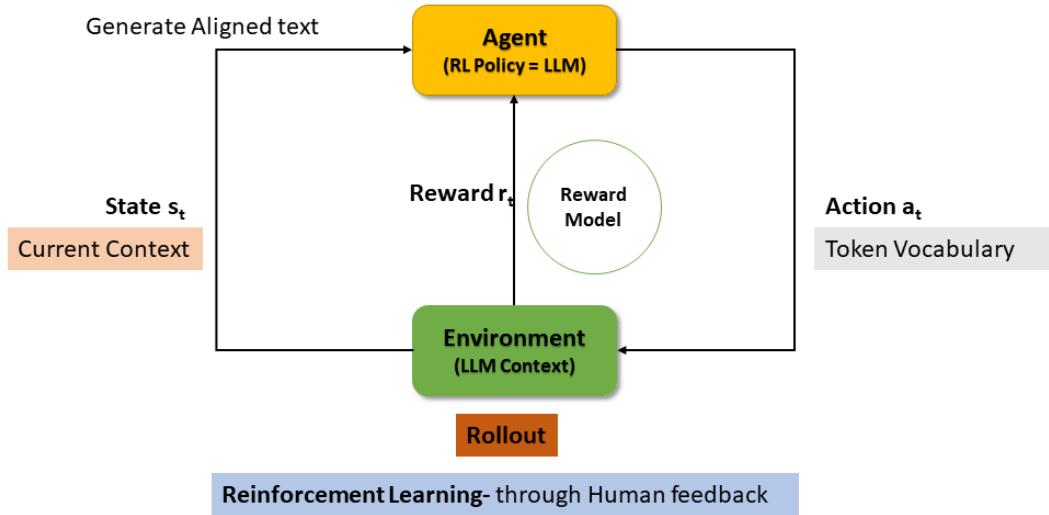


Figure 15-0-11 RLHF for LLMs

The decision-making process of a Large Language Model (LLM) in generating the next token in a sequence relies on its learned statistical understanding of language. During its training, an LLM learns the patterns, relationships, and probabilities between words or tokens in a given context. When determining the next token in a sequence, the LLM utilizes the information it has learned from the training data. At any moment during text generation, the LLM considers the context provided in the prompt or preceding sequence of tokens. It assesses the probability distribution over its entire vocabulary space to predict the most probable token to follow in the sequence. It assigns probabilities to each token in its vocabulary based on the context and the likelihood of each token occurring next. Based on the calculated probabilities, the LLM chooses the next token in the sequence by sampling from or selecting the token with the highest probability. This token becomes the model's output and extends the sequence.

However, assessing rewards based on human preferences presents challenges due to the variability in human responses to language. One approach involves human evaluation of model-generated text against alignment metrics, such as toxicity assessment, often represented as a binary scalar value (0 for toxic, 1 for non-toxic). However, this method is labour-intensive and costly.

To address these challenges and create a more scalable solution, a reward model, an additional model trained to evaluate alignment with human preferences, is introduced. Initially trained with a smaller set of human-labelled examples using traditional supervised learning methods, the reward model learns to classify outputs of the language model (LLM) based on alignment with human preferences, providing a scalable method to assess generated text. Once trained, the reward model assesses the LLM's outputs and assigns a reward value based on alignment with human preferences. This reward value guides the iterative update of LLM weights to improve alignment in subsequent versions of the model. How the weights get updates as model completions are assessed, depends on the choice of algorithm used for optimization.

In language modelling within the context of reinforcement learning, the sequence of actions and states undertaken by the model is referred to as a **rollout**. This term is specific to language generation tasks and

represents the process of iteratively selecting tokens or words based on the current state or context. Contrasting with classic reinforcement learning, where the term playout is used, a rollout in language modelling signifies the model's progression through a sequence of token selections to generate text or complete a language-based task.

Moreover, the reward model holds a pivotal position in the reinforcement learning process for language models. It serves as a central component, leveraging learned human feedback and preferences to guide the model's learning and updates.

An Instruct model, which has been fine-tuned over many tasks and has general capabilities, is the ideal candidate to start with RLHF. This model is provided with a prompt dataset, comprised of multiple prompts, which get processed by the model and produce a set of completions or outputs. Now, the feedback is collected from human labellers, who are chosen across the globe to represent diversity and help in ensuring a broader perspective, which ultimately helps in reducing the bias in completions (see fig 15-0-12).

