

DIVE INTO SQL



MASTER SQL FROM THE GROUND UP

BY RUBEN

INTRODUCTION

“Dive Into SQL” စာအုပ်ကတော့ ကျွန်တော်ကိုယ်တိုင် SQL ကို လေ့လာနေစဉ်မှာ လက်တွေ့လုပ်ကြည့်ပြီးမှ ခြုံငုံနားလည်လာတဲ့အချက်တွေအပေါ် အခြေခံပြီး ရေးသားထားတာပါ။ မိမိနားလည်သလောက် လက်လှမ်းမီသလောက်ကိုသာ စုစည်းထားနိုင်ပါတယ်။ ဒါပေမယ့် တခြားသိထားသင့်တဲ့ အကြောင်းအရာတွေကိုလည်း ထည့်သွင်းဖော်ပြပေးထားတယ်ဆိုတာ သိစေချင်ပါတယ်။

စာအုပ်တစ်အုပ်ဖြစ်မြောက်အောင် ကြိုးစားမယ်လို့ ဆုံးဖြတ်လိုက်ကတည်းက ကျွန်တော့်ရဲ့စိတ်သဘောထားဟာ “မျှဝေလိုတဲ့စိတ်” တစ်ခုတည်းကိုပဲ ဦးတည်ပြီး ရေးသားဖြစ်ခဲ့တာပါ။ ဒါ့ကြောင့် ဒီစာအုပ်ဟာ တစ်ဖက်သတ် ကိုယ်ကျိုးစီးပွားအတွက် လုံးဝမဟုတ်ရပါ။ မိမိသိရှိလာခဲ့သမျှကို မကြွင်းမချန် မျှဝေချင်တဲ့ စိတ်ထဲကနေ ပေါက်ဖွားလာခဲ့တဲ့ စာအုပ်တစ်အုပ်ဖြစ်ပါတယ်လို့ ယုံကြည်စွာနဲ့ ပြောနိုင်ပါတယ်။

ဒီနေ့ခေတ်မှာ data ဆိုတာ e-commerce platform တွေ (ဥပမာ - Amazon), ဘဏ်လုပ်ငန်းတွေ (ဥပမာ - KBZ Bank), နိုင်ငံရေးဆိုင်ရာအဖွဲ့အစည်းတွေ (ဥပမာ - မဲဆန္ဒကို စုဆောင်းသိမ်းဆည်းတဲ့ Union Election Commission)၊ သွားလာရေးဝန်ဆောင်မှု (ဥပမာ - Grab, Uber) တို့လို အရာတွေမှာ အရေးကြီးအခန်းကဏ္ဍတစ်ခုအဖြစ် ပါဝင်နေပါတယ်။

ဒါကို နည်းပညာအတွေ့အကြုံရှိသူတွေအပါအဝင် တခြားနယ်ပယ်အသီးသီးက သူတွေဟာလည်း နားလည်လက်ခံကြပြီးသားပါ။ အဲ့လိုအခြေအနေတွေကြောင့် data တွေကို စနစ်တကျ သိမ်းဆည်းထားနိုင်ဖို့၊ ထိထိရောက်ရောက် စီမံခန့်ခွဲနိုင်ဖို့၊ လိုအပ်တဲ့အချက်အလက်တွေကို ရှာဖွေထုတ်ယူနိုင်ဖို့နဲ့ analyze နိုင်ဖို့အတွက် query language တစ်ခုခုကို တတ်မြောက်ထားရင် အကောင်းဆုံးပါပဲ။

Query language အများကြီးရှိတဲ့အထဲက လူသုံးအများဆုံးဖြစ်တဲ့ **SQL (Structured Query Language)** ကို အနည်းဆုံး ကျွမ်းကျင်နိုင်နင်းထားတာက သင့်တော်ရာဖြစ်ပါမယ်။

ဒီစာအုပ်ကို မှီငြမ်းပြီး လေ့လာနေသူဆိုရင်လည်း ကျွန်တော့်လို နည်းပညာကို စိတ်ဝင်စားသူ ဖြစ်နေနိုင်တာမို့ စာအုပ်တစ်အုပ်လုံးကို နားလည်လွယ်ရုံသာမက လွယ်ကူရိုးရှင်းပြီး လက်တွေ့အသုံးတည့်အောင် ကြိုးစားရေးသားထားပါတယ်။ ဒါ့အပြင် ပိုပြီးတိတိကျကျဖြစ်အောင် generative AI ရဲ့ အကူအညီလည်း ယူထားပါတယ်။ ဒီအချက်ကိုလည်း အပွင့်လင်းဆုံးအနေနဲ့ ကြိုတင်ပြောထားလိုပါတယ်။

အထူးသဖြင့် ဒီစာအုပ်ရဲ့ အဓိက target တွေကတော့ -

- နည်းပညာနယ်ပယ်ကို စတင်ခြေလှမ်းဖို့ စိတ်ပါဝင်စားသူတွေအတွက်
- SQL programming နဲ့ database အကြောင်း စတင်လေ့လာချင်သူတွေအတွက်

- ကိုယ်ပိုင် project တွေမှာ database တစ်ခု တည်ဆောက်တတ်ချင်သူတွေအတွက်
- Database နဲ့ ပတ်သက်တဲ့ အခြေခံတွေကို သိထားချင်သူတွေအတွက်
- လက်တွေ့ဥပမာတွေကြည့်ပြီး သင်ယူတတ်မြောက်လိုသူတွေအတွက်ပါ။

စာအုပ်အတွင်းမှာတော့ SQL ရဲ့ အခြေခံသဘောတရားတွေကနေစပြီး query ရေးနည်း၊ data တွေကို ဘယ်လိုစနစ်တကျ စီမံရမလဲ၊ database ကို ဘယ်လိုတည်ဆောက်မလဲ ဆိုတဲ့အဆင့်အထိ လက်တွေ့နမူနာ တွေနဲ့ နားလည်ရလွယ်အောင် ရှင်းရှင်းလင်းလင်း ဖော်ပြပေးထားပါတယ်။

Database ကို အထူးပြုပြီး လေ့လာလိုသူတွေအတွက်တော့ ဒီစာအုပ်ဟာ လုံးဝဥသန့်ပြည့်စုံမှာ မဟုတ်ကြောင်း ကြိုပြောထားပါရစေ။ ဒါပေမယ့်လို့ အခြေခံသိထားရမယ့်အချက်တွေနဲ့ database ပိုင်းဆိုင်ရာ ထပ်ဆင့်သိသင့်စရာတွေကို တစ်နည်းတစ်ဖုံရနိုင်မယ့် စာအုပ်တစ်အုပ်ဖြစ်မယ်လို့ ကျွန်တော်ကိုယ်တိုင် မျှော်လင့်ရပါတယ်။

SQL ဟာ နှစ်ပေါင်းများစွာ ကြာရှည်တည်တံ့လာခဲ့တဲ့ နည်းပညာတစ်ခုဖြစ်သလို သူနဲ့ပတ်သက်ပြီး အခြေခံကောင်းကောင်း ရှိထားခြင်းဟာ ထပ်ဆင့်နည်းပညာအသစ်တွေကို လေ့လာရာမှာလည်း အထောက်အကူပြုမှာပဲ ဖြစ်ပါတယ်။

ဆိုတော့ ကျွန်တော်တို့တွေ SQL ထဲကို dive လုပ်လိုက်ကြရအောင်ပါ။

@Ruben
rubenhtun.dev@gmail.com

Copyright Notice

© 2025 Ruben Htun. All rights reserved.

ဤစာအုပ်ကို လွတ်လပ်စွာ ဖတ်ရှုလေ့လာနိုင်ရန်အတွက် အခမဲ့ ဖြန့်ဝေထားပါသည်။ သို့သော် မူရင်းစာရေးသူ၏ ခွင့်ပြုချက်မရှိဘဲ ပြန်လည်ပုံနှိပ်ခြင်း၊ အွန်လိုင်းပေါ်တွင် အခကြေးငွေယူ၍ ဖြန့်ချိခြင်း သို့မဟုတ် မည်သည့်နည်းဖြင့်မဆို စီးပွားဖြစ် အသုံးပြုခြင်းများကို ခွင့်မပြုပါ။

သို့သော် ပညာရေးဆိုင်ရာအတွက် အသုံးပြုလိုသူများအနေဖြင့် မူရင်းစာရေးသူ၏ ခွင့်ပြုချက်ကို တောင်းခံနိုင်ပါသည်။ ဆက်သွယ်ရန် — rubenhtun.dev@gmail.com

TABLE OF CONTENTS

Introduction

Part 1: SQL Fundamentals

Chapter 1: Getting Started with SQL.....	13
Chapter 2: Basic SQL Queries.....	22
Chapter 3: Working with Data Types.....	30

Part 2: Intermediate SQL

Chapter 4: Advanced Filtering and Conditions.....	36
Chapter 5: Joins and Relationships.....	47
Chapter 6: Aggregating Data.....	58

Part 3: Advanced SQL Techniques

Chapter 7: Subqueries and Nested Queries.....	67
Chapter 8: Window Functions.....	76
Chapter 9: Common Table Expressions - CTEs.....	87

Part 4: Database Design and Management

Chapter 10: Database Design Basics.....	95
Chapter 11: Modifying Data.....	107
Chapter 12: Performance Optimization.....	115

Part 5: Practical SQL Applications

Chapter 13: SQL in the Real World.....	124
Chapter 14: Advanced Topics.....	135

Part 6: Wrapping Up

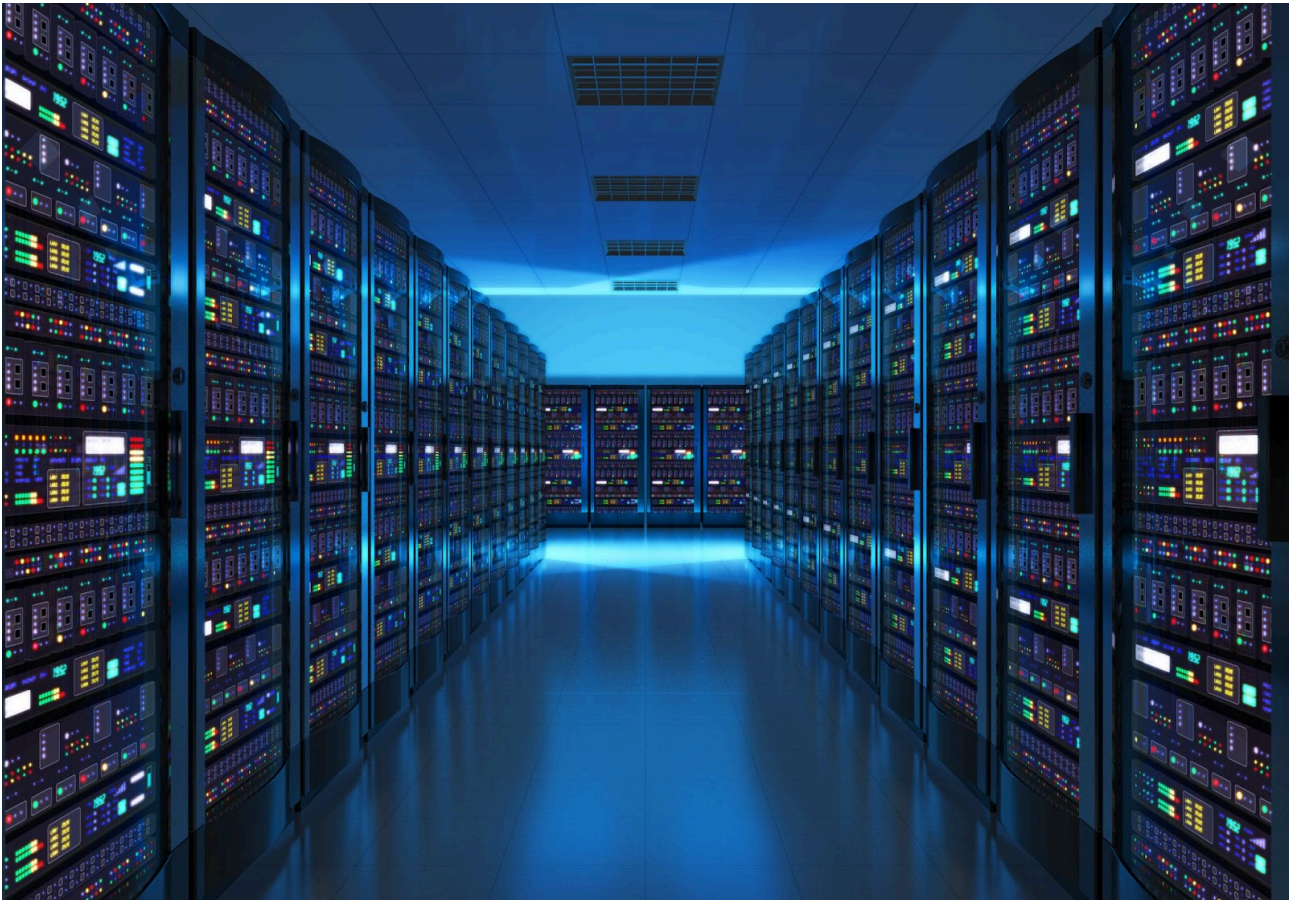
Chapter 15: Best Practices and Next Steps.....	146
--	-----

Conclusion

References

Introduction: What is a database?

Database ဆိုတာ?



Database ဆိုတာ data တွေကို ကျစ်ကျစ်လစ်လစ်ဖြစ်အောင် စနစ်တကျ သိမ်းဆည်းပေးတဲ့ နေရာတစ်ခုလို့ပဲ ပထမဆုံး နားလည်ထားကြရအောင်ပါ။ သူ့ရဲ့ infrastructure အရ data တွေကို လွယ်ကူစွာ စီမံခန့်ခွဲလို့ရအောင်၊ ရှာဖွေနိုင်အောင်နှင့် ပြန်လည်အသုံးပြုနိုင်အောင် ဒီဇိုင်းထုတ်ပြီး တည်ဆောက်ထားတာပါ။

ဥပမာအနေနဲ့ စာအုပ်ဆိုင်တစ်ခုမှာ စာအုပ်တွေရဲ့ နာမည်၊ စာရေးသူအမည်၊ ဈေးနှုန်းနဲ့ stock ရှိမရှိ စတဲ့ အချက်အလက်တွေကို database တစ်ခုထဲမှာ သိမ်းထားလို့ရပါတယ်။ ဒီလို database system တစ်ခုရှိထားတာက ဆိုင်ပိုင်ရှင်အနေနဲ့ စာအုပ်တွေကို လွယ်လွယ်လင့်လင့် ရှာဖွေနိုင်ပါတယ်။ အသစ်ထည့်ချင်တဲ့အခါ ထည့်လို့ရတယ်။ မူဟောင်းတွေ ပြင်ချင်ရင်လည်း အဆင်ပြေအောင် ကူညီပေးနိုင်ပါတယ်။

book_title	author_name	price	in_stock
JavaScript for Beginners	Lin Tun	9500.00	TRUE
CSS Mastery	Nandar Moe	8700.00	FALSE
Python Guidebook	Kyaw Zin	12000.50	TRUE
Database Basics	Ei Mon	10000.00	TRUE

ထပ်ပြီးရှင်းပြလိုတာကတော့ database တွေဟာ tables (ဇယားများ) တွေနဲ့ ဖွဲ့စည်းတည်ဆောက်ထားတာပါ။ ဇယားတစ်ခုမှာတော့ data တွေကို columns (ဒေါင်လိုက်အတန်းများ) နဲ့ rows (အလျားလိုက်အတန်းများ) အဖြစ် စနစ်တကျ စီထားပါတယ်။

ဥပမာ - ကျောင်းတစ်ကျောင်းရဲ့ database ထဲမှာ **students** လို့ခေါ်တဲ့ table တစ်ခု ရှိနိုင်ပြီး ၎င်းတွင် column တွေအဖြစ် **student_name**, **grade**, **birth_date**, **phone_number** တို့ ပါဝင်နိုင်ပါတယ်။ row တစ်ခုချင်းစီကတော့ ကျောင်းသားတစ်ဦးစီအတွက် အချက်အလက်တွေကို ကိုယ်စားပြုပါတယ်။

ဒီလိုဖွဲ့စည်းပုံကြောင့် ကျောင်းသားတွေရဲ့ အချက်အလက်တွေကို စနစ်တကျ စီစဉ်ထားနိုင်ပြီး လိုအပ်တဲ့ အခါမှာ လွယ်လင့်တကူ ရှာဖွေနိုင်ပါတယ်။

student_name	grade	birth_date	phone_number
Ko Nyein Chan	Grade 8	2010-05-22	09451234567
Ma Su Hlaing	Grade 7	2011-08-14	09782345671
Ko Thura	Grade 9	2009-01-10	09212345678

ထိုနည်းတူ ယနေ့ခေတ်မှာလည်း database တွေကို နေ့စဉ်ဘဝအတွင်းမှာ ကျယ်ပြန့်စွာ အသုံးပြုလာကြပြီဖြစ်ပါတယ်။ ဥပမာအနေနဲ့ စာဖတ်သူအနေဖြင့် အွန်လိုင်းဈေးဝယ်တဲ့အခါ၊ ဘဏ်အကောင့်စစ်တဲ့အခါ၊ သို့မဟုတ် ဆိုရှယ်မီဒီယာမှာ post တစ်ခုတင်တဲ့အခါတိုင်းမှာတောင် database တစ်ခုခုဟာ နောက်ကွယ်မှာ အလုပ်လုပ်နေတယ်ဆိုတာကို ရှေးဦးစွာ နားလည်ထားဖို့ အရေးကြီးပါတယ်။

ဒါ့အပြင် mobile apps, websites, airline booking systems တို့ကနေစပြီး ကျန်းမာရေးဆိုင်ရာ electronic health records (EHRs) တို့အထိ ကဏ္ဍအသီးသီးမှာ အချက်အလက်တွေကို သိမ်းဆည်းဖို့နဲ့ ရှာဖွေရယူဖို့ database တွေကိုပဲ အခြေခံအားထားနေရပါတယ်။

What is SQL?

SQL ဆိုတာကော့ ဘာလဲ?



SQL (Structured Query Language) ဆိုတာက database တွေကို စနစ်တကျ စီမံခန့်ခွဲဖို့နဲ့ data တွေကို သိမ်းဆည်းဖို့ အသုံးပြုတဲ့ programming language တစ်ခုပဲ ဖြစ်ပါတယ်။

အထူးသဖြင့် CRUD လုပ်ဆောင်ချက်တွေဖြစ်တဲ့ data တွေကို Create (ဖန်တီးခြင်း)၊ Read (ဖတ်ရှုခြင်း)၊ Update (ပြင်ဆင်ခြင်း)၊ Delete (ဖျက်သိမ်းခြင်း) တို့အတွက် အဓိကအသုံးပြုကြပါတယ်။

SQL ကို အထူးသဖြင့် Relational Database Management Systems (RDBMS) များတွင် တွေ့ရလေ့ရှိပြီး MySQL, PostgreSQL, Oracle, Microsoft SQL Server, SQLite စတဲ့ system တွေမှာ တော်တော်ကျယ်ပြန့်စွာ အသုံးပြုကြပါတယ်။

Database ဆိုတာကတော့ data တွေကို စနစ်တကျ သိမ်းဆည်းထားတဲ့ အချက်အလက် သိုလှောင်ရုံတစ်ခု ဖြစ်သလို ဒီလို data အများကြီးထဲကနေ လိုအပ်တဲ့အချက်အလက်တွေကို ထိထိရောက်ရောက် ရှာဖွေထုတ်ယူနိုင်ဖို့အတွက်တော့ SQL ဟာ အရေးပါတဲ့ query language တစ်ခုပါ။

Relational Database



Relational database တွေဟာ tables တွေဖြင့် ဖွဲ့စည်းထားပြီး ဇယားတစ်ခုနဲ့တစ်ခုအကြား ဆက်နွှယ်မှု (relationship) များ ရှိတတ်ပါတယ်။ SQL ကတော့ အဲဒီဇယားတွေကြား ဆက်သွယ်ဆောင်ရွက်ရာမှာ ပေါင်းကူးတံတားသဖွယ် အရေးကြီးတဲ့အခန်းကဏ္ဍကနေ ပါဝင်နေပါတယ်။

SQL databases တွေမှာတော့ data တွေကို tables, rows, columns များနဲ့ စနစ်တကျ သိမ်းဆည်းထားပါတယ်။ ဒါပေမဲ့ NoSQL databases (ဥပမာ - MongoDB, Cassandra) တွေကတော့ ဒီလို structure မရှိတဲ့ flexible schema များကို အသုံးပြုတာကြောင့် ပုံစံနည်းနည်း မတူကြပါဘူး။ NoSQL ဟာ unstructured data (ဥပမာ - JSON format, key-value pairs) တွေအတွက် ပိုမိုသင့်တော်တဲ့ နည်းလမ်းတစ်ခုပဲ ဖြစ်ပါတယ်။ ဒီအကြောင်းအရာကိုတော့ နောက်ဆုံးအခန်းမှာ ပိုမိုကွဲပြားအောင် ရှင်းရှင်း လင်းလင်းနဲ့ ထပ်မံဖော်ပြထားပါတယ်။

SQL ဟာ database ကို စနစ်တကျ စီမံခန့်ခွဲမှုအတွက် ဆယ်စုနှစ်များစွာကြာကြာ ခိုင်မာစွာအသုံးပြုလာတဲ့ programming language တစ်ခုပဲ ဖြစ်ပါတယ်။ ၁၉၇၄ ခုနှစ်မှာ IBM ရဲ့ သုတေသီတွေက ဗြိတိသျှ ကွန်ပျူတာပညာရှင် Edgar F. Codd ရဲ့ relational database theory ကို အခြေခံပြီး SEQUEL (နောက်မှာ SQL လို့ပြောင်းသုံးလာခဲ့တယ်။) ဆိုတဲ့ နည်းပညာကို စတင်မိတ်ဆက်ခဲ့ပါတယ်။ ပြီးတော့ ၁၉၇၉ ခုနှစ်တုန်းက Oracle ရဲ့ ရှေ့ပြေးကုမ္ပဏီဖြစ်တဲ့ Relational Software Inc. က စီးပွားဖြစ်အနေနဲ့ SQL ကို ပထမဆုံးအသုံးပြုခဲ့တာဖြစ်ပါတယ်။

SQL မှာ အသုံးပြုတဲ့ Database System အလိုက် မူကွဲ (variants) အနည်းငယ်ရှိသော်လည်း အားလုံးကို ANSI (American National Standards Institute) နဲ့ ISO (International Organization for Standardization) မှ သတ်မှတ်ထားတဲ့ စံနှုန်းတွေနဲ့ ကိုက်ညီအောင် ဖန်တီးထားတာ ဖြစ်ပါတယ်။ Database အမျိုးအစားပေါ်မူတည်ပြီး SQL ရဲ့ syntax တွေမှာ နည်းနည်း ကွဲပြားမှုတွေရှိနိုင်သော်လည်း အခြေခံသဘောတရားတွေကတော့ အတူတူပဲ ဖြစ်ပါတယ်။ ဒါကြောင့် SQL ရဲ့ အခြေခံသဘောတရားတွေ ကို တစ်နေရာမှာ သင်ယူလိုက်ရင် အခြား Database System တွေမှာလည်း အလွယ်တကူ အသုံးပြုနိုင်မှာ ဖြစ်ပါတယ်။

ယနေ့ခေတ်မှာ SQL ဟာ ကမ္ဘာပေါ်မှာ အများဆုံးအသုံးပြုနေတဲ့ programming language တွေထဲက တစ်ခုဖြစ်ပြီး အနာဂတ်မှာလည်း ၎င်းရဲ့အရေးပါမှုကို လွယ်လင့်တကူ မပြောင်းလဲသွားနိုင်တော့ဘူးလို့ ယုံကြည်ရပါတယ်။ SQL ဟာ data တွေကို ထိထိရောက်ရောက် စီမံခန့်ခွဲနိုင်တဲ့အတွက် နည်းပညာလောက မှာ အရေးကြီးတဲ့ programming language တစ်ခုအဖြစ် ဆက်လက်ရှိနေမှာ ဖြစ်ပါတယ်။

ဒါကတော့ SQL နဲ့ ပတ်သက်ပြီး သိသင့်စရာတချို့ကို ဖော်ပြချင်းဖြစ်ပါတယ်။ ဆက်လက်ပြီးမှ database အမျိုးအစားတွေကို လေ့လာကြည့်ကြရအောင်ပါ။

Types of databases?

Database တွေကို ၎င်းတို့ရဲ့ ဖွဲ့စည်းပုံ၊ Data သိမ်းဆည်းပုံနှင့် အသုံးပြုပုံအပေါ် မူတည်ပြီး အမျိုးအစား များစွာ ခွဲခြားထားပါတယ်။

1. (Relational Databases)

- **Description:** Data တွေကို tables အနေနဲ့ စုစည်းထားပြီး table တစ်ခုထဲမှာ columns နဲ့ rows တွေ ပါဝင်ပါတယ်။ ဇယားတွေကို primary key နဲ့ foreign key တွေဖြင့် တစ်ခုနဲ့တစ်ခု ဆက်သွယ်ထားတာဖြစ်ပြီး SQL (Structured Query Language) ကို အဓိကအသုံးပြုပါတယ်။
- **Features:**
 - Data ဖွဲ့စည်းပုံက စည်းကမ်းတကျ ဖြစ်ပြီး ယုံကြည်စိတ်ချရပါတယ်။
 - ACID (Atomicity, Consistency, Isolation, Durability) လက္ခဏာတွေကြောင့် data တွေ တည်ငြိမ်မှုရှိတယ်။
- **Examples:** MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server
- **Usage:** ဘဏ္ဍာရေး၊ စီးပွားရေး၊ စာရင်းစီမံခန့်ခွဲမှုနဲ့ data ဖွဲ့စည်းမှု လိုအပ်တဲ့ နယ်ပယ်တွေမှာ အများဆုံး အသုံးပြုကြပါတယ်။

2. (Non-Relational/NoSQL Databases)

- **Description:** NoSQL databases တွေမှာတော့ data ကို traditional table ပုံစံအစား ပို flexible ဖြစ်တဲ့ ဖွဲ့စည်းပုံတွေနဲ့ သိမ်းဆည်းပါတယ်။ ဒီစနစ်က အကြောင်းအမျိုးမျိုးကြောင့် structure မရှိတဲ့ (သို့) format မညီတဲ့ data ပမာဏအများကြီးကို ထိထိရောက်ရောက် စီမံဖို့ အထူးသင့်တော်ပါတယ်။
- **Features:**
 - Data structure ကို လိုအပ်သလို ပြောင်းလဲနိုင်တာကြောင့် ပိုပြီး flexible ဖြစ်ပါတယ်။
 - Data များ တိုးလာတဲ့အခါမှာလည်း အလွယ်တကူ ချဲ့ထွင်နိုင်အောင် ဒီဇိုင်းထုတ်ထားပါတယ်။
 - Data distribution လုပ်ဖို့လိုတဲ့ system တွေမှာတော့ performance ကောင်းကောင်းနဲ့ run နိုင်ပါတယ်။

→ **Types:**

- **Document-Based:** JSON (သို့) BSON format နဲ့ သိမ်းထားပါတယ် (ဥပမာ - MongoDB)။
- **Key-Value:** key နဲ့ value တွေအဖြစ် သိမ်းတယ် (ဥပမာ - Redis, DynamoDB)။
- **Column-Family:** column group အလိုက် သိမ်းတယ် (ဥပမာ - Cassandra)။
- **Graph:** data တွေအကြား ဆက်နွယ်မှုကို အဓိကထားတယ် (ဥပမာ - Neo4j)။

→ **Usage:** Social media apps, IoT systems, big data analysis tools, real-time data processing applications တို့လို နေရာတွေမှာ အသုံးများပါတယ်။

3. (In-Memory Databases)

→ **Description:** ဒီလို databases တွေမှာ data ကို hard disk မှာမထားဘဲ computer ရဲ့ RAM မှာ တိုက်ရိုက်သိမ်းဆည်းပါတယ်။ ဒါကြောင့် data ကို request လုပ်တဲ့အချိန်မှာ တော်တော်မြန်မြန်ဆန်ဆန် ရနိုင်တာပါ။

→ **Features:**

- Data access နဲ့ processing အပိုင်းမှာ တော်တော်မြန်ပါတယ်။
- Data (temporary data) တွေ သိမ်းဖို့ သင့်တော်ပါတယ်။

→ **Examples:** Redis, Memcached, SAP HANA

→ **Usage:** Caching (cache သိမ်းဆည်းဖို့), real-time analytics, gaming systems, advertising platforms စတဲ့ လုပ်ဆောင်ချက်မြန်မြန်လိုတဲ့ နေရာတွေမှာ အသုံးများပါတယ်။

4. (Distributed Databases)

→ **Description:** Distributed database ဆိုတာက data တွေကို တစ်နေရာတည်းမက နေရာအမျိုးမျိုး (သို့) server မျိုးစုံမှာ ဖြန့်ဖြူးထားတဲ့ database ပဲ ဖြစ်ပါတယ်။ ဒီလိုလုပ်ခြင်းက system ကို စကေးချဲ့ထွင်ဖို့လွယ်သလို ယုံကြည်စိတ်ချရမှုလည်း တဖြည်းဖြည်း တိုးလာပါတယ်။

→ **Features:**

- Data ကို နေရာအမျိုးမျိုးမှာ ထပ်မံသိမ်းဆည်းနိုင်တဲ့ replication လုပ်ဆောင်ချက်ရှိပါတယ်။
- Server တစ်ခုမှာ ပြဿနာဖြစ်သွားရင်တောင် data ရရှိနိုင်သေးတယ်။

- **Examples:** Apache Cassandra, MongoDB, Google Spanner
- **Usage:** Worldwide scale ရှိတဲ့ apps, cloud-based systems, data ပမာဏကြီးတဲ့ business systems တွေမှာ Distributed Databases တွေကို သုံးကြပါတယ်။

5. (Graph Databases)

- **Description:** Data ကို nodes နှင့် edges များအဖြစ် သိမ်းဆည်းပြီး data ဆက်နွယ်မှုများကို ခွဲခြမ်းစိတ်ဖြာရန် အထူးဒီဇိုင်းပြုလုပ်ထားတာပါ။
- **Features:**
 - ရှုပ်ထွေးတဲ့ ဆက်နွယ်မှုများကို ထိထိရောက်ရောက် စီမံနိုင်တယ်။
 - Data တွေရဲ့ ဆက်နွယ်မှု (relationships) ကို တော်တော်မြန်မြန်နဲ့ ရယူနိုင်တယ်။
- **Examples:** Neo4j, ArangoDB, OrientDB
- **Usage:** Social networks, recommendation systems, fraud detection, knowledge graphs စတဲ့ ဆက်နွယ်မှုအခြေပြု application များမှာ အသုံးများပါတယ်။

6. (Time-Series Databases)

- **Description:** Time-series database ဆိုတာကတော့ အချိန် (timestamp) ပါတဲ့ data တွေကို သိမ်းဆည်းဖို့နဲ့ ခွဲခြမ်းစိတ်ဖြာဖို့အတွက် အထူးဒီဇိုင်းပြုလုပ်ထားတဲ့ database ဖြစ်ပါတယ်။
- **Features:**
 - Timestamp ပါတဲ့ data တွေကို ထိထိရောက်ရောက် စီမံခန့်ခွဲနိုင်တယ်။
 - Data ချုံ့ရန် (compression) နဲ့ query တွေကို မြန်မြန်ဆန်ဆန် ဆောင်ရွက်နိုင်တယ်။
- **Examples:** InfluxDB, TimescaleDB, Prometheus
- **Usage:** IoT devices, monitoring systems, financial market data, energy management စတဲ့ အချိန်နဲ့ ပတ်သက်တဲ့ data တွေကို သုံးတဲ့ application များမှာ Time-Series DB တွေဟာ အထူးသင့်လျော်ပါတယ်။

7. (Object-Oriented Databases)

- **Description:** ဒီအမျိုးအစား database တွေမှာ data ကို object အဖြစ် သိမ်းဆည်းပါတယ်။ Object-Oriented Programming (OOP) language တွေနဲ့ တိုက်ရိုက်အလုပ်လုပ်လို့ရတာကြောင့် software developer တွေအတွက် အရမ်းအဆင်ပြေပါတယ်။
- **Features:**
 - Data နဲ့ function တွေကို တစ်စုတည်း object အနေနဲ့ သိမ်းထားလို့ရတယ်။
 - ရှုပ်ထွေးတဲ့ data structures တွေကိုလည်း ထိန်းသိမ်းနိုင်တယ်။
- **Examples:** ObjectDB, db4o
- **Usage:** Software development, game creation, complex data models လိုအပ်တဲ့ apps တွေမှာ သုံးပါတယ်။

8. (Cloud-Native Databases)

- **Description:** Cloud environment တွေအတွက် အထူးဒီဇိုင်းပြုလုပ်ထားတဲ့ database တွေဖြစ်ပြီး cloud နဲ့ ချိတ်ဆက်သည်ဖြစ်စေ ချိတ်ဆက်မထားသည်ဖြစ်စေ အလုပ်လုပ်နိုင်စွမ်းရှိပါတယ်။
- **Features:**
 - Auto-scaling feature ပါရှိပြီး maintenance လုပ်ရတာလည်း နည်းပါးပါတယ်။
 - ကမ္ဘာတစ်ဝှမ်းလုံးမှာ data တွေကို ဖြန့်ဝေထားတာကြောင့် availability (ရရှိနိုင်မှု) အမြင့်ဆုံးဖြစ်ပါတယ်။
- **Examples:** Amazon RDS, Google Cloud Spanner, Microsoft Azure Cosmos DB
- **Usage:** Cloud-based apps, mobile apps, DevOps နယ်ပယ်တွေမှာ အထူးသင့်တော်ပါတယ်။

Database အမျိုးအစားတစ်ခုစီမှာ ၎င်း၏ သီးခြားအားသာချက်တွေနှင့် သင့်လျော်တဲ့ အသုံးဝင်မှုတွေရှိပါတယ်။ Relational Database တွေဟာ အစီအစဉ်တကျရှိတဲ့ data တွေအတွက် သင့်လျော်ပြီး NoSQL Database တွေကတော့ စကေးချဲ့ထွင်မှုအတွက် ပြောင်းလွယ်ပြင်လွယ်ဖြစ်တာကြောင့် အကောင်းဆုံးဖြစ်ပါတယ်။ Graph နဲ့ Time-Series Databases တွေကတော့ သီးခြားလိုအပ်ချက်တွေအတွက် အထူးဒီဇိုင်းပြုလုပ်ထားတာတွေပါ။ Database ရွေးချယ်တဲ့အခါမှာတော့ Data ပမာဏ၊ ဖွဲ့စည်းပုံ၊ လိုအပ်တဲ့ စွမ်းဆောင်ရည်နဲ့ အသုံးပြုမယ့် နေရာတွေကို ဦးစားပေးစဉ်းစားဖို့ လိုပါတယ်။

Database အမျိုးအစားတွေနဲ့ ပတ်သက်ပြီး တချို့နားမလည်တဲ့ ဝေါဟာရတွေ၊ အသုံးအနှုန်းတွေ ပါကောင်းပါမှာပါ။ ဒါပေမယ့် အခုအစပိုင်းမှာတော့ အထူးတလည် သိစရာလည်း မလိုအပ်သေးပါ။

Why Should Learn SQL?

SQL ကို ဘာကြောင့်လေ့လာသင့်သလဲ?