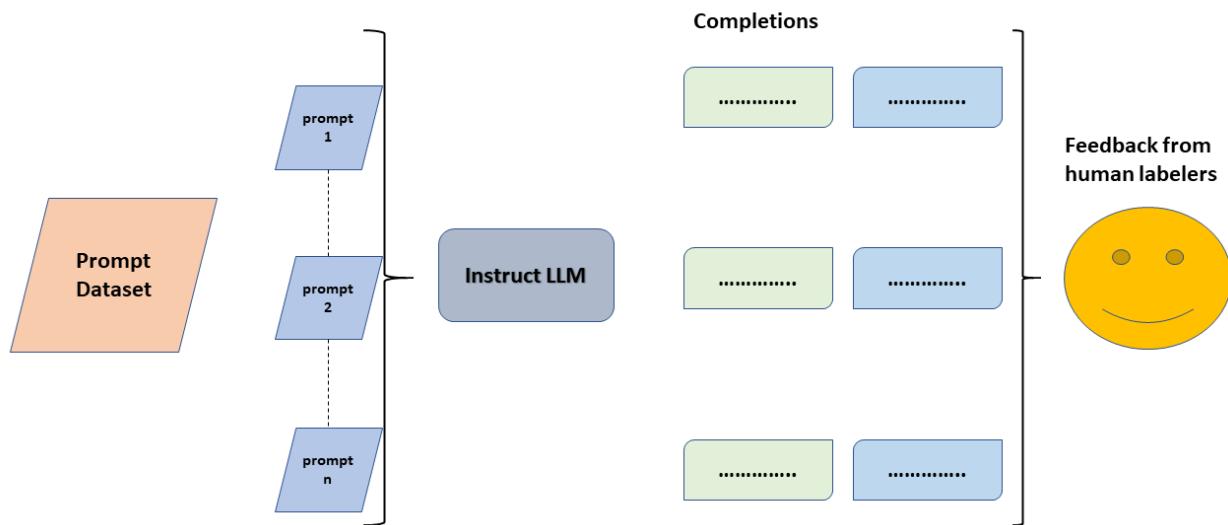


Figure 15-0-12 RLHF setup

Establishing consensus among multiple human labellers is a fundamental strategy to ensure the quality and reliability of labelled data. Assigning the same prompt completions to multiple labellers helps mitigate the impact of individual errors or misunderstandings, promoting consensus-based assessments and minimizing the influence of outliers or discrepancies.

Instructions provided to labellers play a crucial role in achieving accurate and consistent assessments. Clear and precise instructions help labellers understand the task, reducing ambiguity and ensuring that their assessments align with the intended criteria. Misunderstandings or discrepancies, as observed in cases like the third labellers' responses, may indicate unclear or ambiguous instructions that lead to divergent evaluations.

Once multiple labellers complete their assessments of prompt completion sets, the collected data serves as the foundation for training the reward model (see fig 15-0-3). The reward model acts as a substitute for human labellers in classifying model completions during the reinforcement learning fine-tuning process. However, before training the reward model, the ranking data obtained from human assessments needs to be

transformed into pairwise comparisons of completions. This involves comparing all possible pairs of completions generated for a given prompt and assigning a score (0 or 1) based on relative preference or alignment with human preferences. This transformation facilitates the creation of a reliable training dataset for the reward model, enabling it to learn and classify model completions based on their relative quality or alignment with human-labelled preferences.

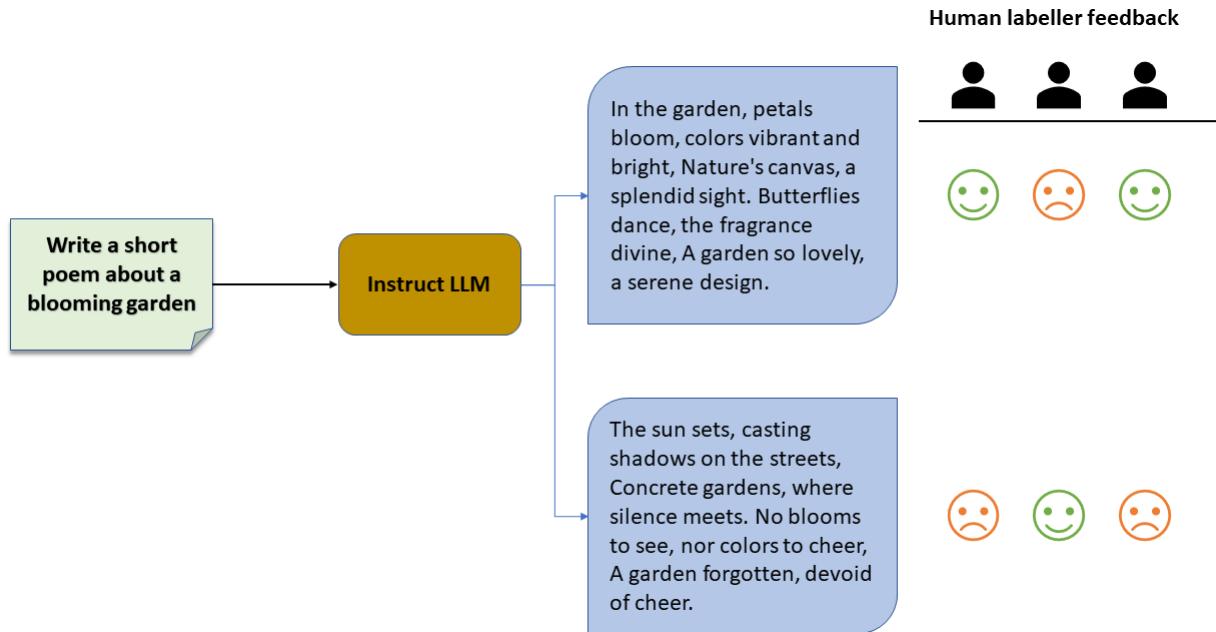


Figure 15-0-13 Feedback collection in RLHF

After training the reward model on the human-ranked prompt-completion pairs, the reward model acts as a binary classifier to assess model-generated completions. It evaluates the prompt-completion pair based on alignment with human preferences and returns a reward value indicating the alignment level (e.g., 0.24 for more aligned or -0.53 for less aligned).