လေ့လာသူအများစုဟာ data တွေကို analyze လုပ်တတ်ဖို့ စတင်လေ့လာတဲ့အခါ SQL ကို ပထမဆုံး ရွေးချယ်လေ့ရှိပါဘူး။ များသောအားဖြင့် Microsoft Excel နဲ့ စတင်လေ့လာကြပြီး Excel ရဲ့ အချက်အလက်ဆိုင်ရာ လုပ်ဆောင်ချက်တွေကို အသုံးပြုကြပါတယ်။ Excel ကို အသုံးပြုပြီးတဲ့နောက် Microsoft Office ထဲမှာ ပါဝင်တဲ့ Access လို့ database system တွေဆီ ကူးပြောင်းလေ့ရှိပါတယ်။ Access မှာ data queries တွေကို graphical query interface နဲ့လွယ်ကူစွာ ဖန်တီးနိုင်တာကြောင့် SQL ကို မသိလည်း အလုပ်ဖြစ်နိုင်ပါတယ်။

ဒါပေမယ့် Excel နဲ့ Access တို့မှာ ကန့်သတ်ချက်တွေ ရှိနေပါတယ်။

- Excel မှာ worksheet တစ်ခုအတွက် အများဆုံး 1,048,576 Rows သာထည့်လို့ရပြီး
- Access မှာ database အရွယ်အစား two gigabytes အထိသာ ကန့်သတ်ထားကာ table တစ်ခုလျှင် 16,384 columns ခု အထိသာ ထည့်သွင်းနိုင်ပါတယ်။

အထူးသဖြင့် e-commerce businesses များလို့ ကြီးမားတဲ့ data အချက်အလက်တွေနဲ့ အလုပ်လုပ်ရတဲ့ အခါ ဒီကန့်သတ်ချက်တွေက ပြဿနာဖြစ်စေနိုင်ပါတယ်။ အလုပ်တစ်ခုကို အချိန်မီပြီးအောင် လုပ်နေရင်းနဲ့ သင့်ရဲ့ database system က လိုအပ်တဲ့ပမာဏကို မထောက်ပံ့နိုင်ဘူးဆိုတာ သိရတဲ့အခါ စိန်ခေါ်မှုတစ်ရပ် ဖြစ်လာနိုင်ပါတယ်။

SQL ကတော့ ကြီးမားတဲ့ database system တွေမှာ terabytes အရွယ်အစားရှိတဲ့ data တွေ၊ relational tables အများအပြားနဲ့ columns ထောင်ချီအထိပါဝင်တဲ့ အချက်အလက်တွေကို ထိရောက်စွာ စီမံခန့်ခွဲနိုင်ပါတယ်။ SQL ရဲ့ အားသာချက်ကတော့ data ဖွဲ့စည်းပုံကို programmatically ထိန်းချုပ်နိုင်တာဖြစ်ပြီး ဒါကြောင့် လုပ်ငန်းဆိုင်ရာ လည်ပတ်မှုတွေကို ပိုမိုမြန်ဆန်တိကျစွာ ပြုလုပ်နိုင်ပါတယ်။ အထူးသဖြင့် ရလဒ်တွေကို မှန်ကန်တိကျစွာ ရရှိစေတဲ့ အားသာချက်ကို ပိုင်ဆိုင်ထားပါတယ်။

ထို့အပြင် SQL ကတော့ R နဲ့ Python တို့လို Data Science programming languages တွေနဲ့လည်း လွယ်ကူစွာ ပေါင်းစပ်အသုံးပြုနိုင်တဲ့ language တစ်ခုပါ။ ဒီ programming languages တွေကနေ SQL database တွေကို ချိတ်ဆက်အသုံးပြုနိုင်ရုံသာမကပဲ အချို့အခြေအနေတွေမှာတော့ SQL syntax တွေကို တိုက်ရိုက်ထည့်သွင်းနိုင်ပါသေးတယ်။ Programming မှာ အခြေခံမရှိသူတွေအတွက် SQL ကတော့ data ဖွဲ့စည်းပုံနဲ့ programming logic ရဲ့ အခြေခံအယူအဆတွေကို လွယ်ကူစွာ နားလည်စေတဲ့ အကောင်းဆုံး နည်းလမ်းတစ်ခုလည်း ဖြစ်နေပါတယ်။

SQL ကို ကျွမ်းကျင်ထားမှုက အချက်အလက်စိစစ်နိုင်ခြင်း ဆိုတာထက် ပိုကျယ်ပြန့်တဲ့ အကျိုးကျေးဇူးတွေကို ရရှိစေနိုင်ပါတယ်။ Web framework တွေ တည်ဆောက်တဲ့အခါ interactive maps ဖန်တီးရာမှာ သို့မဟုတ် content management systems တွေမှာ SQL-powered database တွေက နောက်ကွယ်က အချက်အလက်တွေကို ထိထိရောက်ရောက် စီမံခန့်ခွဲပေးဖို့ အလွန်အရေးကြီးတဲ့ အခန်းကဏ္ဍတစ်ခုကနေ ပါဝင်နေပါတယ်။

What You'll Gain from This Book?

ရှေ့မှာလည်း ပြောခဲ့သလို ဒီစာအုပ်ကိုတော့ ကျွန်တော်ကိုယ်တိုင် SQL ကို သင်ယူခဲ့တဲ့ ခရီးတစ်လျှောက် နားလည်လာခဲ့တဲ့ အကြောင်းအရာတွေကို မျှဝေရန် ရည်ရွယ်ပြီး ရေးသားထားတာပါ။ ကျွန်တော်ဟာ SQL ကို အထူးကျွမ်းကျင်တဲ့သူတစ်ယောက် မဟုတ်သေးပါဘူး။ ဒါပေမယ့် ကိုယ်တိုင်လေ့လာခဲ့ချိန်အတွင်း ကျော်ဖြတ်ခဲ့ရတဲ့ အတွေ့အကြုံတွေနဲ့ နားလည်မှုအပေါ် အခြေခံပြီး တခြားသူတွေအတွက်လည်း SQL ကို ပိုမိုလွယ်ကူစွာ သင်ယူနိုင်အောင် ဒီစာအုပ်လေးကို ရေးဖြစ်ခဲ့တာပါ။

စာအုပ်အတွင်းမှာတော့ SQL ရဲ့ အခြေခံအဆင့်တွေနဲ့အတူ ထပ်မံသိထားသင့်တဲ့ အကြောင်းအရာတွေကိုလည်း အတတ်နိုင်ဆုံး ဖော်ပြပေးထားပါတယ်။ ဒီစာအုပ်ကို ဖတ်ရှုပြီးတဲ့အချိန်မှာ စာဖတ်သူတွေကတော့ -

- > SQL ကိုအသုံးပြုပြီး database ထဲက data တွေကို ထုတ်ယူခြင်း၊ analyze ပြုလုပ်နိုင်မယ်။
- > ရှုပ်ထွေးတဲ့ data relationships တွေကို နားလည်ပြီး ထိထိရောက်ရောက် query များရေးနိုင်မယ်။
- > Database design နှင့် performance ကို ပိုမိုကောင်းမွန်အောင် ပြုလုပ်နည်းကို သိရှိလာမယ်။
- > SQL အသုံးပြုပုံဆိုင်ရာ လက်တွေ့ဥပမာများနှင့် အတွေ့အကြုံများကို ရရှိလာမယ်။

ဘယ်ပညာရပ်တွင်မဆို တကယ်စမ်းသပ်ကြည့်ပြီး လုပ်ကြည့်မှသာ သင်ယူလေ့လာမှုကို ပိုမိုထိရောက်စေတဲ့ အတွက် "Trial and error leads to mastery." လို့ အကြံပေးနိုင်ပါတယ်။

How to Use This Book?

ဤစာအုပ်ကို တစ်ခန်းချင်းစီ အစဉ်လိုက်ဖတ်ရှုနိုင်ဖို့ စီစဉ်ထားပေမယ့် စာဖတ်သူက လိုအပ်ချက်အလိုက် သီးခြားအခန်းများကိုလည်း ရွေးချယ်ဖတ်ရှုနိုင်ပါတယ်။ ဤစာအုပ်တွင် -

- > ရိုးရိုးရှင်းရှင်း ရှင်းပြထားတဲ့ SQL သဘောတရားများ။
- > လက်တွေ့ကျတဲ့ code ဥပမာများနှင့် ၎င်းတို့၏ output ရလဒ်များ။
- > စာဖတ်သူတစ်ဦးအနေနဲ့ ကိုယ်တိုင်စမ်းသပ်ကြည့်ဖို့ လေ့ကျင့်ခန်းများ။
- > SQL ကို ပိုမိုထိရောက်စွာ အသုံးပြုနည်းဆိုင်ရာ လမ်းညွှန်ချက်များ၊ အကြံပြုချက်များ။

Chapter(1) မှာတော့ SQL ကို စတင်နိုင်ဖို့အတွက် စာဖတ်သူရဲ့ ကွန်ပျူတာတွင် SQL environment တစ်ခု (ဥပမာ MySQL၊ PostgreSQL) ထည့်သွင်းပြီး စာအုပ်နှင့်အတူပါဝင်တဲ့ နမူနာ database ကို အသုံးပြုပါ။ Chapter တစ်ခုဖတ်ပြီးရင် နမူနာတွေကို ကိုယ်တိုင်လိုက်လုပ်ကြည့်ပြီး လေ့ကျင့်ခန်းမေးခွန်းတွေကိုလည်း စမ်းဖြေကြည့်ဖို့ အကြံပြုပါတယ်။ ဒီလိုဆက်လုပ်သွားရင် SQL ကို လက်တွေ့အခြေခံကနေတစ်ဆင့် တဖြည်းဖြည်း နားလည်သွားမှာဖြစ်ပါတယ်။

Chapter 1: Getting Started with SQL

1. Database Basics

Database ဆိုတာက အချက်အလက်တွေကို စနစ်တကျသိမ်းဆည်းပြီး စီမံခန့်ခွဲဖို့ အသုံးပြုတဲ့ system နည်းပညာတစ်ခုပဲ ဖြစ်ပါတယ်။ ဒီအကြောင်းအရာကိုလည်း ရှေ့မှာအပြည့်အစုံ ရှင်းပြပြီးသားပါ။ ၎င်းတွင် အဓိကအစိတ်အပိုင်းသုံးခုဖြစ်တဲ့ ဇယား (Tables)၊ အတန်း (Rows) နှင့် ကော်လံ (Columns) တို့ ပါဝင်တယ်လို့ ရိုးရှင်းစွာ နားလည်ထားရပါမယ်။

1.1 ဇယား (Tables)

Table ဇယားဟာ database အတွင်း အချက်အလက်တွေကို စနစ်တကျ စုစည်းသိမ်းဆည်းထားနိုင်တဲ့ အခြေခံဖွဲ့စည်းပုံတစ်ခု ဖြစ်ပါတယ်။ ၎င်းကို ရိုးရိုးဇယား (သို့) ကွက်ချထားသော ဇယားကွက်တစ်ခုလို မြင်ယောင်သတ်မှတ်နိုင်ပါတယ်။

ဥပမာအားဖြင့် "ဧည့်သည်မှတ်တမ်း" ဇယားတစ်ခုဖန်တီးပြီး ဟိုတယ်တစ်ခုမှာ လာရောက်တည်းခိုတဲ့ ဧည့်သည်တွေရဲ့ အချက်အလက်တွေ (အမည်၊ ဖုန်းနံပါတ်၊ ရက်စွဲ) ကို စနစ်တကျ စီမံထားသည်ဆိုပါစို့။ Table ဇယားတစ်ခုစီတွင် သီးသန့်အမည်တစ်ခုရှိပြီး ၎င်းအတွင်းရှိ အချက်အလက်များကို ကော်လံ (columns) နှင့် အတန်း (rows) အဖြစ် စီစဉ်ထားသည်ကို လေ့လာကြည့်ရအောင်ပါ။

(ဤဥပမာတွင် ဟိုတယ်ဧည့်သည်များ၏ အချက်အလက်ကို သိမ်းဆည်းထားသော ဇယားနမူနာကို အောက်တွင်ဖော်ပြထားပါတယ်။)

No	Name	Phone	Date	Room
1	U Myint Oo	09785432101	2023-11-01	201
2	Daw Khin San	09223456789	2023-11-02	305
3	U Aung Min	09443322111	2023-11-03	108

1.2 ကော်လံ (Columns)

ကော်လံဆိုတာက ဇယားထဲမှာ ဒေတာတစ်မျိုးမျိုးကို စနစ်တကျ စုစည်းဖော်ပြထားတဲ့ ဒေါင်လိုက်လိုင်းတွေပါ။ ဒါကို “အကွက်” (field) လည်းခေါ်တတ်ကြပါတယ်။ ဥပမာ - “ဧည့်သည်မှတ်တမ်း” ဇယားထဲမှာ

“အမည်”, “ဖုန်း”, “ရောက်ရှိတဲ့နေ့”, “အခန်းနံပါတ်” တို့လို ကော်လံတွေ ပါနိုင်ပါတယ်။ ကော်လံတစ်ခုချင်းစီက ဒေတာတစ်ခုရဲ့ သတ်မှတ်ထားတဲ့ attribute (ဂုဏ်သတ္တိ) တစ်ခုကို ကိုယ်စားပြုပေးပါတယ်။ အဲဒီထဲမှာ ထည့်သွင်းမယ့် ဒေတာတွေကလည်း တူညီတဲ့အမျိုးအစား ဖြစ်ရပါတယ်။ ဥပမာ - စာသား ဖြစ်နိုင်တယ်။ နံပါတ် ဖြစ်နိုင်ပါတယ်။

ဒါကြောင့် တစ်ကော်လံချင်းစီအတွက် column name နဲ့ data type ကို ကြိုတင်သတ်မှတ်ထားသည်ပင် table တစ်ခုကို properly ဒီဇိုင်းဆွဲတာလို့ မှတ်ယူရပါမယ်။

name	age	phone	check-in
U Aung Kyaw	34	09781234567	2024-03-15
Daw Hla Hla	30	09223344556	2024-03-20

ကော်လံတစ်ခုစီမှာတော့ သတ်မှတ်ထားတဲ့ ဒေတာအမျိုးအစားတစ်ခုကိုသာ လက်ခံနိုင်ပါတယ်။ ထို့ကြောင့် 'check-in' ကော်လံတွင် ရက်စွဲပုံစံ (YYYY-MM-DD) များသာ ထည့်နိုင်ပြီး 'phone' ကော်လံတွင် ဂဏန်းတွေပဲ ထည့်လို့ရပါမယ်။ တကယ်လို့ ဒီကော်လံတွေထဲကို မကိုက်ညီတဲ့ data type တစ်ခုကို ထည့်မိရင် error ဖြစ်ပေါ်နိုင်ပါတယ်။ ဒါကြောင့် ဒီလို format နဲ့ rules တွေကို ဇယားတည်ဆောက်တဲ့အချိန်မှာ (table design level မှာ) ကြိုတင်သတ်မှတ်ထားဖို့ အရေးကြီးပါတယ်။ အဲဒါမှ ဒေတာတွေကို တိကျမှန်ကန်စွာ သိမ်းဆည်းနိုင်ပြီး အချက်အလက်များ ရှာဖွေရာတွင် လွယ်ကူမြန်ဆန်စေမှာပါ။

အဲဒီကော်လံတွေဟာ ဇယားထဲမှာရှိတဲ့ data တွေရဲ့ အရည်အသွေးကို ထိန်းသိမ်းရာမှာလည်း အရမ်းအရေးကြီးပါတယ်။ ဥပမာ - 'phone' ကော်လံမှာတော့ နံပါတ်ပဲ ထည့်ဖို့ သတ်မှတ်ထားတာဆိုရင် မှားယွင်းတဲ့ data တွေ မဝင်လာစေဖို့ ကာကွယ်ပေးနိုင်တာပါ။ ဒီလိုစနစ်တကျ စီမံထားတာတွေကြောင့် ကော်လံတွေဟာ database design နဲ့ စီမံခန့်ခွဲရာမှာ မဖြစ်မနေလိုအပ်တဲ့ အပိုင်းတစ်ခု ဖြစ်လာတာပါ။

1.3 အတန်း (Rows)

အတန်းဆိုတာ ဇယားထဲမှာ တစ်ခုချင်းစီဖြစ်တဲ့ data တွေကို စုစည်းထားတဲ့ မှတ်တမ်းတစ်ခုပါ။ “မှတ်တမ်း” (record) လို့လည်း ခေါ်ကြပါတယ်။ ဥပမာ - guests table မှာ အတန်းတစ်ခုစီက ဧည့်သည်တစ်ယောက်ချင်းစီရဲ့ အချက်အလက်တွေကို ဖော်ပြပေးတာပါ။ data တစ်ခုချင်းကိုစုပြီး အတန်းတစ်ခုအဖြစ် ထားတာဖြစ်ပြီး data အသစ်ထည့်လိုက်တာနဲ့ အတန်းအသစ်လည်း ဖန်တီးပေးရပါတယ်။

U Aung Kyaw	34	09781234567	2024-03-15
-------------	----	-------------	------------

ဒီအချက်တွေက database ရဲ့ အခြေခံသဘောတရားတွေပါ။ ဒီစာအုပ်တစ်လျှောက်လုံးမှာတော့ Postgres SQL ကိုသာ အခြေခံပြီး ရေးသားထားတာဖြစ်လို့ Postgres SQL ကို စပြီး install ပြုလုပ်ရမှာဖြစ်ပါတယ်။

2. PostgreSQL Installation

PostgreSQL ကို download လုပ်ပြီး မိမိ OS ထဲမှာ ဘယ်လိုထည့်သွင်းရမယ်ဆိုတာကို ဒီနေရာမှာ အသေးစိတ် မဖော်ပြတွေ့ပါ။

→ Official PostgreSQL Documentation

<https://www.postgresql.org/docs/current/tutorial-install.html> တွင် တစ်ဆင့်ချင်းစီ ဖတ်ရှုပြီး installation လုပ်နိုင်ပါတယ်။

(သို့တည်းမဟုတ်)

→ W3Schools Guide

https://www.w3schools.com/postgresql/postgresql_install.php တွင်လည်း installation လုပ်ပုံအဆင့်ဆင့်ကို ရှင်းလင်းစွာ ဖော်ပြထားပါတယ်။

→ MAC OS Users

MAC Operating System အသုံးပြုသူများအနေဖြင့်တော့ အောက်ပါ tutorial ကိုကြည့်ပြီး install လုပ်နိုင်ပါတယ်။ <https://www.youtube.com/watch?v=wCMXbM5J0X8&t=5s>

3. Setting Up PSQL (Terminal-Based CLI)

Terminal-Based Command-Line Interface (CLI) ကိုသာ အဓိကထား အသုံးပြုသွားမှာ ဖြစ်တဲ့အတွက် PSQL setup လုပ်ပုံကို အောက်ပါ tutorials များမှာ သေချာကြည့်ရှုပြီး ပြင်ဆင်နိုင်ပါတယ်။

→ PSQL & pgAdmin 4 (Windows OS)

<https://www.youtube.com/watch?v=XxyxkmH87nU>

→ How to Setup PSQL (MAC OS)

<https://www.youtube.com/watch?v=feuRPRpy0VU>

မှတ်ချက် - Installation နှင့် setup ပြုလုပ်စဉ်မှာ အခက်အခဲ တစ်စုံတစ်ရာတွေ့တယ်ဆိုရင် သက်ဆိုင်ရာ documentation နှင့် video guides တွေကို အသေးစိတ် ပြန်လည်ကြည့်ရှုစေချင်ပါတယ်။

4. Creating a Database

ဒီအဆင့်အထိ ရောက်လာပြီဆိုရင်တော့ စာဖတ်သူဟာ PostgreSQL ကို SQL shell နှင့်တကွ ချိတ်ဆက်ထားပြီးလောက်ပြီလို့ ယူဆပါတယ်။ Database ဆိုတာ ဒေတာတွေကို စနစ်တကျသိမ်းဆည်းထားတဲ့ နေရာတစ်ခုဖြစ်သည်ဆိုတာလည်း သိထားပြီး ဖြစ်ပါတယ်။ SQL shell terminal မှာ **CREATE DATABASE** ဆိုတဲ့ command ကို ရိုက်ထည့်ပြီး ပထမဆုံးသော database တစ်ခုကို တည်ဆောက်သွားကြ ရအောင်ပါ။

// Sample Syntax

```
CREATE DATABASE database_name;
```

ဥပမာ - ကျောင်းတစ်ခုရဲ့ database တစ်ခု ဖန်တီးလိုပါက အောက်ပါအတိုင်း ရေးသားနိုင်ပါတယ်။

```
CREATE DATABASE school_db;
```

CREATE DATABASE ဆိုတာကတော့ database အသစ်တစ်ခုဖန်တီးဖို့ အသုံးပြုတဲ့ DDL (Data Definition Language) keyword ဖြစ်ပါတယ်။ ဒီဥပမာမှာ **school_db** ကတော့ ဖန်တီးမယ့် database ရဲ့နာမည် ဖြစ်ပြီး ထူးခြားပြီး ရှိရင်းနေရပါမယ်။

ဒီ statement ကို လုပ်ဆောင်ပြီးရင်တော့ စာဖတ်သူဟာ **school_db** အတွင်းမှာ tables တွေ ဖန်တီးပြီး ဖွဲ့စည်းလို့ရပြီ ဖြစ်ပါတယ်။ database ကို နာမည်ပေးတဲ့အခါမှာ စာလုံးအကြီးအသေး (case-sensitive) ရှိနေနိုင်လို့ နာမည်တွေကို တသမတ်တည်း အသုံးပြုတာက ပိုသင့်လျော်ပါတယ်။ ဒါ့အပြင် နာမည်အတွင်းမှာ space သို့မဟုတ် #, @, ! စတဲ့ special characters တွေ မသုံးဘဲ ရှိရင်းတဲ့ အက္ခရာနံပါတ်တွေကိုပဲ သုံးသင့်ပါတယ်။

စောနားက ကျနော်တို့ ဖန်တီးလိုက်တဲ့ database ကို ပြန်စစ်ချင်တယ်ဆိုရင် SQL shell (psql) terminal မှာ **\l** လို့ (backslash နဲ့ စာလုံး L) ရိုက်ထည့်ရင် database list တွေကို ကြည့်ရှုနိုင်ပါတယ်။ ဒါနဲ့ မိမိတည်ဆောက်ထားပြီးတဲ့ database အားလုံးကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။

```
postgres=# CREATE DATABASE school_db;
CREATE DATABASE
postgres=# \l
```

List of databases								
Name	Owner	Encoding	Locale Provider	Collate	Ctype	Locale	ICU Rules	Access privileges
postgres	postgres	UTF8	libc	C	C			
school_db	postgres	UTF8	libc	C	C			
template0	postgres	UTF8	libc	C	C			=c/postgres +
template1	postgres	UTF8	libc	C	C			postgres=CTc/postgres +
								=c/postgres +
								postgres=CTc/postgres

(4 rows)

// Exercise

library_db ဟု အမည်ပေးထားတဲ့ database တစ်ခုဖန်တီးရန် SQL statement တစ်ကြောင်း စာဖတ်သူကိုယ်တိုင် စမ်းရေးသားကြည့်ပါ။

5. Creating Tables

Database တစ်ခုဖန်တီးပြီးနောက်မှာတော့ အဲဒီ database အတွင်းမှာ data တွေကို စနစ်တကျ သိမ်းဆည်းဖို့ table တွေဆောက်ဖို့ လိုလာပါပြီ။ table တစ်ခုဆိုတာ column တွေ row တွေနဲ့ ဖွဲ့စည်းထားတာ ဖြစ်တဲ့အတွက်ကြောင့် ဒီတစ်ခါမှာတော့ **CREATE TABLE** ဆိုတဲ့ SQL statement ကို သုံးပြီး **school_db** ဆိုတဲ့ database ထဲမှာ table တစ်ခု ဖန်တီးကြည့်မှာ ဖြစ်ပါတယ်။ အခုက table တစ်ခုဆောက်မှာဖြစ်တဲ့အတွက် မဆောက်ခင်မှာ SQL shell တွင် **\c school_db** လို့ရိုက်ပြီး တည်ဆောက်ပြီးခဲ့တဲ့ **school_db** database နဲ့ချိတ်ဆက်ရမှာဖြစ်ပါတယ်။

```
postgres=# \c school_db
You are now connected to database "school_db" as user "postgres".
school_db=#
```

ပြီးမှ အောက် syntax နမူနာအတိုင်း table တစ်ခု ဆောက်ရပါမယ်။

// Sample Syntax

```
CREATE TABLE table_name (
    column_name1 data_type,
    column_name2 data_type,
    column_name3 data_type
);
```

ဥပမာအနေနဲ့ **school_db** database ထဲမှာ ကျောင်းသားတွေရဲ့ သက်ဆိုင်ရာအချက်အလက်တွေကို သိမ်းဆည်းဖို့ **students** ဆိုတဲ့ table တစ်ခု ဖန်တီးမှာပါ။ အဲဒီ table ထဲမှာ columns အနေနဲ့ **student_id**, **name** နဲ့ **birthdate** တို့ ပါဝင်မှာ ဖြစ်ပါတယ်။

```
CREATE TABLE students (
    student_id INT,
    name VARCHAR(100),
    birthdate DATE
);
```

ပထမတစ်ခုက **student_id** ဖြစ်ပြီး ဒါဟာ ကျောင်းသားတစ်ဦးချင်းစီအတွက် ထူးခြားတဲ့ ID နံပါတ်ကို သိမ်းဆည်းမယ့် column ဖြစ်ပါတယ်။ ဒါကြောင့် **INT** (ကိန်းဂဏန်း) data type ကို သုံးထားပါတယ်။ နောက်ထပ် column က **name** ဖြစ်ပြီး ဒီ column ထဲမှာတော့ ကျောင်းသားအမည်ကို သိမ်းမှာဖြစ်ပါတယ်။ အများဆုံးစာလုံးရေ ၁၀၀ လုံးအထိ ရိုက်လို့ရအောင် **VARCHAR(100)** data type ကို သုံးထားပါတယ်။ နောက်ဆုံးမှာတော့ **birthdate** ဆိုတဲ့ column ပါလာပါတယ်။ ဒီမှာတော့ မွေးသက္ကရာဇ်ကို သိမ်းဖို့ **DATE** type ကို သုံးပါတယ်။ အရေးကြီးတာတစ်ခုကတော့ column တစ်ခုနဲ့တစ်ခုကြားမှာ comma (,) ခြားထားဖို့ လိုပါတယ်။

ထပ်ဂရုပြုသင့်တာက column အမည်များကိုလည်း ရှင်းလင်းပြီး အဓိပ္ပာယ်ရှိအောင် ပေးသင့်ပါတယ်။ **student_id**, **name** စသဖြင့် သတ်မှတ်လိုက်တာက ပိုမိုထူးခြားပြီး အသုံးချရလွယ်ကူပါတယ်။

Data အမျိုးအစားကိုလည်း သေချာစွာရွေးချယ်သင့်ပါတယ်။ နာမည်အတွက် **VARCHAR** ကို သုံးရခြင်းက မျှော်မှန်းထားတဲ့ စာလုံးအရေအတွက်အတိုင်း သိမ်းဆည်းနိုင်ပေးပြီး ပိုမိုထိရောက်စေပါတယ်။

students ဆိုတဲ့ table ကို ပြန်ကြည့်ချင်ရင် terminal တွင် **\d students** လို့ ရိုက်ပြီး ကြည့်နိုင်ပါတယ်။ အဲဒီမှာတော့ column သုံးခုနဲ့ သူတို့ရဲ့ datatype တွေကိုတွေ့ရမှာ ဖြစ်ပါတယ်။

```
school_db=# \d students
Table "public.students"
  Column | Type          | Collation | Nullable | Default
-----+-----+-----+-----+-----
 student_id | integer       |           |          |
  name      | character varying(100) |           |          |
 birthdate  | date          |           |          |
school_db=#
```


// Exercise

books ဆိုတဲ့ table တစ်ခုကို ကိုယ်တိုင်ထပ်ဖန်တီးကြည့်ပါ။ ၎င်းမှာတော့ **book_id** (INT), **title** (VARCHAR(100)) နဲ့ **publication_year** (INT) ဆိုတဲ့ columns တွေ ပါဝင်ရပါမယ်။

students ဆိုတဲ့ table မှာ နောက်ထပ် column အသစ်တစ်ခု ထပ်ထည့်ကြည့်ရအောင်ပါ။

// Sample Syntax

```
ALTER TABLE table_name ADD column_name datatype;
```

ALTER နဲ့ **ADD** ကိုသုံးပြီး ရှိပြီးသား **students** table တွင် **grade** ဆိုတဲ့ column အသစ်တစ်ခု ထည့်လိုက်တာပါ။ ဒီလို table structure ကို ပြန်ပြင်ချင်တဲ့အခါ **ALTER** command ကို သုံးပါတယ်။

```
ALTER TABLE students ADD grade VARCHAR(50);
```

Table ကို ပြန် check ကြည့်ရင် **grade** ဆိုတဲ့ column အသစ်တစ်ခု ရောက်နေတာ တွေ့ရမှာပဲဖြစ်ပါတယ်။

```
[school_db=# \d students
Table "public.students"
  Column      |          Type          | Collation | Nullable | Default
-----+-----+-----+-----+-----
 student_id   | integer                |           |          |
  name        | character varying(100) |           |          |
 birthdate    | date                   |           |          |
  grade       | character varying(50)  |           |          |
school_db=#
```

SQL အကြောင်းကို အနည်းငယ်နားလည်လာပြီဖြစ်တဲ့အတွက် ဒီတစ်ကြိမ်မှာ **students** table ထဲကို ကျောင်းသား ၅ ဦးရဲ့ သက်ဆိုင်ရာ data records တွေကို **INSERT** keyword နဲ့ တပြိုင်တည်း ထည့်ကြည့်ကြမှာပါ။

```
INSERT INTO students (student_id, name, birth_date, grade)
VALUES
  (1, 'Alice Johnson', '2008-05-14', '8th Grade'),
  (2, 'Brian Smith', '2007-11-22', '9th Grade'),
  (3, 'Cindy Lee', '2009-03-03', '7th Grade'),
  (4, 'David Kim', '2008-08-30', '8th Grade'),
  (5, 'Eva Martinez', '2006-12-12', '10th Grade');
```

INSERT INTO students ဆိုတာက **students** အမည်ရှိတဲ့ table ထဲသို့ data အသစ်ထည့်ရန် ညွှန်ကြားပေးလိုက်တာပါ။ Data values တွေကိုတော့ (**student_id**, **name**, **birth_date**, **grade**) column အမျိုးအစားတွေအလိုက် သူ့နေရာနဲ့သူ ထည့်သွားမယ်လို့ ပြောတာပါ။

```
school_db=# INSERT INTO students (student_id, name, birthdate, grade)
school_db=# VALUES
school_db=# (1, 'Alice Johnson', '2008-05-14', '8th Grade'),
school_db=# (2, 'Brian Smith', '2007-11-22', '9th Grade'),
school_db=# (3, 'Cindy Lee', '2009-03-03', '7th Grade'),
school_db=# (4, 'David Kim', '2008-08-30', '8th Grade'),
school_db=# (5, 'Eva Martinez', '2006-12-12', '10th Grade');
INSERT 0 5
```

// Exercise

ရှေ့မှာ တည်ဆောက်ခိုင်းခဲ့တဲ့ **books** ဆိုတဲ့ table ထဲမှာ ဒီတန်ဖိုးတွေသုံးပြီး လေ့ကျင့်ကြည့်ပါ။

```
INSERT INTO books (book_id, title, publication_year)
VALUES
(1, 'To Kill a Mockingbird', 1960),
(2, 'The Lord of the Rings', 1954),
(3, 'Pride and Prejudice', 1813),
(4, 'The Great Gatsby', 1925),
(5, 'The Catcher in the Rye', 1951);
```

// Top 10 PostgreSQL Commands for Chapter One

1. CREATE DATABASE database_name;

- Database အသစ်တစ်ခုဖန်တီးရန်။

2. \c database_name

- psql ထဲတွင် database တစ်ခုသို့ ပြောင်းချိတ်ဆက်ရန်။

3. DROP DATABASE database_name;

- တည်ဆောက်ထားတဲ့ database တစ်ခုလုံးဖျက်ရန်။

4. \l

- Server ထဲရှိ database များအားလုံးစာရင်းကို ပြသရန်။

5. CREATE TABLE table_name (column1 datatype, column2 datatype);

- Database ထဲတွင် table အသစ်ဖန်တီးရန်။

6. ALTER TABLE table_name ADD column_name datatype;

- Table တွင် column အသစ်ထည့်ရန်။

7. DROP TABLE table_name;

- Table တစ်ခုဖျက်ရန်။

8. \dt

- လက်ရှိ database ထဲရှိ table များအားလုံးကို ပြရန်။

9. \d table_name

- Table ထဲရှိ (column များ၊ datatype များ) ပြရန်။

10. SELECT current_database();

- လက်ရှိအသုံးပြုနေသော database နာမည်ပြရန်။

Chapter 2: Basic SQL Queries

1. SELECT Statements: Retrieving Data

SELECT query statement ကတော့ SQL မှာ database ထဲက data တွေကို ပြန်ယူဖို့ အသုံးပြုတာပါ။ ဒီ statement က table တစ်ခုထဲက မိမိလိုတဲ့အချက်အလက်တွေကို ရွေးချယ်ပြီး ပြပေးတာပဲ ဖြစ်ပါတယ်။ ဥပမာအနေနဲ့ table ထဲက ကျောင်းသားတွေရဲ့ အမည်တွေကို ရယူချင်တယ်ဆိုရင် **SELECT** ကို အသုံးပြုနိုင်ပါတယ်။

```
SELECT name FROM students;
```

ဒီဥပမာမှာတော့ **students** table ထဲက **name** column ကို ရွေးချယ်ပြီး ကျောင်းသားတွေရဲ့ အမည်အားလုံးကို ပြပေးမှာ ဖြစ်ပါတယ်။

```
[school_db=# SELECT name FROM students;
 name
-----
Alice Johnson
Brian Smith
Cindy Lee
David Kim
Eva Martinez
(5 rows)
```

1.1 Selecting Specific Columns

Table တစ်ခုတွင် column များစွာ ရှိနေမှာဖြစ်သော်လည်း မိမိလိုအပ်တဲ့ column များကိုသာ ရွေးချယ်ရမယ့် အချိန်အခါဆိုတာ ရှိလာမှာပဲဖြစ်ပါတယ်။ အဲ့တာကြောင့် data ရယူတဲ့အခါမှာ ပိုမိုထိရောက်ပြီး မလိုအပ်တဲ့ အချက်အလက်တွေကိုလည်း ချန်ထားခဲ့နိုင်ပါတယ်။

```
SELECT student_id, name FROM students;
```

ဒီ query statement ကတော့ **students** table ထဲက **student_id** နဲ့ **name** column တွေကိုသာ ရယူပေးမှာ ဖြစ်ပါတယ်။ **birthdate** နဲ့ **grade** column တွေကိုတော့ ဘယ်နည်းနဲ့မှ ထုတ်ပြမှာမဟုတ်ပါဘူး။

```
[school_db=# SELECT student_id, name FROM students;
 student_id |      name
-----+-----
          1 | Alice Johnson
          2 | Brian Smith
          3 | Cindy Lee
          4 | David Kim
          5 | Eva Martinez
(5 rows)
```

1.2 Retrieving All Columns with *

* ဆိုတာက table ထဲရှိ column အားလုံးကို ရွေးချယ်ချင်တဲ့အခါမှာ အသုံးပြုနိုင်ပါတယ်။ တစ်ကြိမ်တည်းနဲ့ column အားလုံးရဲ့ data ကို ရယူချင်တဲ့အချိန်မှာတော့ အရမ်းအသုံးဝင်ပါတယ်။ ဒါပေမယ့် မလိုအပ်တဲ့ data တွေလည်း ပါလာနိုင်တာမို့ သတိထားပြီးသာ အသုံးပြုသင့်ပါတယ်။

```
SELECT * FROM students;
```

ဒီ query statement ကတော့ **students** table ထဲက **student_id**, **name**, **birthdate** နဲ့ **grade** column အားလုံးကို ပြသပေးမှာ ဖြစ်ပါတယ်။

```
[school_db=# SELECT * FROM students;
 student_id |      name      | birthdate |      grade
-----+-----+-----+-----
          1 | Alice Johnson | 2008-05-14 | 8th Grade
          2 | Brian Smith   | 2007-11-22 | 9th Grade
          3 | Cindy Lee     | 2009-03-03 | 7th Grade
          4 | David Kim     | 2008-08-30 | 8th Grade
          5 | Eva Martinez  | 2006-12-12 | 10th Grade
(5 rows)
```

2. Filtering Data with WHERE

WHERE keyword ကတော့ အထူးလိုအပ်ချက်အရ data တွေကို စစ်ထုတ်ချင်တဲ့အခါ အသုံးဝင်ပါတယ်။ မိမိလိုချင်တဲ့ specific data တွေကို သီးသန့် ရွေးထုတ်ဖော်ပြဖို့အတွက်ပါ။

```
SELECT name FROM students WHERE student_id = 1;
```

student_id တန်ဖိုး 1 ရှိတဲ့ ကျောင်းသား Alice Johnson ကိုသာ ရွေးယူပြီး ပြမှာဖြစ်ပါတယ်။

```
[school_db=# SELECT name FROM students WHERE student_id = 1;
      name
-----
Alice Johnson
(1 row)

school_db=#
```