This classification process involves assigning logits to the positive and negative classes, representing the unnormalized model outputs before applying any activation function. The obtained reward value is used in a reinforcement learning algorithm. This algorithm updates the weights of the LLM (RL-updated LLM) to optimize for higher rewards in subsequent iterations. These steps constitute a single iteration of the RLHF process.

Proximal Policy Optimization (PPO) is one such algorithm for the reinforcement learning component in RLHF due to its stability and performance in training models with policy-based methods. At its core, PPO aims to optimize policy functions in reinforcement learning by striking a balance between stability and sample efficiency. It operates by iteratively updating the policy while ensuring that the updates are relatively conservative to maintain stability and prevent drastic policy changes.

Iterations continue for a specified number of epochs, typically similar to other fine-tuning processes. Through iterative updates, the RL updated LLM aims to generate increasingly aligned responses, as indicated by an improvement in reward scores after each iteration. Iterations continue until reaching a predefined stopping criterion. This could include achieving a specific threshold value for alignment or reaching a maximum number of steps (e.g., 20,000). At convergence, the fine-tuned model is referred to as the **human-aligned LLM**, producing text aligned with specified human preferences (see fig 15-0-14).

The higher the logits in the positive class, the more aligned the completion is with the intended preferences. To obtain the reward value used in RLHF, the largest value of the positive class's logits is extracted. This value serves as the reward signal guiding the LLM's updates and optimizations during the fine-tuning process. Applying a Softmax function to the logits can convert them into probabilities, providing normalized values indicating the likelihood of a completion belonging to each class. However, for RLHF, the focus is primarily on the logits of the positive class to determine the reward value guiding the LLM's learning process.

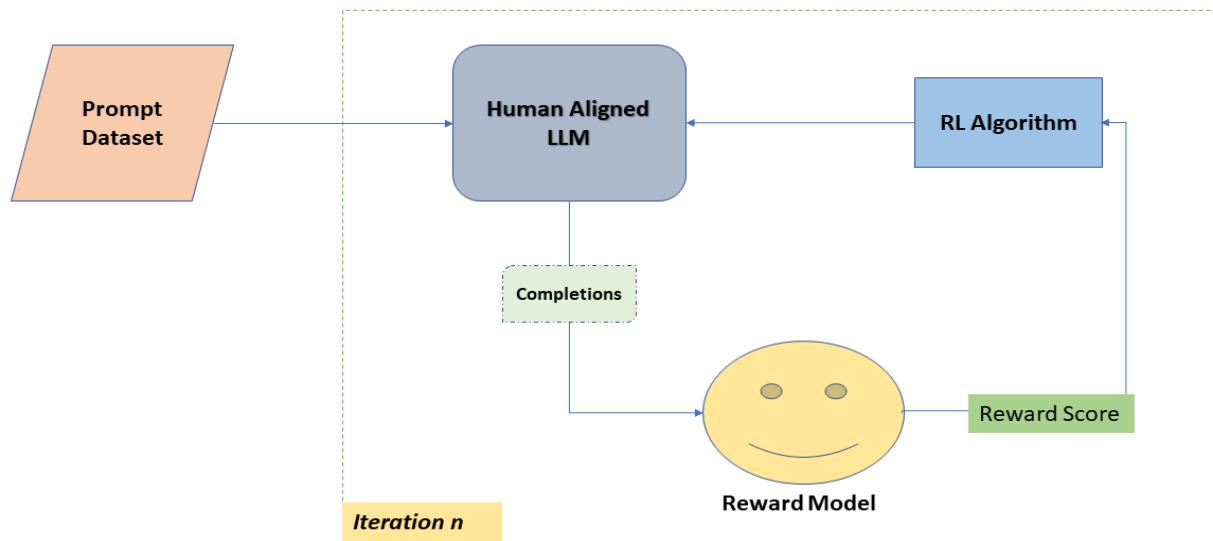


Figure 15-14 RLHF Process leading to Human Aligned model

One of the significant advantages of RLHF is its potential to enhance model safety and relevance. By training LLMs to acknowledge limitations, avoid toxic language, and steer away from sensitive topics, RLHF can help mitigate the risks associated with AI-generated content. It ensures that the model's outputs are more responsible, ethical, and aligned with societal norms.

Moreover, RLHF holds promise for personalizing LLMs based on individual user preferences. Through continuous feedback loops, these models can adapt to each user's unique preferences, potentially leading to personalized learning plans or AI assistants tailored to individual needs and preferences.

However, it is not all that sweet and simple as it has been shown so far. RLHF is usually encountered with a very interesting problem which can be caused by its own agent, which can learn to cheat the system in order to maximize its cumulative reward over time. The phenomenon is called **Reward hacking**. A poorly designed reward function might not capture the true intent of the desired behavior or might be too ambiguous, leading to unintended behaviors. However, it can not be ascribed as the sole culprit, as there can be other unseen factors at play.