2.1 Comparison Operators (=, >, <, etc.)

Comparison operators (နှိုင်းယှဉ်အော်ပရေတာတွေ) ကို **WHERE** clause ထဲမှာ အသုံးပြုပြီး data တွေကို သတ်မှတ်ထားတဲ့ စံနှုန်းတွေနဲ့ နှိုင်းယှဉ်ဖို့ အသုံးပြုပါတယ်။ အဓိက အသုံးပြုတဲ့ operators တွေကတော့

- > = (တူညီသည်)
- > > (ကြီးသည်)
- > < (ငယ်သည်)
- > >= (ကြီးသည် သို့မဟုတ် တူညီသည်)
- > <= (ငယ်သည် သို့မဟုတ် တူညီသည်)
- > <> , != (မတူညီပါ)

```
SELECT name FROM students WHERE birthdate < '2008-01-01';
```

ဒီ query statement ကတော့ ၂၀၀၈ ခုနှစ် ဇန်နဝါရီ ၁ ရက် မတိုင်ခင် မွေးဖွားတဲ့သူတွေရဲ့ နာမည်တွေကို ရယူပေးမှာ ဖြစ်ပါတယ်။

```
[school_db=# SELECT name FROM students WHERE birthdate < '2008-01-01';
      name
-----
Brian Smith
Eva Martinez
(2 rows)

school_db=#
```


2.2 Combining Conditions

AND, **OR**, နဲ့ **NOT** အော်ပရေတာတွေကို **WHERE** clause ထဲမှာ conditions တွေ ပေါင်းစပ်ဖို့ အသုံးပြုနိုင်ပါတယ်။

- **AND:** condition နှစ်ခုလုံး မှန်ကန်ဖို့လိုပါတယ်။
- **OR:** condition တခုခု မှန်ရင်ရပါတယ်။
- **NOT:** သတ်မှတ်ထားတဲ့ condition ကို ဆန့်ကျင်ဘက်အဖြစ် ပြောင်းလဲဖို့ အသုံးပြုပါတယ်။

```
SELECT name FROM students WHERE birthdate > '2000-01-01' AND student_id > 2;
```

(First condition) 2000 ခုနှစ်နောက်ပိုင်း မွေးဖွားသူ ဖြစ်ရပါမယ်။ (Second condition) **student_id** က 2 ထက်ကြီးတဲ့ ကျောင်းသားတွေရဲ့ နာမည်တွေကို ပြောပြရပါမယ်။

```
[school_db=# SELECT name FROM students WHERE birthdate > '2000-01-01' AND student_id > 2;
 name
-----
Cindy Lee
David Kim
Eva Martinez
(3 rows)
```

3. Sorting Results with ORDER BY

ORDER BY keyword ကတော့ data record တွေကို သတ်မှတ်ထားတဲ့ column အလိုက် အစဉ်လိုက်စီဖို့ အသုံးပြုပါတယ်။

```
SELECT name FROM students ORDER BY name;
```

ဒီနည်းနဲ့ ကျောင်းသားတွေရဲ့ နာမည်တွေကို အက္ခရာစဉ်အတိုင်းစီပြီး ပြသပေးမှာ ဖြစ်ပါတယ်။

```
[school_db=# SELECT name FROM students ORDER BY name;
 name
-----
Alice Johnson
Brian Smith
Cindy Lee
David Kim
Eva Martinez
(5 rows)
```

3.1 Ascending vs. Descending Order

ORDER BY မှာ အစီအစဉ် သတ်မှတ်ဖို့ **ASC** (ငယ်စဉ်ကြီးလိုက်) နဲ့ **DESC** (ကြီးစဉ်ငယ်လိုက်) ဆိုတဲ့ keywords နှစ်ခုကို အသုံးပြုပါတယ်။ အဲဒီ keywords တွေ သတ်မှတ်မထားရင်တော့ ပုံမှန်အားဖြင့် **ASC** (ငယ်စဉ်ကြီးလိုက်) အတိုင်းစီပြီး ပြသပေးမှာ ဖြစ်ပါတယ်။

```
SELECT name FROM students ORDER BY birthdate DESC;
```

ကျောင်းသားတွေကို မွေးသက္ကရာဇ်အလိုက် နောက်ပိုင်းမွေးသူတွေကနေ အစောပိုင်းမွေးသူတွေကို အစဉ်လိုက် ပြသပေးမှာ ဖြစ်ပါတယ်။

```
[school_db=# SELECT name FROM students ORDER BY birthdate DESC;
 name
-----
Cindy Lee
David Kim
Alice Johnson
Brian Smith
Eva Martinez
(5 rows)
```

3.2 Sorting by Multiple Columns

ORDER BY မှာ column တွေအများကြီးကို အဆင့်လိုက် ထပ်မံစီနိုင်ပါသေးတယ်။ ဥပမာ - ပထမ column နဲ့ ascending အတိုင်းစီတဲ့အခါ တူညီတဲ့တန်ဖိုးတွေရှိရင် ဒုတိယ column နဲ့ ဆက်လက်စီနိုင်ပါတယ်။

```
SELECT name, birthdate FROM students ORDER BY birthdate ASC, name ASC;
```

ဒီ query က **birthdate** ကို **ASC** (အစောပိုင်းမှနောက်ပိုင်းသို့) အတိုင်း စီပေးမှာဖြစ်ပြီး မွေးသက္ကရာဇ် တူတဲ့ ကျောင်းသားတွေရှိရင် **name** တွေကို **ASC** (အက္ခရာစဉ်) အတိုင်း ထပ်မံစီပေးမှာ ဖြစ်ပါတယ်။

```
[school_db=# SELECT name, birthdate FROM students ORDER BY birthdate ASC, name ASC;
 name | birthdate
-----+-----
Eva Martinez | 2006-12-12
Brian Smith | 2007-11-22
Alice Johnson | 2008-05-14
David Kim | 2008-08-30
Cindy Lee | 2009-03-03
Daniel Park | 2009-03-03
(6 rows)
```

4. Limiting Results with LIMIT

LIMIT ကတော့ data records အရေအတွက်ကို ကန့်သတ်ချင်တဲ့အခါ အသုံးပြုတဲ့ keyword ပါ။ Data များများပါတဲ့ table တွေမှာ အထူးအသုံးဝင်ပါတယ်။

```
SELECT name FROM students LIMIT 3;
```

students table ထဲက ထိပ်ဆုံးကျောင်းသား ၃ ဦးရဲ့ အမည်တွေကိုသာ ပြပေးမှာ ဖြစ်ပါတယ်။

```
school_db=# SELECT name FROM students LIMIT 3;
          name
-----
Alice Johnson
Brian Smith
Cindy Lee
(3 rows)

school_db=#
```

4.1 Use Cases for Limiting Output

LIMIT ကို အောက်ပါအခြေအနေတွေမှာ အသုံးပြုနိုင်ပါတယ်။

- စမ်းသပ်ခြင်း (Testing): data အနည်းငယ်သာ ကြည့်ချင်တဲ့အခါ။
- စာမျက်နှာခွဲခြင်း (Pagination): website တစ်ခုမှာ data တွေကို စာမျက်နှာအလိုက် ပြသချင်တဲ့အခါ။
- ထိရောက်မှု (Efficiency): မလိုအပ်တဲ့ data တွေကို မဆွဲထုတ်ဖို့။

ဥပမာ - Website တစ်ခုမှာ ကျောင်းသား ၁၀ ဦးစီကို စာမျက်နှာတစ်ခုစီမှာ database ကနေလှမ်းယူပြီး ပြချင်ရင်တော့ **LIMIT** ကို အသုံးပြုနိုင်ပါတယ်။

4.2 OFFSET for Pagination

OFFSET ကို **LIMIT** နဲ့အတူသုံးပြီး ရလာမယ့် results ထဲမှာ ဘယ် row နံပါတ်ကို စတင်ပြမလဲဆိုတာ သတ်မှတ်နိုင်ပါတယ်။ ဒီဟာက Website တွေမှာ စာမျက်နှာခွဲခြင်း (pagination) အတွက် အရမ်းအသုံးဝင်ပါတယ်။

```
SELECT name FROM students ORDER BY name LIMIT 3 OFFSET 3;
```

LIMIT 3 ဆိုတာက row သုံးခုတည်းကို ရယူမယ်လို့ ဆိုတာပါ။ **OFFSET 3** ဆိုတာကတော့ ပထမ row သုံးခုကို မဆွဲထုတ်ဘဲ skip လုပ်ခိုင်းတာပါ။ **ORDER BY name** ပါလာလို့ နာမည်တွေကို အက္ခရာစဉ် အတိုင်းစီပြီး ကျောင်းသားအားလုံးထဲက (ပထမ ၃ ဦးကို ကျော်ပြီး နောက်ထပ် ၃ ဦး) ကို ဆွဲထုတ်ပေးမှာပဲ ဖြစ်ပါတယ်။

```
[school_db=# SELECT name FROM students ORDER BY name LIMIT 3 OFFSET 3;
      name
-----
Daniel Park
David Kim
Eva Martinez
(3 rows)

school_db=#
```

ORDER BY name ကို မထည့်ပဲ ရေးမယ်ဆိုရင်။

```
SELECT name FROM students LIMIT 3 OFFSET 3;
```

```
[school_db=# SELECT name FROM students LIMIT 3 OFFSET 3;
      name
-----
David Kim
Eva Martinez
Daniel Park
(3 rows)

school_db=#
```

ဒီအခန်းမှာတော့ **SELECT**, **WHERE**, **ORDER BY**, **LIMIT**, နဲ့ **OFFSET** တို့လို clauses တွေကို အသုံးပြုပြီး အခြေခံ SQL query များစွာကို လေ့လာခဲ့ပါတယ်။ အဲဒီ clause တွေက database ထဲက data တွေကို ရယူဖို့၊ စစ်ထုတ်ဖို့၊ စီမံ နဲ့ ကန့်သတ်ဖို့အတွက် အရေးကြီးတဲ့ အခြေခံသဘောတရားတွေပါ။

အထက်ပါ clause တွေကို ကျွမ်းကျင်စွာ အသုံးပြုတတ်လာရင် fundamental database management အတွက် အထိုက်အလျောက်ပိုင်နိုင်ပြီလို့ ယူဆနိုင်ပါတယ်။

5. Exercise

အပေါ်က SQL clauses တွေသုံးပြီး ကိုယ်တိုင်လေ့ကျင့်ကြည့်ပါ။

// Create the products table

```
CREATE TABLE products (  
    id INT,  
    product_name VARCHAR(100),  
    category VARCHAR(50),  
    price INT,  
    stock INT  
);
```

// Insert sample data

```
INSERT INTO products (id, product_name, category, price, stock)  
VALUES  
    (1, 'Book', 'Stationery', 5000, 100),  
    (2, 'Pen', 'Stationery', 2000, 200),  
    (3, 'Electric Lamp', 'Electronics', 15000, 50),  
    (4, 'Phone', 'Electronics', 300000, 20),  
    (5, 'Notebook', 'Stationery', 3000, 150),  
    (6, 'Refrigerator', 'Electronics', 500000, 10);
```

Question 1:

Write an SQL query to retrieve the product names and prices of items in the Stationery category with a price greater than **2500**. Show only **2 rows**.

Question 2:

Write an SQL query to retrieve the product names, prices, and categories, sorted by price in **descending order** (highest to lowest). Skip the first **2 products** and show the next **3**.

Question 3:

Write an SQL query to retrieve the product names, categories, and prices of products that are either in the Electronics category or have a stock less than **50**. Sort the results by price in **ascending order** and show only **4 products**.

Chapter 3: Working with Data Types

1. Common SQL Data Types

SQL မှာ table တစ်ခုထဲက column တွေမှာ သိမ်းဖို့ data တွေရှိပါတယ်။ အဲဒီ data တွေကို ဘယ် format နဲ့ သိမ်းမလဲဆိုတာကို data type သတ်မှတ်ပေးရပါတယ်။ မတူညီတဲ့ data type တွေကို သုံးတာကလည်း မတူညီတဲ့ရည်ရွယ်ချက်တွေ လုပ်ဆောင်နိုင်ဖို့အတွက်ပါ။ အသုံးများတဲ့ data type တွေကတော့ -

- **Numeric:** ကိန်းဂဏန်းများအတွက် (ဥပမာ - INT, DECIMAL)။
- **Text:** စာသား သို့မဟုတ် အက္ခရာများအတွက် (ဥပမာ - VARCHAR, CHAR)။
- **Date and Time:** ရက်စွဲ သို့မဟုတ် တိကျတဲ့အချိန်အတွက် (ဥပမာ - DATE, TIMESTAMP)။

SQL တွင် data type များကို မှန်ကန်စွာ ရွေးချယ်အသုံးပြုမှသာ data သိမ်းဆည်းမှုနဲ့ ရှာဖွေမှုများကို ပိုမိုထိရောက်စွာ ဆောင်ရွက်နိုင်မယ်ဆိုတာကိုလည်း ရှေ့အခန်းတွေမှာ ဖော်ပြပြီးသား ဖြစ်ပါတယ်။

1.1 Numeric (INT, DECIMAL)

SQL တွင် ဂဏန်းဆိုင်ရာ အချက်အလက်များကို သိမ်းဆည်းဖို့အတွက် အဓိကအသုံးများတဲ့ data type တွေက **INT** နဲ့ **DECIMAL** တို့ပဲ ဖြစ်ပါတယ်။ **INT** (integer) ဟာ ကိန်းပြည့်များကိုသာ သိမ်းဆည်းနိုင်တဲ့ data type ဖြစ်ပြီး ပုံမှန်အားဖြင့် 4 bytes အရွယ်အစားရှိပါတယ်။ ဒါကြောင့် -2,147,483,648 မှ +2,147,483,647 အထိ ကိန်းပြည့်တန်ဖိုးများကို သိမ်းဆည်းနိုင်ပါတယ်။ ကုန်ပစ္စည်းအရေအတွက်၊ အသုံးပြုသူ ID နံပါတ်၊ အသက်အရွယ်ကဲ့သို့သော ကိန်းပြည့်တန်ဖိုးများအတွက် **INT** ဟာ အထူးသင့်တော်ပါတယ်။

DECIMAL data type ကတော့ ဒဿမကိန်းတွေကို တိတိကျကျ သိမ်းချင်တဲ့အခါ အသုံးများပါတယ်။ ငွေကြေးဆိုင်ရာတွက်ချက်မှုတွေ၊ သိပ္ပံဆိုင်ရာတွက်ချက်မှုတွေလို တိကျမှုလိုအပ်တဲ့အရာတွေမှာ အထူးသင့်လျော်ပါတယ်။ **DECIMAL(p, s)** ပုံစံနဲ့ သတ်မှတ်လို့ရပြီး **p** က စုစုပေါင်း digit အရေအတွက် (precision) ကို ဆိုလိုပြီး **s** က ဒဿမနောက်က digit အရေအတွက် (scale) ဖြစ်ပါတယ်။ ဥပမာ **DECIMAL(5, 2)** လို့ရေးမယ်ဆိုရင် 123.45 ဆိုတဲ့တန်ဖိုးကို သိမ်းလို့ရပါတယ်။

ဒါ့အပြင် **FLOAT** နဲ့ **REAL** တို့နဲ့မတူဘဲ **DECIMAL** က တိကျတဲ့တန်ဖိုးကိုသာ သိမ်းပေးပါတယ်။ ဒါကြောင့် ငွေကြေး၊ ဘဏ္ဍာရေးဆိုင်ရာ app တွေမှာတော့ **DECIMAL** ကို အသုံးပြုတာ ပိုအဆင်ပြေတတ်ပါတယ်။


```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    name VARCHAR(100),
    position VARCHAR(50),
    salary DECIMAL(10,2)
);
```

employees ဆိုတဲ့ table မှာ -

- **employee_id** ကို **PRIMARY KEY** အဖြစ် သတ်မှတ်လိုက်တာက **employee_id** များဟာ တစ်ခုနှင့်တစ်ခု လုံးဝတူညီစရာအကြောင်း မရှိတော့ပါ။ ပြီးတော့ ၎င်း column တွင် တန်ဖိုးမဲ့ (NULL) မဖြစ်ရပါ။
- **name** ကတော့ **VARCHAR(100)** ဆိုတဲ့ text data type ဖြစ်ပြီး ဝန်ထမ်းအမည်ကို သိမ်းဆည်းဖို့ပါ။ အများဆုံးစာလုံးရေ 100 အထိ သိမ်းနိုင်ပါတယ်။
- **position** ကလည်း **VARCHAR(50)** ဖြစ်ပြီး ဝန်ထမ်းရဲ့ရာထူးအမည်ကို သိမ်းဆည်းဖို့ပါ။ အများဆုံးစာလုံးရေ 50 အထိ သိမ်းနိုင်ပါတယ်။
- **salary** ဟာ **DECIMAL(10,2)** ဖြစ်တာကြောင့် စုစုပေါင်း ဂဏန်း 10 လုံးအထိ၊ ဒသမနောက် 2 လုံးအထိ တိကျစွာ သိမ်းဆည်းနိုင်ပါတယ်။ ဥပမာ - 650000.00 လိုမျိုးတန်ဖိုးတွေကို သိမ်းနိုင်ပါတယ်။

1.2 Text (VARCHAR, CHAR)

VARCHAR နဲ့ **CHAR** တို့ဟာ database ထဲမှာ text စာသားတွေ သိမ်းဆည်းဖို့ အသုံးပြုတဲ့ data type နှစ်မျိုးဖြစ်ပါတယ်။ ဒါပေမဲ့ သူတို့ရဲ့ သိမ်းဆည်းပုံနဲ့ memory အသုံးပြုပုံမှာတော့ ကွဲပြားမှုရှိပါတယ်။

VARCHAR (Variable Character) ဆိုတာက သတ်မှတ်ထားတဲ့ စာလုံးအရေအတွက်အတွင်းမှာ စာသားရဲ့ တကယ်ရှိတဲ့အရှည် (actual length) ကိုသာ memory တွင် နေရာယူတာဖြစ်ပါတယ်။ ဥပမာ - **VARCHAR(100)** ဟာ အကွာရာ 10 လုံးပါတဲ့စာသားကို သိမ်းမယ်ဆိုရင် အဲဒီ 10 လုံးအတွက်သာ memory storage ကို အသုံးပြုမှာဖြစ်ပြီး ကျန်တဲ့စာလုံးရေ 90 စာအတွက်တော့ memory ထဲမှာ မသုံးဘဲ ချန်ထားပါလိမ့်မယ်။