Usually an agent aims for gaining reward via interacting with an environment, taking actions based on its current state, and receiving rewards (or penalties) based on the outcomes of those actions. Although with time, it can find some loopholes in the system which can help it to earn rewards by minimizing the loss function without truly learning the crucial aspects of the problem and not actually solving the actual problem. Hallucination is one prime example which is a result of reward hacking. The phenomenon where a Gen AI model generates content that is not accurate or faithful to the original input data or intended output

is called **Hallucination**. The generated content might look very familiar or even preferable to human eyes but the essence of it might not be correct or even intended to solve the intended problem.

15.4.9. LLM use cases with Enterprises

LLMs are more than just conversational chatbots. We can leverage LLMs to carry out smaller, focused tasks like information retrieval. LLMs can be provided a context of enterprise data and user queries can be utilized as prompts to answer questions from the enterprise data. This can free up human customer service representatives to focus on more complex tasks. LLMs could be leveraged to do sentiment analysis over feedback of a product in a much efficient manner than using complex NLP algorithms. The statement does not however symbolize that LLM outputs are perfect. At the time of writing this textbook, LLMs are pretty new and significant amount of development is being done to improve their response. LLMs can be used to create personalized marketing campaigns that are more likely to resonate with customers. They can also be used to generate creative content, such as blog posts, social media posts, and email campaigns. They can also be used to analyse large amounts of data to identify potential risks. They can also be used to generate reports that help enterprises to make better decisions. LLM use cases can vary enterprise to enterprise based on requirements. An area of active development is augmenting LLMs by connecting them to external data sources or using them to invoke external APIs. You can use this ability to provide the model with information it doesn't know from its pre-training and to enable your model to power interactions with the real-world. We have not even discussed the tip of the iceberg!

Caution:

However, while using LLMs with enterprise data, we shall always consider the downsides. We have already discussed about reward hacking in the section before. Any LLM can be infected with it. Remember, LLMs are trained on massive datasets of text and code. If these datasets are biased, the LLMs will also be biased. This can lead to the LLM generating outputs that are discriminatory or inaccurate. Sensitive data like patient health details, intellectual property, etc. should not be fed to LLMs. If this data is not properly protected, it could be exposed to unauthorized users. There is a clear lack of accountability from LLM side. If an LLM generates inaccurate or misleading information, it could lead to financial losses for the enterprise. For example, an LLM that is used to generate financial reports could generate inaccurate numbers, which could lead to the enterprise making bad investment decisions. Hallucination is always something to be cautious about.

Moreover, LLMs do not come cheap, as they are associated with huge computational costs, we already discussed earlier. Companies providing LLMs are still trying to work on feasible pricing models for LLMs. At the time of writing the text, LLMs have a token limit for one prompt request, as the computational resource cost is high. Hence, working with large number of documents or media can be a costly operation with LLMs with slow response time. LLM providers like OpenAI, Google, AWS, etc. are working on improving its efficiency. If an enterprise decides to train its data with LLM, it should use a diverse dataset, which is encrypted, and the results generated from LLMs should be encrypted as well to prevent unauthorized usage. Moreover, LLMs should be monitored for their sentient behavior, in the case it generates or engages in any suspicious activities, it should be shut down immediately.

Multimodality is one major requirement which has not been addressed completely by LLM makers by the time of writing this book. Often there are lot of use cases where multimodal data is required, especially in industries that deal with a mixture of data modalities such as healthcare, robotics, e-commerce, retail, gaming, entertainment, etc. For example, medical predictions require both text (e.g. doctor's notes, patients' questionnaires) and images (e.g. CT, X-ray, MRI scans).

15.4.10. LangChain: Bridge between LLMs and Software Development

LLM providers have made LLMs accessible via REST APIs. After purchasing the required subscription of a LLM provider, be it Azure OpenAI's GPT model, Google's Vertex AI, AWS Bedrock, etc. we are provided with an access key, which is used to make REST request authentication possible for further usage. By prompting an LLM, it is now possible to develop AI applications much faster than ever before. However, an application can require prompting an LLM multiple times and parsing its output, which can be time-consuming and error-prone. Developers would have to write a significant amount of code to make things work.

LangChain makes this easier by providing a framework for developing AI applications that use LLMs. LangChain is a framework, designed by Andrew Chase, to simplify the creation of applications using large language models (LLMs). It abstracts away the details of prompting an LLM and parsing its output, so that developers can focus on the logic of their application. It provides a high-level API that allows developers to easily connect LLMs to other data sources and applications. LangChain also provides a number of features that make it easier to develop applications that are data-aware (connect a LLM to other sources of data) and agentic (allow a LLM to interact with its environment). It helps to reduce the costs of developing AI applications by reducing the amount of custom code that needs to be written.

15.4.11. A Sample Use Case: Document Retrieval and Question Answering

One sample use-case involving Gen-AI LLMs can be to query your enterprise data. Enterprise data can be anything from text files to PDFs to CSVs. LLMs are accessed using Prompt Engineering and LangChain makes this process easy.

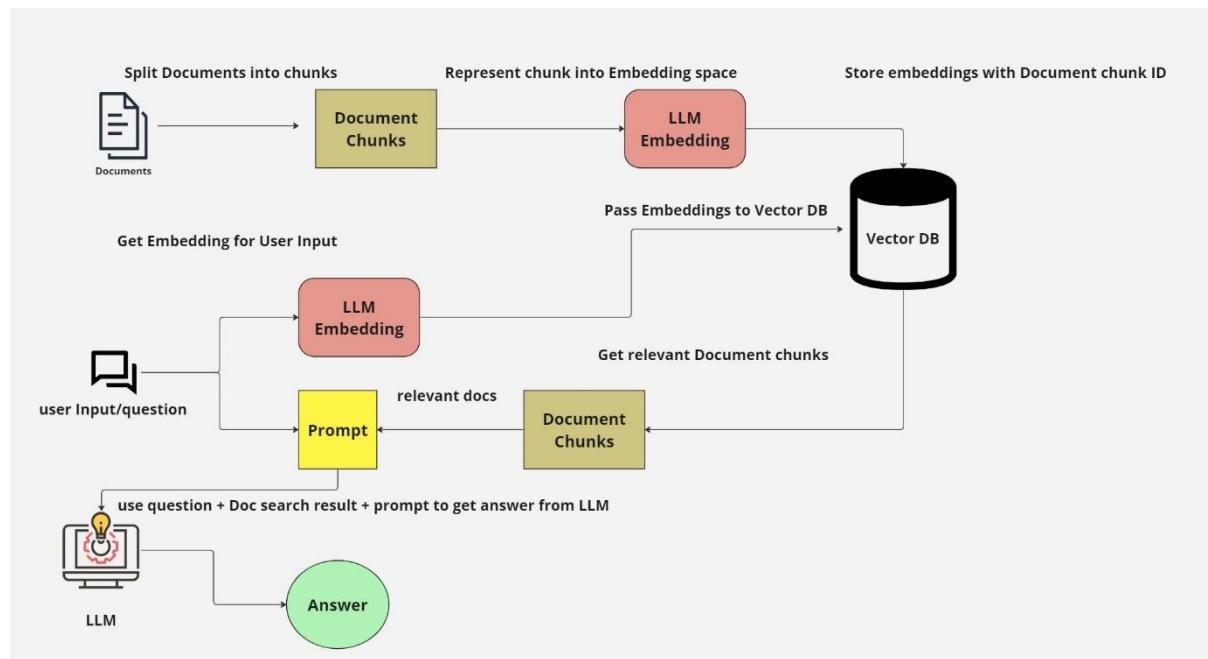


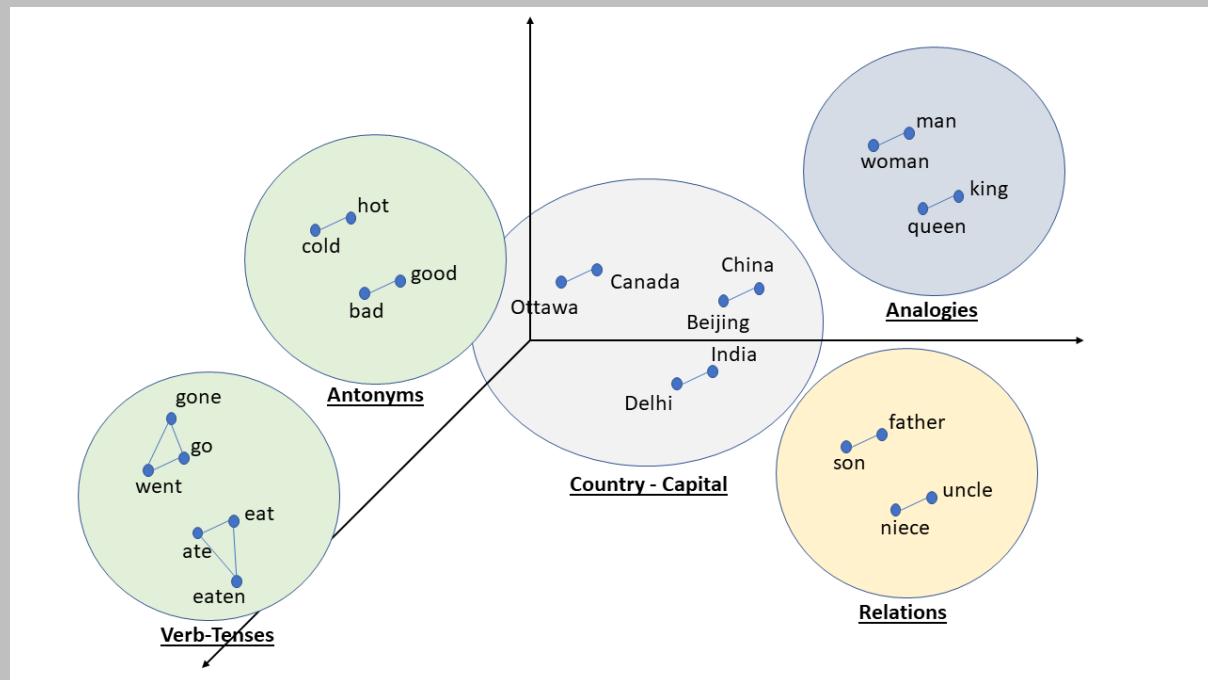
Figure 14-0-15 LLM Document Retrieval and Question Answering

The general approach is to break a set of documents into chunks and then represent these chunks in the embedding space. If we have smaller chunks, embeddings better represent the information in the text. Representing a set of documents in an embedding space, helps in doing similarity search. Embeddings of documents are stored in a **Vector Database (Vector DB)**. It is a special type of database which is optimized for storing and querying high-dimensional vector data (in our case, it is embedding space of documents).