အဲဒီအကြောင်းကြောင့် စာလုံးအရေအတွက်မတူတဲ့ text data တွေကို သိမ်းချင်တဲ့အခါ **VARCHAR** ဟာ ပိုမိုထိရောက်ပြီး memory ပိုမိုချွေတာနိုင်တဲ့ data type တစ်ခုဖြစ်ပါတယ်။

အခြားတစ်ဖက်မှာတော့ **CHAR** (Character) က သတ်မှတ်ထားတဲ့ actual length အတိုင်း နေရာအပြည့် ယူပြီး ကျန်တဲ့နေရာတွေကို space ဖြင့် ဖြည့်ပေးပါတယ်။ ဥပမာ - **CHAR(10)** မှာ "Hello" ဆိုတဲ့ စာသားကို သိမ်းမယ်ဆိုရင် အကွာရာ 5 လုံးစာအတွက်သာ သိမ်းဆည်းပေးပြီး ကျန်တဲ့ 5 နေရာကို space နဲ့ဖြည့်ပေးကာ စုစုပေါင်း 10 နေရာလုံးကို အသုံးပြုသွားမှာပါ။ **VARCHAR** က နေရာလွတ်ထားနိုင်သော်လည်း **CHAR** ဟာ fixed-length data တိကျတဲ့ နံပါတ်အရေအတွက်ရှိတဲ့ (ဥပမာ - ID နံပါတ်တွေ၊ hash codes) တွေအတွက် ပိုပြီးအသုံးပြုလေ့ရှိပါတယ်။

ထပ်မံပြောပါမယ်။ **VARCHAR** ဟာ ပုံသေအတိအကျမရှိတဲ့ စာသားများ (ဥပမာ - အမည်၊ လိပ်စာ) အတွက် သင့်တော်ပြီး **CHAR** က ပုံသေအတိအကျရှိတဲ့ data (ဥပမာ - ဖုန်းနံပါတ်၊ fixed-format code) များအတွက် ပိုမိုသင့်တော်ပါတယ်။ ထို့ကြောင့် database ဒီဇိုင်းဆွဲရာမှာ စာသားရဲ့ သဘောသဘာဝနဲ့ လိုအပ်ချက်တွေကို လေ့လာပြီး မှန်ကန်စွာ ရွေးချယ်ပေးဖို့ အရေးကြီးပါတယ်။

```
ALTER TABLE employees ADD email VARCHAR(100);
ALTER TABLE employees ADD gender CHAR(1);
```

email လိပ်စာတွေက text length မတူညီတာကြောင့် **VARCHAR(100)** နဲ့ သိမ်းဆည်းပြီး gender က M သို့မဟုတ် F စတဲ့ character တစ်လုံးသာ သိမ်းဖို့လိုအပ်တဲ့အတွက် **CHAR(1)** ကို အသုံးပြုနိုင်ပါတယ်။ တချို့ database တွေမှာတော့ gender ကို 1 သို့မဟုတ် 0 နဲ့ သိမ်းဆည်းတာကိုလည်း တွေ့ရတတ်ပါတယ်။

1.3 Date and Time (DATE, TIMESTAMP)

→ DATE:

- ရက်စွဲကိုပဲ သိမ်းဆည်းပေးပါတယ် (YYYY-MM-DD Format)။
- ဥပမာ - '2003-04-28' (ကျောင်းသား၏မွေးနေ့)။
- Memory 3 bytes သာ နေရာယူပါတယ်။
- အသုံးပြုသင့်တဲ့နေရာများ - မွေးနေ့များ၊ ရုံးပိတ်ရက်များ၊ အားလပ်ရက်များ။

→ TIMESTAMP:

- ရက်စွဲနှင့်အချိန်နှစ်ခုလုံး သိမ်းဆည်းပေးပါတယ်။ (YYYY-MM-DD HH:MM:SS Format)။
- ဥပမာ - '2025-04-28 14:30:00' (Data ပြောင်းလဲမှုအချိန်)။
- Time zone ကိုပါ ထည့်သွင်းစဉ်းစားနိုင်ပါတယ်။
- 4 bytes (သို့) 8 bytes နေရာယူပါတယ်။ Database အမျိုးအစားပေါ်မူတည်ပါတယ်။

- အသုံးပြုသင့်တဲ့နေရာများ - ငွေလွှဲမှုမှတ်တမ်းများ၊ login time၊ submit မှတ်တမ်းများ၊ စနစ်ပြောင်းလဲမှုမှတ်တမ်းများ စတဲ့ အချိန်အတိအကျ သိမ်းဆည်းဖို့ လိုအပ်တဲ့အခါ။

```
ALTER TABLE employees ADD contract_end_date DATE;
ALTER TABLE employees ADD last_updated TIMESTAMP;
```

Key considerations when choosing

1. **DATE** ကို အချိန်အပိုင်းအခြား အတိအကျ မလိုအပ်တဲ့အခြေအနေများမှာ အသုံးပြုသင့်ပါတယ်။
2. **TIMESTAMP** ကို အဖြစ်အပျက်တစ်ခုရဲ့ တိကျတဲ့အချိန်ကို မှတ်တမ်းတင်ဖို့ အသုံးပြုပါတယ်။
3. **TIMESTAMP** ဟာ Time zone ကိုပါ ထည့်သွင်းစဉ်းစားနိုင်တဲ့အတွက် နိုင်ငံတကာအသုံးပြုမှုတွေမှာ ပိုမိုသင့်တော်ပါတယ်။

2. Choosing the Right Data Type

Data အမျိုးအစား ရွေးချယ်ခြင်းက database ရဲ့ performance နဲ့ accuracy အတွက် အရေးကြီးတာမို့လို့ **company_db** ရဲ့ **employees** table ထဲမှာ **employee_id** ကို **INT** နဲ့ သိမ်းထားပြီး **name** နဲ့ **position** အတွက် **VARCHAR()** ကို သုံးခဲ့ပါတယ်။ **salary** ကတော့ **DECIMAL(10,2)** နဲ့ **contract_end_date** ကိုတော့ **DATE** နဲ့ သေချာရွေးချယ် သတ်မှတ်ခဲ့တာဖြစ်ပါတယ်။

2.1 Impact on Storage and Performance

Data type ရွေးချယ်ခြင်းက data storage နဲ့ database ရဲ့ performance ပေါ်မှာ အကျိုးသက်ရောက်မှုရှိ တယ်ဆိုတာက ဒီလိုပါ။

- **INT** - Storage နည်းပြီး calculation တွေအတွက် မြန်မြန်ဆန်ဆန် အလုပ်လုပ်နိုင်ပါတယ်။
- **VARCHAR** - Text length ပမာဏအပေါ်မူတည်ပြီး storage လိုအပ်ချက် ပြောင်းလဲနိုင်ပါတယ်။
- **DECIMAL** - Precision မြင့်တာကြောင့် တိကျမှုကောင်းပေမဲ့ storage ပိုမိုလိုအပ်တတ်ပါတယ်။

ဥပမာ - **employees** table ထဲမှာ **name** အတွက် **VARCHAR(1000)** လောက်သုံးမယ်ဆိုရင် index size ပိုကြီးလာပြီး memory ပိုစားမှာဖြစ်ပါတယ်။ အဲ့ဒါကို **VARCHAR(100)** နဲ့ပဲ သုံးလိုက်မယ်ဆိုရင် storage သက်သာသလို data access လုပ်တာလည်း မြန်ဆန်လာနိုင်ပါတယ်။

3. Type Casting and Conversion

SQL မှာ data type တစ်မျိုးကို တခြားတစ်မျိုးပြောင်းချင်ရင် **CAST()** နဲ့ **CONVERT()** ဆိုတဲ့ function တွေကို အသုံးပြုနိုင်ပါတယ်။ Function ဖြစ်တဲ့အတွက် ပြောင်းချင်တဲ့တန်ဖိုးကို () ထဲထည့်ရေးရပါတယ်။ ဒီ function တွေက အထူးသဖြင့် original data type က မလိုအပ်တဲ့ format ဖြစ်နေတယ် ဆိုတဲ့အခါ အသုံးဝင်ပါတယ်။ ကိုယ့်လိုချင်တဲ့ data type ကို ယာယီပြောင်းလို့ရတဲ့အတွက် flexible ဖြစ်ပါတယ်။

```
SELECT name, CAST(salary AS VARCHAR(20)) AS salary_text FROM employees;
```

ဤ query မှာ **CAST(salary AS VARCHAR(20))** ဆိုတာက **salary** column မှာရှိတဲ့ **DECIMAL(10,2)** data type ကို **VARCHAR(20)** (text format) ပြောင်းလဲပြထားတာပါ။ ပြောင်းပြီးတဲ့ data ကို **salary_text** ဆိုတဲ့ ယာယီ column နာမည်နဲ့သတ်မှတ်ထားတာဖြစ်ပါတယ်။

CAST query ကို run မလုပ်ခင်မှာ ပထမဆုံး **employees** table ထဲကို အချို့ employee records တွေ ထည့်ထားဖို့လိုပါတယ်။ ပြီးရင်တော့ SQL shell (psql) မှာ အောက်ပါပုံစံအတိုင်း query တစ်ခုကို run ကြည့်နိုင်ပါတယ်။

```
company_db=# SELECT name, CAST(salary AS VARCHAR(20)) AS salary_text FROM employees;
 name      | salary_text
-----+-----
 Alice Smith | 5000.00
 Bob Johnson | 7200.50
(2 rows)

company_db=#
```

salary_text column က **employees** တွေရဲ့ **salary** ကို text ပုံစံဖြင့် ပြထားတာကို တွေ့ရမှာပါ။

3.1 Using CAST and CONVERT

CAST က SQL standard function တစ်ခုဖြစ်ပြီး data type ပြောင်းလဲဖို့ အသုံးပြုပါတယ်။ **CONVERT** ကတော့ SQL server စတဲ့ database တချို့မှာ သုံးတဲ့ function ဖြစ်ပြီး **CAST** နဲ့ လုပ်ဆောင်ချက် သဘောတရားချင်း တူညီပါတယ်။ ဒါပေမဲ့ syntax ပုံစံ အနည်းငယ်ကွာခြားချက် ရှိနိုင်ပါတယ်။

```
SELECT CONVERT(VARCHAR, contract_end_date) AS date_string
FROM employees;
```

contract_end_date column ထဲက **date** data type ကို **VARCHAR** ပုံစံသို့ ပြောင်းပြီး **date_string** ဆိုတဲ့ alias (ယာယီအမည်) နဲ့ ပြသပေးမှာဖြစ်ပါတယ်။ ဒါပေမဲ့ ဒီ query statement ဟာ PostgreSQL မှာတော့ အလုပ်လုပ်မှာ မဟုတ်ပါဘူး။ **'YYYY-MM-DD'** ကို သုံးရင်တော့ အဆင်ပြေပါတယ်။

```
[company_db=# SELECT CONVERT(VARCHAR, contract_end_date) AS date_string FROM employees;
ERROR: column "varchar" does not exist
LINE 1: SELECT CONVERT(VARCHAR, contract_end_date) AS date_string FR...
                        ^
```

ဘာလို့လဲဆိုတော့ **CONVERT(VARCHAR, ...)** ဆိုတဲ့ syntax က SQL server မှာပဲ အသုံးပြုလို့ရတဲ့ format ဖြစ်ပါတယ်။ PostgreSQL မှာတော့ ဒီ syntax ကို မသုံးနိုင်ပါဘူး။

3.2 Handling Data Type Mismatches

Data type မတူညီမှု (ဥပမာ - text ကို int နဲ့ နှိုင်းယှဉ်ဖို့ ကြိုးစားတာ) များက error များ ဖြစ်စေနိုင်ပါတယ်။ ဒီလိုပြဿနာတွေကို ဖြေရှင်းဖို့ **CAST** သို့မဟုတ် **CONVERT** ကိုသုံးပြီး data type တွေကို တူညီအောင် ပြောင်းလဲနိုင်တာက အရမ်းအသုံးဝင်ပါတယ်။

```
SELECT name FROM employees WHERE CAST(employee_id AS VARCHAR) = '1';
```

ဒီ query မှာ **employee_id** (INT) ကို **VARCHAR** (Text) ပုံစံသို့ ပြောင်းပြီး '1' နဲ့ နှိုင်းယှဉ်ထားပါတယ်။ အဲဒီလိုပြောင်းလိုက်ခြင်းဖြင့် data type မကိုက်ညီတဲ့ error တို့ကို ရှောင်ရှားနိုင်ပါတယ်။

ဒီအခန်းမှာတော့ SQL မှာ အသုံးများတဲ့ data type တွေဖြစ်တဲ့ **INT**, **DECIMAL**, **VARCHAR**, **CHAR**, **DATE**, **TIMESTAMP** စတာတွေကို လေ့လာခဲ့ပါတယ်။ ဘာကြောင့် data type ရွေးချယ်တာက အရေးကြီးလဲ၊ သူတို့က storage နဲ့ query speed တွေကို ဘယ်လိုသက်ရောက်မှုရှိနိုင်လဲ ဆိုတာတွေကို နားလည်လာပါပြီ။

ထို့ပြင် **CAST** နဲ့ **CONVERT** ကိုသုံးပြီး data type ပြောင်းတာတွေကိုလည်း လေ့လာခဲ့ပါတယ်။ အဲဒီအရာတွေကတော့ database design ပြုလုပ်တာ၊ query တွေ ရေးတာမှာ အခြေခံအဖြစ် အရမ်းအရေးကြီးပါတယ်။

Chapter 4: Advanced Filtering and Conditions

ရှေ့အခန်း (၂) မှာ **WHERE** clause နောက်မှာ conditions တွေကို ပေါင်းစပ်နိုင်ဖို့ **AND** ကို logical operator တစ်ခုအနေဖြင့် အသုံးပြုခဲ့တာကို သတိပြုမိမယ် ထင်ပါတယ်။ တိကျသေချာတဲ့ data records တွေကို filtered လုပ်ပြီး ဆွဲထုတ်ယူတဲ့အခါမှာ ထိရောက်တဲ့ SQL queries တွေကို ရေးသားနိုင်ဖို့ အင်မတန် အရေးကြီးလာပါပြီ။ ဆက်လက်ပြီးမှ ယခုအခန်းမှာ logical operators တွေရဲ့ အလုပ်လုပ်ပုံကို အသေးစိတ် လေ့လာသွားကြရအောင်ပါ။

1. Logical Operators

1.1 AND, OR, NOT for Complex Conditions

SQL မှာ **AND**, **OR**, **NOT** operators တွေကို အသုံးပြုခြင်းဖြင့် ရှုပ်ထွေးတဲ့ filtering conditions တွေကို ပြုလုပ်နိုင်ပါတယ်။

- **AND**: **AND** နဲ့ ချိတ်ထားတဲ့ conditions တွေအားလုံး မှန်မှသာ result က **TRUE** ဖြစ်မှာပါ။
- **OR**: **OR** နဲ့ ချိတ်ထားတဲ့ conditions တွေထဲက တစ်ခုမဟုတ်တစ်ခုမှန်ရုံနဲ့ **TRUE** ဖြစ်ပါတယ်။
- **NOT**: **NOT** ကတော့ condition နဲ့ ဆန့်ကျင်ဘက် data records တွေကိုသာ ရွေးပေးတာပါ။

တစ်ခါတည်းမှာပဲ တည်ဆောက်ပြီးသား **employees** table ထဲကို အောက်ပါဝန်ထမ်းတွေရဲ့ data records များကို မိမိကိုယ်တိုင် ထည့်သွင်းနိုင်မယ်လို့ ယူဆပါတယ်။

// Source Code Link:

[https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-4/01-insert-employees-data.s](https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-4/01-insert-employees-data.sql)
[ql](#)

```
company_db=# SELECT * FROM employees;
 employee_id | name       | position      | salary  | gender | email              | contract_end_date | department | age
-----
1 | Alice Smith | Engineer      | 5000.00 | F      | alice@example.com  | 2026-12-31        | IT         | 28
2 | Bob Johnson | Manager      | 7200.50 | M      | bob@example.com    | 2025-11-30        | MGMT      | 42
3 | Chan Myae  | Developer    | 5000.00 | M      | chan@example.com   | 2026-03-15        | IT         | 30
4 | Devi Kumar | Analyst       | 5000.00 | F      | devi@example.com   | 2025-09-30        | ANLY      | 35
5 | Eaint San  | Designer      | 5200.25 | F      | eaint@example.com  | 2026-06-30        | DSGN      | 27
6 | Faisal Rahman | Technician   | 5000.00 | M      | faisal@example.com | 2025-12-15        | IT         | 32
7 | Hla Myint  | HR Specialist | 4700.50 | F      | hla@example.com    | 2026-01-31        | HR        | 38
8 | Imran Khan | Project Lead  | 6800.00 | M      | imran@example.com  | 2026-08-31        | MGMT      | 40
9 | Khin Zaw   | Support Staff | 3500.00 | M      | khin@example.com   | 2025-07-31        | SUPP      | 25
10 | Lin Htet   | Marketing     | 5100.00 | F      | lin@example.com    | 2026-04-30        | MKTG      | 29
(10 rows)
```

company_db=#

ထည့်ပြီးသွားပြီဆိုရင် အောက်က query ကို SQL shell မှာ ရေးပြီး စမ်းကြည့်နိုင်ပါတယ်။


```
SELECT name, salary, gender FROM employees WHERE salary = 5000 AND
gender = 'M';
```

ပထမဆုံးအားဖြင့် salary 5000 ရတဲ့ employee ဟူသမျှကို ဆွဲထုတ်ပါတယ်။ ဒုတိယအနေဖြင့် male ဖြစ်တဲ့ employee တွေကို ထပ်မံစစ်ထုတ်ပါတယ်။ နောက်ဆုံးမှာတော့ AND operator ကိုသုံးပြီး condition နှစ်ရပ်လုံးမှာ true ဖြစ်တဲ့ employee တွေကို ပြသပေးတာ ဖြစ်ပါတယ်။ ဆိုလိုတာက လစာ 5000 ရရှိပြီး တချိန်တည်းမှာ အမျိုးသားဖြစ်ရပါမယ်လို့ instruction ပေးတာပါ။

```
company_db=# SELECT name, salary, gender FROM employees WHERE salary = 5000 AND gender = 'M';
 name      | salary | gender
-----+-----+-----
 Chan Myae | 5000.00 | M
 Faisal Rahman | 5000.00 | M
(2 rows)

company_db=#
```