A vector is a mathematical object that has both magnitude and direction. In this context, an **embedding** is a high-dimensional vector that represents a piece of data, such as a word, image, or audio clip. The dimensions of the vector represent different features of the data, such as the meaning of a word, the colours in an image, or the frequencies in an audio clip.

Vector data usually consists of numerical values that represent various features or attributes of objects in a multi-dimensional space. These databases are designed to efficiently handle similarity search and analytics tasks on large sets of vectors and can be much faster than traditional databases for similarity search queries. The task of finding documents that are semantically related to a given query is called **Semantic Search**. Similarity search can be performed over images and audio files which are helpful in building recommender systems.

There can be different embedding models capturing different semantic relationships between entities and these small models can even be grouped together under a Large Language Model. Pre-trained embedding models like Word2Vec, GloVe, FastText, and transformer-based models like BERT or GPT capture semantic relationships between words, phrases, or sentences in a high-dimensional space. BERT and GPT can even incorporate contextual information, understanding how the meaning of a word changes based on its context in a sentence or document. These models are trained on large dataset, learning from vast amounts of text data. This enables them to capture general language patterns and nuances, allowing models to transfer this knowledge to new, smaller datasets or specific tasks with limited labelled data.



Embeddings capture semantic meaning by positioning words, sentences, images, sound, etc. in a continuous vector space where similar semantics correspond to closer vector representations. Various dimensionality reduction techniques like PCA, UMAP, etc. are used to represent high dimensional data to low dimensions, which makes a case for better analysis. Embeddings are stored in Vector DB and Vector DB can be leveraged for fast retrieval of vectors that are similar or closest to a query vector, facilitating efficient similarity searches in massive datasets. Modern Vector databases efficiently handle large volumes of vector data, scaling to accommodate millions or billions of vectors, yet maintaining search performance even as the dataset size increases, which is crucial for real-time applications and systems dealing with massive data.

This aids in building recommendation systems by retrieving similar items or content based on user preferences or item embeddings. It is also useful in semantic search tasks by efficiently retrieving similar documents, sentences, or word embeddings based on their semantic similarity. Clustering and Classification can be easily performed using Vector DB.

User Input, a command or question, will also be converted into embeddings and based upon the similarity search, relevant documents with their chunks will be retrieved from vector DB. Since, we stored our information in smaller chunks, chunks are retrieved from vector DB. Another benefit of smaller chunks is that we can keep the context of LLM smaller, and it leads to better accuracy and lowers the cost. The search result will then be passed to a LLM model along with appropriate prompt for inference and LLM will generate an output.

For our sample use case, I would be presenting example based on Azure platform. Some of the key resources like Azure Blob Storage, Key Vault, services, Azure Open AI studio would be required for full implementation. You would definitely need an Azure Open AI key to access the deployed models. The details for usage and accessibility of these resources is available on Azure webpage.

Disclaimer: The reason for choosing Azure as the platform is solely because my current working organization Eversana has a subscription of Azure portal and it was easier for me to access the resources on Azure. I have been exploring capabilities of Google's Vertex AI and AWS Bedrock parallelly; however, they are just different LLM models. The overall picture for the usage of LLM models remains the same.

Before we leverage LLMs and LangChain, we need to make the data accessible and readable to LLMs and this is where we leverage Azure COGNITIVE Search service. Azure Cognitive Search is a cloud search service that provides developers with infrastructure, APIs, and tools for building a rich search experience over private, heterogeneous content in web, mobile, and enterprise applications.

It has following capabilities:

- A search engine for full text search.
- Indexing with features of lexical analysis and AI enrichment for content extraction and transformation, for e.g. extracting images from PDF texts.
- Query syntax for various search paradigms such as text search, fuzzy search, geo-search, etc.
- Accessibility via REST APIs and SDKs.
- Easy Integration with data layer using Azure Storage Clients such as Blob Storage, etc.

Another alternative to Cognitive Services could have been Vector DBs like Chroma, however, due to computational ability of Azure Cognitive Services, I would stick to this for my use case.

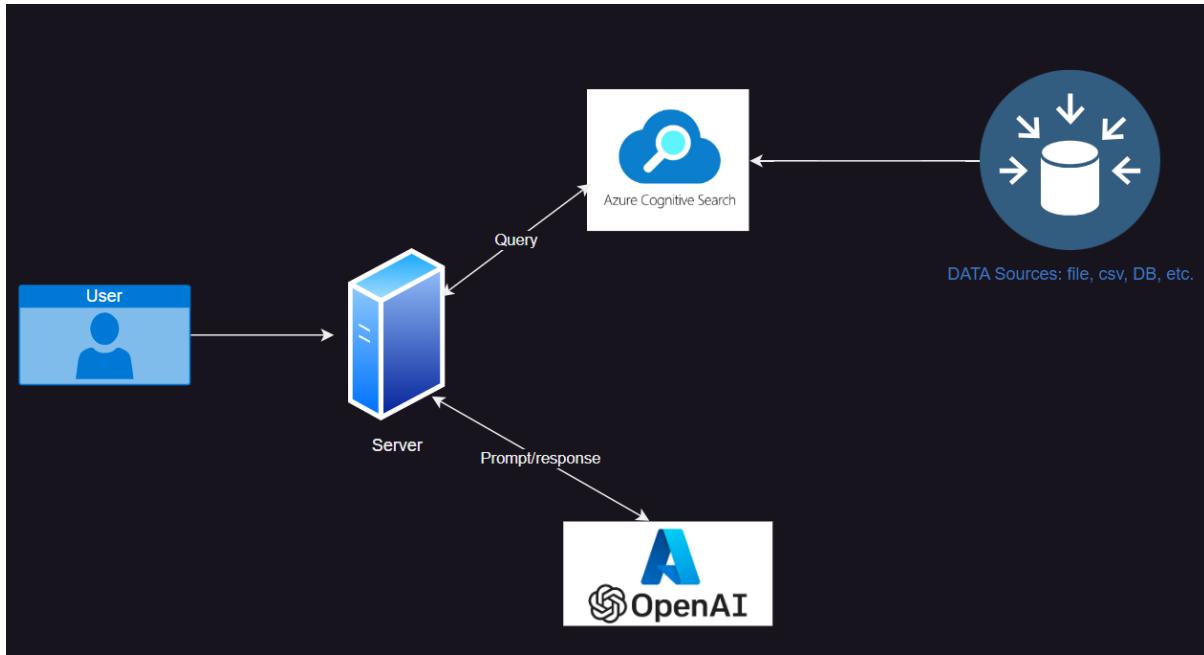


Figure 14-0-16 Document Query architecture model using Azure Open AI

Let us define our use case for the programming:

We have uploaded PDF files into Azure Blob Storage, and we will be leveraging Azure Search Document client (Cognitive Search SDK) in python for querying our data based on user input. Once search results are obtained, we extract relevant information from the result and pass it on to Azure OpenAI to create Prompt and obtain respective response from our models using LangChain.

Let us look at a sample code which leverages Azure Cognitive Search to fetch relevant documents based on user query, from enterprise data. The full code can be found at GitHub: <https://github.com/reeshabh90/PDF-reader/tree/master>

Say, our enterprise data contains lot of text files, like PDF or Excel or Word files, etc. having content about Diabetes.

```

user_input = "What are traits of Type 2 Diabetes??"
# Exclude category, to simulate scenarios where there's a set of docs you can't see
exclude_category = None
search = user_input
print("Searching:", search)
print("-----")
# Filter out documents with a specific category
filter = "category ne '{}'".format(exclude_category.replace("", ""))
# Perform the search using Azure Cognitive Search
r = search_client.search(search,
                         query_type=QueryType.FULL,
                         top=3)
print(r)
# Extract the relevant information from the search results
results = [doc[KB_FIELDS_SOURCEPAGE] + ": " + doc[KB_FIELDS_CONTENT].replace("\n", "").replace("\r", "") for doc in r]

```

Objects, Data & AI: Build thought process to develop Enterprise Applications

```
content = "\n".join(results)
print("*****")
print(content)
```

The above code is quite verbose. It connects and leverages a Cognitive Search Client to fetch relevant documents based on user query and then compiles the output in proper format. The same process can be done in a much simpler way using LangChain. Look at the alternate code:

```
from langchain.retrievers import AzureCognitiveSearchRetriever

retriever = AzureCognitiveSearchRetriever(content_key="content", index_name=AZURE_SEARCH_INDEX, api_key=search_key,
service_name=AZURE_SEARCH_SERVICE)
docs = retriever.get_relevant_documents(user_input)

docs[:5]
```

The abstraction done by LangChain here is self-evident. Once the relevant documents are leveraged from Cognitive Search, they can be fed to Azure Open AI's LLM model by setting the context via prompt. However, if the results obtained from Cognitive Search are large texts, we can leverage basic NLP algorithms to summarize the text and then feed the summarized result to LLM model. However, this operation will result in potential loss of information. One other way could be to leverage LangChain agents and chain interface like *Refine*, which would construct a response by looping over the input documents and iteratively updating its answer or *Map-Reduce*, which would apply an LLM chain to each document individually (the Map step), treating the chain output as a new document and then pass all the new documents to a separate combine documents chain to get a single output (the Reduce step).

Following is the code for a prompt template.

```
template = \
"You are an intelligent assistant helping users with their Diabetes related questions. " + \
"Use 'you' to refer to the individual asking the questions even if they ask with 'I'. " + \
"Answer the following question using only the data provided in the sources below. " + \
"""

### 

Question: 'What is Type 1 Diabetes?'

Sources:
info1.txt: Type 1 diabetes, also known as insulin-dependent diabetes or juvenile-onset diabetes, typically develops in childhood or adolescence.
info2.pdf: Type 1 diabetes occurs when the immune system mistakenly attacks and destroys the insulin-producing beta cells in the pancreas.

Answer:
Type 1 diabetes, also known as insulin-dependent diabetes or juvenile-onset diabetes, typically develops in childhood or adolescence. It occurs when the immune system mistakenly attacks and destroys the insulin-producing beta cells in the pancreas. The exact cause of this autoimmune response is not fully understood, but genetic and environmental factors are thought to play a role.

###
```

Question: '{q}'?