အကယ်၍ employee ဟာ 'M' ဒါမှမဟုတ် 'F' ဖြစ်ရမယ်ဆိုရင် male ရော၊ female ရော နှစ်မျိုးလုံး လက်ခံနိုင်တယ်လို့ ဆိုလိုတာပါ။ ဒီလိုအခြေအနေမှာတော့ ထပ်ဆင့်ပြီးမှ OR operator ကို အသုံးပြုနိုင်ပါတယ်။ ဒါပေမယ့် query မှာ AND နဲ့ OR နှစ်ခုလုံးပါဝင်နေတဲ့အတွက် OR conditions တွေကို သီးသန့် () အတွင်း ထည့်ရေးဖို့လိုပါတယ်။ Precedence အရ ရေးတာဖြစ်ပါတယ်။

```
SELECT name, salary, gender FROM employees WHERE salary = 5000 AND
(gender = 'M' OR gender = 'F');
```

ဒါကြောင့် OR နဲ့ ထပ်စစ်လိုက်တဲ့အခါ employee လေးယောက်ကို ဆွဲထုတ်ပေးမှာ ဖြစ်ပါတယ်။

```
company_db=# SELECT name, salary, gender FROM employees WHERE salary = 5000 AND (gender = 'M' OR gender = 'F');
 name      | salary | gender
-----+-----+-----
 Alice Smith | 5000.00 | F
 Chan Myae | 5000.00 | M
 Devi Kumar | 5000.00 | F
 Faisal Rahman | 5000.00 | M
(4 rows)
```

လိုအပ်ချက်အရ အလုပ်အကိုင်မှာ Manager မဟုတ်တဲ့ ဝန်ထမ်းတွေကို ရှာဖွေချင်ရင်တော့ NOT operator ကို သုံးပြီး 'Manager' မဟုတ်တဲ့ ဝန်ထမ်းတွေကို ဆွဲထုတ်နိုင်ပါတယ်။

```
SELECT name, position, salary FROM employees WHERE NOT position =
'Manager';
```

အခုလိုပဲ Manager ကလွဲပြီး ကျန်တဲ့ employee အားလုံးကို ပြပေးမှာ ဖြစ်ပါတယ်။

```
[company_db=# SELECT name, position, salary FROM employees WHERE NOT position = 'Manager';
name          | position    | salary
-----+-----+-----
Alice Smith   | Engineer   | 5000.00
Chan Myae     | Developer  | 5000.00
Devi Kumar    | Analyst     | 5000.00
Eaint San     | Designer    | 5200.25
Faisal Rahman | Technician | 5000.00
Hla Myint     | HR Specialist | 4700.50
Imran Khan    | Project Lead | 6800.00
Khin Zaw      | Support Staff | 3500.00
Lin Htet      | Marketing  | 5100.00
(9 rows)
```

Note: NOT အစား <> (not equal) သို့မဟုတ် != ကိုလည်း အသုံးပြုနိုင်တာကို တစ်ခါတည်း သိထားစေချင်ပါတယ်။

1.2 Operator Precedence

ကျွန်တော်တို့ ငယ်ငယ်တုန်းက ကျောင်းမှာသင်ယူခဲ့ရတဲ့ ရိုးရိုးသင်္ချာပုစ္ဆာတွေကို မှတ်မိကြမယ် ထင်ပါတယ်။ သင်္ချာပုစ္ဆာတစ်ပုဒ်ကို ဖြေရှင်းဖို့ဆိုရင် +, -, ×, ÷ တို့လို သင်္ကေတတွေ (operators) ကို အသုံးပြုရပါတယ်။ ဒါပေမယ့် အဲဒီ operators တွေကို မိမိစိတ်ကြိုက် လွတ်လပ်စွာ အစဉ်လိုက် ဖြေရှင်းလိုက်ရင် မှားသွားနိုင်ပါတယ်။ အဲဒါကြောင့် သင်္ချာမှာ လုပ်ဆောင်မှုဦးစားပေးအစီအစဉ် (order of operations) ဆိုတဲ့ စည်းမျဉ်းတစ်ခုရှိပါတယ်။ အဲဒီစည်းမျဉ်းကိုအခြေခံပြီးမှ တစ်ဆင့်ချင်းစီ စနစ်တကျ ဖြေရှင်းရတာ ဖြစ်ပါတယ်။

ဥပမာ: $2 + 3 \times (8 - 2) / 6 - 1$

ပထမဦးဆုံး (8 - 2) ကို ဖြေရှင်းပါတယ်။ အဲဒါကနေ 6 ရပါတယ်။ ပြီးတော့ ဘယ်ဘက်ကနေ ညာဘက်ကို မြှောက် (×) နဲ့ စား (÷) ကို လုပ်ပါမယ်။ $3 \times 6 = 18$ ၊ $18 \div 6 = 3$ ဖြစ်သွားပါတယ်။ နောက်ဆုံးအဆင့်မှာတော့ ပေါင်း (+) နဲ့ နှုတ် (-) ကို လုပ်ပါမယ်။ $2 + 3 = 5$ ၊ $5 - 1 = 4$ ဖြစ်ပါတယ်။ ဒီလို တစ်ဆင့်ချင်း ဖြေရှင်းရတာဟာ operator precedence ကြောင့်ပဲ ဖြစ်ပါတယ်။

SQL မှာလည်း AND, OR, NOT operators တွေကို ပေါင်းစပ်အသုံးပြုတဲ့အခါ ဦးစားပေးမှုအစီအစဉ် (operator precedence) ကို လိုက်နာဖို့လိုပါတယ်။ SQL မှာ operators တွေရဲ့ ဦးစားပေးအဆင့် တွေကတော့ အောက်ပါအတိုင်း ဖြစ်ပါတယ်။

1. NOT
2. AND
3. OR

```
SELECT name, salary, department FROM employees WHERE salary > 5000 OR
department = 'IT' AND age < 30;
```

Operator precedence အရ SQL မှာ **OR** ထက် **AND** ကို ဦးစားပေးလုပ်ဆောင်ပါတယ်။ ဆိုလိုတာက SQL ဟာ ပထမဦးဆုံး `department = 'IT' AND age < 30` ကို အရင်စစ်ပြီး ထိုရလဒ်နဲ့ `salary > 50000` ကို **OR** နဲ့ ပေါင်းပြီး data filtering လုပ်ပေးတာပဲ ဖြစ်ပါတယ်။

```
company_db=# SELECT name, salary, department FROM employees WHERE salary > 5000 OR department = 'IT' AND age < 30;
 name      | salary | department
-----+-----+-----
 Alice Smith | 5000.00 | IT
 Bob Johnson | 7200.50 | MGMT
 Eaint San   | 5200.25 | DSGN
 Imran Khan  | 6800.00 | MGMT
 Lin Htet    | 5100.00 | MKTG
(5 rows)

company_db=#
```

ပိုပြီးတိကျရှင်းလင်းအောင် ဖော်ပြချင်တယ်ဆိုရင် **parentheses ()** ကိုသုံးဖို့ အကြံပြုပါတယ်။ ဥပမာ - `salary > 50000` ထက်များပြီး `department = 'IT'` လည်းဖြစ်ရမယ် ဒါမှမဟုတ် `age < 30` အောက်ဖြစ်ရမယ် ဆိုရင်တော့။

```
SELECT name, salary, department, age FROM employees WHERE (salary >
5000 AND department = 'IT') OR age < 30;
```

ဒါဆိုတော့ 'IT' ဌာနထဲက ဝင်ငွေ 5000 ကျော်တဲ့သူတွေ ဒါမှမဟုတ် အသက် 30 အောက်ရှိတဲ့ ဝန်ထမ်းတွေကို ရွေးထုတ်ပြသပေးမှာ ဖြစ်ပါတယ်။

```
company_db=# SELECT name, salary, department, age FROM employees WHERE (salary > 5000 AND department = 'IT') OR age < 30;
 name      | salary | department | age
-----+-----+-----+----
 Alice Smith | 5000.00 | IT          | 28
 Eaint San   | 5200.25 | DSGN        | 27
 Khin Zaw    | 3500.00 | SUPP        | 25
 Lin Htet    | 5100.00 | MKTG        | 29
(4 rows)

company_db=#
```

2. Pattern Matching and Ranges

SQL မှာ data တွေကို စစ်ထုတ်ဖို့အတွက် **IN**, **BETWEEN**, **LIKE** တို့လို နောက်ထပ် operator တွေလည်း အသုံးပြုနိုင်ပါတယ်။

2.1 IN for Lists

IN operator ကို SQL မှာ **WHERE** clause ထဲမှာ အသုံးပြုပြီး တန်ဖိုးများစွာ သတ်မှတ်ချင်တဲ့အခါ သုံးပါတယ်။ ဒါမှမဟုတ် subquery ခွဲတစ်ခုပဲဖြစ်ဖြစ် သတ်မှတ်ချင်တဲ့အခါလည်း သုံးလို့ရပါတယ်။ အဓိပ္ပါယ်ကတော့ column တစ်ခုထဲမှာ မိမိသတ်မှတ်ထားတဲ့ **values list** ထဲက တစ်ခုခုနဲ့ ကိုက်ညီတဲ့ row တွေအကုန်လုံးကို ဆွဲထုတ်ပေးတာပဲ ဖြစ်ပါတယ်။

Sample SQL Query Using IN Operator

```
SELECT column1, column2 FROM table_name WHERE column_name IN (Value1, Value2, Value3);
```

```
SELECT name, department FROM employees WHERE department IN ('IT', 'HR', 'MGMT');
```

'IT', 'HR', ဒါမှမဟုတ် 'MGMT' ဌာနတွေမှာ လုပ်ကိုင်နေတဲ့ ဝန်ထမ်းတွေအကုန်လုံးကို ပြပါလို့ instruction ပေးလိုက်တာပါ။

```
[company_db=# SELECT name, department FROM employees WHERE department IN ('IT', 'HR', 'MGMT');
 name | department 
-----+-----
 Alice Smith | IT
 Bob Johnson | MGMT
 Chan Myae | IT
 Faisal Rahman | IT
 Hla Myint | HR
 Imran Khan | MGMT
 (6 rows)
```

IN operator မှာ သတ်မှတ်ထားတဲ့ တန်ဖိုးတွေ (Value1, Value2, ...) ကို သုံးလို့ရသလို **subquery** တစ်ခုကိုပါ ထည့်သုံးနိုင်ပါသေးတယ်။

```
SELECT name, department, age FROM employees WHERE department IN (SELECT department FROM employees WHERE age ≥ 35);
```

Subquery (**SELECT department FROM employees WHERE age ≥ 35**) ကတော့ အသက် ၃၅ နဲ့အထက်ရှိတဲ့ ဝန်ထမ်းတွေ အလုပ်လုပ်နေတဲ့ ဌာနနာမည်တွေကို ရှာထုတ်ပေးတာပါ။ အပြင်ဘက် outer query ကတော့ အဲ့ဒီဌာနတွေထဲမှာ ပါဝင်တဲ့ ဝန်ထမ်းတွေရဲ့ **name**, **department** နဲ့ **age** ကို ထပ်ရွေးထုတ် ပြသပေးတာဖြစ်ပါတယ်။

```
company_db=# SELECT name, department, age FROM employees WHERE department IN (SELECT department FROM employees WHERE age >= 35);
 name | department | age 
-----+-----+-----
 Bob Johnson | MGMT      | 42 
 Devi Kumar  | ANLY      | 35 
 Hla Myint   | HR        | 38 
 Imran Khan  | MGMT      | 40 
(4 rows)

company_db=#
```

Note: တခါတလေ **OR** operator ကို အကြိမ်ကြိမ်သုံးနေရင် ရှုပ်နိုင်ပါသေးတယ်။ အဲဒီအစား **IN** operator ကို သုံးလိုက်တာနဲ့ တူညီတဲ့ရလဒ် ရမှာဖြစ်သလို ပိုမိုရှင်းလင်းသွားမှာ ဖြစ်ပါတယ်။

2.2 BETWEEN for Ranges

BETWEEN ဆိုတာဟာလည်း logical operator တစ်ခုဖြစ်ပြီး တန်ဖိုးတစ်ခုဟာ သတ်မှတ်ထားတဲ့ ကြားအကွာအဝေးအတွင်းမှာ ရှိလားဆိုတာ စစ်ဖို့ အသုံးပြုပါတယ်။ ဒီ operator ကို ကဏန်းတွေ (integers), စာလုံးတွေ (characters), ဒါမှမဟုတ် ရက်စွဲတွေ (dates) နဲ့အတူ သုံးလို့ရပါတယ်။ များသောအားဖြင့် $\geq$ နဲ့ $\leq$ ဆိုတဲ့ syntax ကို **BETWEEN** နဲ့ အစားထိုးပြီး ရှင်းရှင်းလင်းလင်းဖြစ်အောင် ရေးနိုင်ပါတယ်။

```
SELECT name, position, salary FROM employees WHERE salary >= 4000 AND
salary <= 6000;

SELECT name, position, salary FROM employees WHERE salary BETWEEN 4000
AND 6000;
```

```
company_db=# SELECT name, position, salary FROM employees WHERE salary BETWEEN 4000 AND 6000;
 name | position | salary 
-----+-----+-----
 Alice Smith | Engineer | 5000.00
 Chan Myae  | Developer | 5000.00
 Devi Kumar  | Analyst  | 5000.00
 Eaint San  | Designer | 5200.25
 Faisal Rahman | Technician | 5000.00
 Hla Myint   | HR Specialist | 4700.50
 Lin Htet    | Marketing | 5100.00
(7 rows)
```

Salary 40000 နဲ့ 60000 တိတိရတဲ့ ဝန်ထမ်းတွေအပြင် အဲဒီတန်ဖိုးနှစ်ခုကြားမှာရှိတဲ့ ဝန်ထမ်းတွေကိုလည်း ရယူပေးမှာ ဖြစ်ပါတယ်။

2.3 LIKE for Pattern Matching

PostgreSQL မှာ **LIKE** operator ကို table ထဲက data တွေကို သတ်မှတ်ထားတဲ့ pattern နဲ့ ကိုက်ညီမှုရှိမရှိစစ်ဖို့ အသုံးပြုပါတယ်။ အထူးသဖြင့် employee records တို့လို data တွေကို filter လုပ်ချင်တဲ့အခါမှာ အသုံးဝင်ပါတယ်။ **LIKE** operator သုံးတဲ့အခါမှာ wildcards (အစားထိုးသင်္ကေတ) နှစ်မျိုးကို သုံးလို့ရပါတယ်။ '%' ဟာ sequence of characters အကွာရာစဉ်တစ်ခုလုံးနဲ့ ကိုက်ညီမှုရှိမရှိ စစ်ဆေးပြီး '_' သည် အကွာရာတစ်လုံးတည်းနှင့် ကိုက်ညီမှုကို စစ်ဆေးပါတယ်။

PostgreSQL မှာ **LIKE** operator က ပုံမှန်အားဖြင့် case-sensitive ဖြစ်ပါတယ်။ ဒါဆို **LIKE 'A%'** လို့ ရိုက်ရင် **Alice** လို အကြီးစာလုံးနဲ့ စတဲ့ နာမည်တွေကိုသာ ရှာပေးပါလိမ့်မယ်။ **alice** လို အသေးစာလုံးနဲ့ စတဲ့ နာမည်တွေကို ရှာချင်ရင်တော့ **ILIKE** operator ကို သုံးရပါတယ်။ **ILIKE** က **LIKE** ကဲ့သို့ အလုပ်လုပ်ပေမယ့် စာလုံးအကြီးအသေးကို မခွဲခြားပဲ ရှာပေးတာ ဖြစ်ပါတယ်။

MySQL နဲ့ ကွာခြားချက်တစ်ခုကတော့ PostgreSQL မှာ **LIKE** operator က case-sensitive ဖြစ်တာပါ။ ဒါကြောင့် စာလုံးအကြီးအသေးကို မခွဲခြားချင်ရင် **ILIKE** ကိုသုံးသင့်ပါတယ်။ **LIKE** နဲ့ **ILIKE** ဟာ **SELECT, UPDATE, DELETE** statement တွေရဲ့ **WHERE** clause နဲ့ တွဲဖက် အသုံးပြုနိုင်ပါတယ်။

ဥပမာအားဖြင့် 'son' နဲ့ဆုံးတဲ့ နာမည်တွေ (Johnson, Wilson) ကို ရှာချင်ရင် '%son' ကို သုံးနိုင်ပြီး ဒုတိယစာလုံးမှာ 'a' ပါတဲ့ နာမည်တွေ (Sarah, Daniel) ကို ရှာဖို့တော့ '_a%' ကို သုံးနိုင်ပါတယ်။

```
SELECT name FROM employees WHERE name LIKE 'A%';
```

```
[company_db=# SELECT name FROM employees WHERE name LIKE 'A%';
 name
-----
 Alice Smith
(1 row)
```

3. Handling NULL Values

SQL မှာ **NULL** ဆိုတာက data မရှိတာ သို့မဟုတ် မသိရသေးတဲ့ တန်ဖိုးကို ကိုယ်စားပြုတာပါ။ **NULL** တန်ဖိုးတွေကို စစ်တဲ့အခါမှာတော့ သတိထားစရာ ရှိပါတယ်။