Sources:

{retrieved}

Answer:

"""

Following code uses the prompt template defined above to set the context using the search results from Cognitive search and feeds the content to Azure OpenAI LLM model to get a response and then parses the output.

```
# Generate the prompt for the OpenAI model using the template and retrieved information
prompt = template.format(q=user_input, retrieved=result)

# Call the OpenAI GPT model to get the answer
completion = openai.Completion.create(
    engine= AZURE_OPENAI_GPT_DEPLOYMENT,
    prompt=prompt,
    temperature= 0.3,
    max_tokens=1024,
    n=1,
    stop=[ "\n"])
# Print the answer and additional information
print(completion)
print({ "data_points": results, "answer": completion.choices[0].text, "thoughts": f"Question:<br>{user_input}<br><br>Prompt:<br>" +
    prompt.replace('\n', '<br>')})
```

LangChain provides the building blocks to interface with any language model. Using LangChain, developers can write code to load, transform, store and query data. It provides interfaces to various document loaders, document transformers, text embedding models, vector stores and retrievers. Above code is leveraging Azure Cognitive Search retriever. Most LLM applications have a conversational interface, and an essential component of a conversation is being able to refer to information introduced earlier in the conversation. LangChain provides several utilities like *ConversationBufferMemory*, *EntityMemory*, *KnowledgeGraph memory*, etc. for adding memory to a system, which can be used to store information about past interactions.

Keeping up with the scope of our discussion, we would cut short out conversation on LangChain framework. Further details about this framework can be easily accessed on their official home page.

15.5. Road Ahead

Generative AI is still in the early stages of its lifecycle at the time of writing the text. It has a huge potential to be a game changer for enterprises, but it should be dealt with caution. In coming days, we might be heading towards multimodal capabilities of Gen-AI, where a single model can generate text, images, audio, etc. LLM providers are also working towards more innovative ways of prompt engineering with reduced prompt texts and better capabilities.

Gen- AI-generated content can spread misinformation, fake news, and deceptive information at an unprecedented scale. It becomes essential to ensure that generated content adheres to factual accuracy and doesn't contribute to the dissemination of false information. Ownership and accountability of the generated content is another complex domain where LLM providers would like to improve upon. As the LLMs are trained over vast set of data, it has also led to debates about plagiarism and privacy issues. Maintaining trust and authenticity in the information landscape becomes critical. Enterprises can lose their face and incur heavy losses on a potential downside of LLMs.

Generated contents from Gen-AI models can be used to shape narratives, influence public perception and even affect election results. Hence, its impact on culture and narratives needs a thorough discussion.

There is no iota of doubt that there is going to be a significant change in the landscape of software development in the coming days. We are already witnessing AI code assistants like GitHub CoPilot, GPT code extension, etc. which are able to generate efficient programming code in majority of programming languages like Java, Python, C#, JavaScript, etc. This takes away a lot of pain points from the developer, however, it would be foolish to solely rely on AI models for code generation and integration. They are prone to errors and then the question of accountability of the code is evident. For a bad code, who should be held accountable? AI model? Or the organization or the developer who leveraged these models for code generation.

15.6. End Note

The goal of this book was to compile a bigger picture for a new software enthusiast or student to form an intuition towards developing enterprise applications on a robust scale with sturdy architecture. I would again like to reiterate that I am a mere compiler of the already discovered information, which is already present at different sources like books, online resources, etc. I have tried to present my perspective of Software Development with the experience which I have gathered over 15 years as a student of Computer Science and have been lucky enough to be presented opportunities to work with some esteemed organizations in due course. I bow my head once again to Maa Saraswati (Hindu Deity for knowledge), Ganesh ji and Hanuman ji (Hindu Deity for intelligence and strength) and my Guru Mr Rupam Das for planting the seed to compile my knowledge learnt over the years in the form of documentation for the future students. This book and me would have been incomplete without the love of my beloveds Richa & Maithili!

About the Author



Reeshabh Choudhary is a Software Developer with more than 11 years of experience in designing and developing software applications. He is currently working as a Technical Architect at Eversana India, Pune, Maharashtra.

Reeshabh was born in 1990 at Samastipur, Bihar, where he completed his primary education. He pursued his engineering degree at P.D.A. College of Engineering, Gulbarga (now, Kalburgi) and graduated in 2012. He joined Integrated Ideas Consultancy Services as a trainee developer, where he worked under his mentor, Mr. Rupam Das, for a year. Later, he worked in organizations like Infosys and Schlumberger, before starting a software development venture for a product company called Credivia. However, during the 2019-20 Covid-19 pandemic, the operations at Credivia had to be closed. Since, then he is working at Eversana India at Pune.

Apart from technology, Reeshabh takes a keen interest in Indic History and sports. He represented his college in Cricket and Basketball tournaments. Later, he developed an interest in functional training for muscle rehab after sustaining ACL injury while playing Basketball. He is an active sportsman and gives physical and mental fitness a top priority.



github.com/reeshabh90



<https://twitter.com/reesh0908>



linkedin.com/in/@reeshabh.choudhary