3.1 IS NULL and IS NOT NULL

NULL တန်ဖိုးတွေကို စစ်ဖို့တော့ **IS NULL** နဲ့ **IS NOT NULL** ဆိုတဲ့ condition တွေကို သုံးရပါတယ်။ **=, !=** စတဲ့ ပုံမှန် comparison operators တွေဟာ **NULL** နဲ့တော့ အလုပ်လုပ်မှာ မဟုတ်ပါဘူး။


```
SELECT name, email FROM employees WHERE email IS NULL;
```

employees table ထဲက employee တစ်ယောက်ချင်းစီမှာ email ရှိနေပြီးသားဖြစ်တာကြောင့် query ရဲ့ result က **0 rows** ပဲ ပြမှာဖြစ်ပါတယ်။

```
[company_db=# SELECT name, email FROM employees WHERE email IS NULL;
 name | email
-----+-----
(0 rows)

company_db=#
```

email ရှိတဲ့ employees များကို ရှာချင်ရင်တော့ ဒီလို ရေးလို့ရပါတယ်။

```
SELECT name, email FROM employees WHERE email IS NOT NULL;
```

email column မှာ **NULL** တန်ဖိုးမရှိဘဲ အကုန်လုံးမှာ email ရှိနေတော့ employee တွေရဲ့ record အားလုံးကို ပြပေးတာပါ။

```
[company_db=# SELECT name, email FROM employees WHERE email IS NOT NULL;
 name | email
-----+-----
Alice Smith | alice@example.com
Bob Johnson | bob@example.com
Chan Myae | chan@example.com
Devi Kumar | devi@example.com
Eaint San | eaint@example.com
Faisal Rahman | faisal@example.com
Hla Myint | hla@example.com
Imran Khan | imran@example.com
Khin Zaw | khin@example.com
Lin Htet | lin@example.com
(10 rows)
```

3.2 Common NULL-Related Mistakes

NULL ကို စစ်တဲ့အခါ အနည်းဆုံး သတိထားရမယ့် အမှားလေးတွေရှိတတ်ပါတယ်။

Checking NULL with =

```
SELECT name, email FROM employees WHERE email = NULL;
```

ဒီ query statement ကနေဆိုရင် ဘာ result မှ မပြန်ပေးပါဘူး။ ဘာလို့ဆို `email = NULL` လို့ check လုပ်ထားလို့ပဲ ဖြစ်ပါတယ်။ ဒီလိုရေးတာနဲ့ အမြဲတမ်း `false` သို့မဟုတ် `unknown` ဖြစ်ပြီးတော့ `true` မဖြစ်တော့ပါဘူး။ အဲဒါကြောင့် email column ထဲမှာ `NULL` ပါတဲ့ row တွေလည်း ဒီ query ကနေ ပြန်မလာနိုင်ပါဘူး။ အဲဒီအစား `IS NULL` ကို အသုံးပြုရပါမယ်။

Problem with NOT IN

ထိုနည်းတူ `NOT IN` ကို SQL မှာ သုံးတဲ့အခါ သတိထားရမယ့်ဟာ တစ်ခုရှိပါတယ်။ Subquery သို့မဟုတ် value list ထဲမှာ `NULL` တန်ဖိုးတစ်ခု ပါလာလိုက်တာနဲ့တော့ `NOT IN` condition ကနေ အမြဲတမ်း `FALSE` သို့မဟုတ် `UNKNOWN` ပဲဖြစ်နိုင်ပါတယ်။ `TRUE` ဖြစ်တဲ့ row တွေကိုတောင် မပြပေးတော့ပါဘူး။

```
SELECT name, department FROM employees WHERE department NOT IN ('IT', 'HR', NULL);
```

ဘာလို့လဲဆိုတော့ `NOT IN` ဟာ value တစ်ခုချင်းစီနဲ့ နှိုင်းယှဉ်သွားမှာပဲ ဖြစ်ပါတယ်။ `NULL` နဲ့ နှိုင်းယှဉ်တဲ့ အခါ result က `UNKNOWN` ဖြစ်ပြီး SQL မှာတော့ အဲဒါကို `FALSE` အဖြစ် ယူသွားလို့ပါ။ ဒါကြောင့် `NOT IN` သုံးမယ်ဆိုရင် `NULL` တွေကို သက်သက်စီ ဖယ်ထုတ်ထားဖို့လိုပါတယ်။

4. Case Statements for Conditional Logic

SQL မှာ `CASE` statement ကို conditions ပေါ်မူတည်ပြီး မတူညီတဲ့ အခြေအနေတိုင်းအတွက် မတူညီတဲ့ result တွေပေးချင်တဲ့အခါ အသုံးပြုပါတယ်။ ဒါက programming language တွေမှာရှိတဲ့ if-then-else လို့ အလုပ်လုပ်ပေးသွားတာပါ။

4.1 Syntax and Use Cases

```
SELECT column_name,
       CASE
         WHEN condition1 THEN result1
         WHEN condition2 THEN result2
         ELSE other_result
       END AS result_column
FROM table_name;
```

Example 1: Salary Level Classification

```
SELECT name, salary,
       CASE
         WHEN salary > 6000 THEN 'High'
         WHEN salary BETWEEN 4000 AND 6000 THEN 'Medium'
         ELSE 'Low'
       END AS salary_level
FROM employees;
```

salary ဟာ 6000 ထက်ကြီးရင် 'High' လို့သတ်မှတ်ပြီး 4000 နှင့် 6000 ကြားဆိုရင် 'Medium'။ နောက်ဆုံးဘယ်ဟာနဲ့မှ မကိုက်ညီဘူးဆိုရင် 'Low' လို့သတ်မှတ်လိုက်တာပါ။ CASE ကနေ ထွက်လာတဲ့ ရလဒ်ကို salary_level ဆိုတဲ့ column အသစ်တစ်ခုအနေနဲ့ သိမ်းပြီး ဖော်ပြပေးတာပါ။ ဒါပေမယ့် အဲ့ဒီ column ကို employees table ထဲမှာ တကယ်သိမ်းသွားမှာ မဟုတ်ပါဘူး။ မိမိကိုယ်တိုင် SQL shell မှာ ရေးပြီး စမ်းကြည့်နိုင်ပါတယ်။

Example 2: Department Name Change

ဒီတစ်ခါ department နာမည်တွေကို long form ပုံစံအဖြစ် ပြောင်းဖော်ပြကြည့်မှာ ဖြစ်ပါတယ်။

```
SELECT name, department,
       CASE department
         WHEN 'IT' THEN 'Information Technology'
         WHEN 'HR' THEN 'Human Resources'
         ELSE 'Other'
       END AS department_name
FROM employees;
```

တစ်ခုသတိပြုရမှာက CASE statement ဟာ employees table ထဲက data ကို ဖတ်ပြီး ပြသပေးတာသာ ဖြစ်ပြီး database ထဲမှာ တကယ်သိမ်းဆည်းတာ မဟုတ်ပါဘူး။ original data အပေါ်မှာ ဘာသက်ရောက်မှုမှ မရှိပါဘူး။ ဒါကြောင့် သူ့ကို read-only operation လို့ ခေါ်တာဖြစ်ပါတယ်။

4.2 Simplifying Complex Queries

CASE က ရှုပ်ထွေးတဲ့ logic တွေကို တစ်နေရာတည်းမှာ စုစည်းပြီး လွယ်ကူစွာ ဖြေရှင်းပေးနိုင်ပါတယ်။ ဒီတော့ employees တွေရဲ့ လစာတွေကို department အလိုက် မတူညီအောင် တိုးကြည့်ကြပါမယ်။

```
SELECT name, salary, department,
       CASE
         WHEN department = 'IT' THEN salary * 1.10
         WHEN department = 'HR' THEN salary * 1.05
         ELSE salary
       END AS adjusted_salary
FROM employees;
```

IT department အတွက် 10% နှင့် **HR** department အတွက် 5% လစာတိုးပေးပြီး အခြား department တွေအတွက်တော့ မပြောင်းလဲပါ။

```
company_db=# SELECT name, salary, department,
company_db=#       CASE
company_db=#         WHEN department = 'IT' THEN salary * 1.10
company_db=#         WHEN department = 'HR' THEN salary * 1.05
company_db=#         ELSE salary
company_db=#       END AS adjusted_salary
company_db=# FROM employees;
 name      | salary | department | adjusted_salary
-----+-----+-----+-----
Alice Smith | 5000.00 | IT          | 5500.0000
Bob Johnson | 7200.50 | MGMT        | 7200.50
Chan Myae   | 5000.00 | IT          | 5500.0000
Devi Kumar  | 5000.00 | ANLY        | 5000.00
Eaint San   | 5200.25 | DSGN        | 5200.25
Faisal Rahman | 5000.00 | IT          | 5500.0000
Hla Myint   | 4700.50 | HR          | 4935.5250
Imran Khan  | 6800.00 | MGMT        | 6800.00
Khin Zaw    | 3500.00 | SUPP        | 3500.00
Lin Htet    | 5100.00 | MKTG        | 5100.00
(10 rows)
```

ဒီအခန်းမှာတော့ data တွေကို ပိုတိကျပြီး စနစ်တကျ စစ်ထုတ်နိုင်ဖို့ နည်းလမ်းတချို့ကို လေ့လာခဲ့ပါတယ်။ **AND**, **OR**, **NOT** operators နှင့် **()** ကို အသုံးပြုပြီး ရှုပ်ထွေးတဲ့ conditions တွေကို အချိုးကျဖန်တီးနိုင်တာကို မြင်တွေ့ခဲ့ပါတယ်။ **IN**, **BETWEEN**, **LIKE** တို့ကလည်း data ရွေးထုတ်တဲ့အခါမှာ အရမ်းအသုံးဝင်တယ်ဆိုတာ သိလာခဲ့ကြပါတယ်။ **NULL** တန်ဖိုးတွေကို စစ်ဖို့ **IS NULL** နဲ့ **IS NOT NULL** ကို သုံးရမယ်ဆိုတာနဲ့ အတူ သတိထားစရာလေးတွေကိုလည်း ပြောပြခဲ့ပါတယ်။ နောက်ဆုံးမှာ **CASE** statement နဲ့ ရှုပ်တဲ့ logic တွေကို တစ်နေရာတည်းမှာ စနစ်တကျရေးနိုင်လို့ query တွေ ဖတ်ရလွယ်သွားတာကို သိလာခဲ့ကြပါတယ်။

Chapter 5: Joins and Relationships

Databases ထဲမှာရှိတဲ့ table တွေဟာ တစ်ခုနဲ့တစ်ခု ဆက်စပ်နေတတ်ပါတယ်။ အဲ့ဒီဆက်စပ်မှုတွေကို အခြေခံပြီး မတူညီတဲ့ table တွေထဲက data တွေကိုပေါင်းပြီး တစ်စုတည်းအဖြစ် ထုတ်ယူနိုင်ပါတယ်။ ဒီအခန်းမှာတော့ table တွေကြားက relationship တွေ ဘယ်လိုအလုပ်လုပ်လဲ၊ SQL မှာ joins တွေကို ဘယ်လိုသုံးရသလဲဆိုတာကို ဆက်လက်လေ့လာသွားပါမယ်။

1. Understanding Table Relationships

Database တစ်ခုအတွင်းမှာ data တွေကို table တစ်ခုချင်းစီခွဲပြီး သိမ်းဆည်းထားတတ်ပါတယ်။ ဥပမာ - စာကြည့်တိုက်တစ်ခုရဲ့ database မှာ books ဆိုတဲ့ table နဲ့ readers ဆိုတဲ့ table တို့လိုပဲ မတူညီတဲ့ table များစွာရှိနိုင်ပါတယ်။ ဒီ Table တွေဟာ တစ်ခုနဲ့တစ်ခု ဆက်နွှယ်မှု relationship ရှိနေနိုင်ပြီး အဲ့ဒီဆက်နွှယ်မှုကို အသုံးပြုပြီး data တွေကို လိုအပ်သလို ချိတ်ဆက်ပေါင်းစပ်နိုင်ပါတယ်။

Table တွေကြား relationship တစ်ခု တည်ဆောက်ခြင်းဟာ database ထဲမှာ data တွေကို စနစ်တကျ စုစည်းနိုင်ဖို့နဲ့ ထိရောက်စွာ ရှာဖွေထုတ်ယူနိုင်ဖို့အတွက် အရေးပါပါတယ်။ ဒါပေမယ့် relationship တစ်ခု တည်ဆောက်တဲ့အခါမှာတော့ primary key နဲ့ foreign key ဆိုတဲ့ အရေးကြီးတဲ့ key နှစ်ခုကို အသုံးပြုရပါတယ်။ ဒီ key နှစ်ခုရဲ့ အလုပ်လုပ်ပုံဟာ နည်းနည်းနားလည်ရခက်နိုင်တာမလို့ နောက်အပိုင်းမှာ နမူနာတွေနဲ့အတူ ဆက်လက်ရှင်းပြသွားပါမယ်။

/* Relationship ပုံစံ သုံးမျိုးသုံးစား ရှိတယ်လို့ လွယ်လွယ်မှတ်ယူနိုင်ပါတယ်။ */

1.1 One-to-One (တစ်ခုချင်းဆက်နွှယ်မှု)

ဒီလို relationship မှာ table တစ်ခုထဲက record တစ်ခုဟာ တခြား table ထဲက record တစ်ခုတည်းနဲ့သာ ဆက်နွှယ်နေပါတယ်။ ဒီလိုပါ - ကျောင်းသားတစ်ဦးမှာ မှတ်ပုံတင်နံပါတ်တစ်ခုသာရှိပြီး အဲ့ဒီနံပါတ်ကလည်း ကျောင်းသားတစ်ဦးတစ်ယောက်တည်းနဲ့ပဲ သက်ဆိုင်ပါတယ်။ ဒီလိုမျိုး relationship type ကို one-to-one အပြန်အလှန်တစ်ခုချင်း ဆက်နွှယ်မှုလို့ခေါ်ပါတယ်။

1.2 One-to-Many (တစ်ခုမှ အများနှင့်ဆက်နွယ်မှု)

One-to-Many မှာကျတော့ table တစ်ခုထဲက record တစ်ခုဟာ တခြား table ထဲက record များစွာနဲ့ ဆက်နွယ်နေနိုင်ပါတယ်။ ဥပမာအနေနဲ့ စာကြည့်တိုက်တစ်ခုမှာ စာအုပ်တစ်အုပ်တည်းကိုပဲ လူအများအပြားက မိမိအချိန်အလိုက် ငှားယူလို့ရပါတယ်။ ဒါဆို အဲ့ဒီစာအုပ်တစ်အုပ်တည်းပင် ငှားယူမှုမှတ်တမ်း (borrow records) များစွာနဲ့ ဆက်စပ်နေနိုင်တာ ဖြစ်ပါတယ်။ ဒါကို One-to-Many relationship လို့ခေါ်ပါတယ်။

1.3 Many-to-Many (အများနှင့် အများဆက်နွယ်မှု)

ဒီအမျိုးအစားမှာတော့ table ထဲက record များစွာဟာ တခြား table ထဲက record များစွာနဲ့ ဆက်နွယ်နေနိုင်ပါတယ်။

ဥပမာ - ကျောင်းသားတစ်ဦးဟာ သင်တန်းတစ်ခုထက်ပိုပြီး တက်ရောက်နိုင်သလို လက်ရှိသူတက်နေတဲ့ သင်တန်းမှာပင် သူနဲ့အတူ တခြားကျောင်းသားများစွာ ရှိနိုင်ပါတယ်။ ဒါဆို သင်တန်းများစွာဟာ မတူညီတဲ့ ကျောင်းသားများစွာနဲ့ ဆက်သွယ်ရတော့မှာပါ။ ဒီလို Many-to-Many ဆက်နွယ်မှုကို တည်ဆောက်ဖို့ အတွက် ကျောင်းသားနဲ့ သင်တန်းတို့ကို ချိတ်ဆက်ပေးမယ့် junction table သို့မဟုတ် association table (ဆက်စပ်ဇယား) အသစ်တစ်ခု ထပ်တည်ဆောက်ပေးရမှာပဲ ဖြစ်ပါတယ်။

1.4 Role of Keys in Relationships

Primary Key - Table တစ်ခုအတွင်းမှာ record တစ်ခုချင်းစီကို ထူးခြားအောင် မှတ်သားပေးနိုင်တဲ့ column တစ်ခုပဲ ဖြစ်ပါတယ်။

ဥပမာ - **books** table ထဲမှာဆိုရင် **book_id** က စာအုပ်တစ်အုပ်ချင်းစီကို ခွဲခြားသိနိုင်အောင် မှတ်ထားတာမို့ **primary key** အဖြစ် သတ်မှတ်နိုင်ပါတယ်။

Foreign Key - တခြား table တစ်ခုထဲမှာရှိတဲ့ primary key ကို reference လုပ်ထားတဲ့ column တစ်ခုပဲ ဖြစ်ပါတယ်။

ဥပမာ - **borrow_records** ဆိုတဲ့ table ထဲမှာ **book_id** ဆိုတဲ့ column ရှိတယ်ဆိုပါစို့။ ဒါဟာ **books** table ထဲမှာရှိတဲ့ **book_id** (primary key) ကို ညွှန်းပြနေတဲ့ foreign key ဖြစ်နိုင်ပါတယ်။

ဒါကြောင့် **borrow_records** ထဲမှာ အသုံးပြုတဲ့ **book_id** တန်ဖိုးတွေဟာ **books** table ထဲမှာ တကယ်ရှိတဲ့ **book_id** တွေနဲ့ ကိုက်ညီဖို့ လိုပါတယ်။ အဲ့မှ ဘယ်စာအုပ်ကိုငှားထားသလဲဆိုတာ သိရမှာပါ။

ဒီလို primary key နှင့် foreign key ကြောင့်ပဲ table များအကြား ဆက်နွယ်မှု (relationships) တွေကို တည်ဆောက်နိုင်တာဖြစ်ပါတယ်။ ဒါကြောင့် **JOIN** keyword ကို အသုံးပြုလိုက်ရင်ဖြင့် မတူညီတဲ့ table တွေထဲမှာရှိတဲ့ data တွေကို အတူတူပေါင်းပြီး ရယူနိုင်ပြီဖြစ်ပါတယ်။

2. Types of Joins

SQL မှာ joins ဆိုတာ table နှစ်ခု သို့မဟုတ် အဲ့ထက်ပိုတဲ့ table တွေကို ချိတ်ဆက်ပြီး တစုတစည်းအဖြစ် data တွေကို ဆွဲထုတ်ဖို့ အသုံးပြုတဲ့ keyword ပါ။

Joins ကို ဘယ်လိုသုံးရမလဲဆိုတာ နမူနာတွေနဲ့ အောက်မှာ ဆက်ရှင်းပြပါမယ်။ ဒီနေရာမှာ database အသစ်ဆောက်တာ၊ table တွေ column နဲ့ row တွေထည့်တာကိုတော့ မရှင်းပြတော့ပါဘူး။ မိမိကြိုက်တဲ့ နမူနာတွေကို အသုံးပြုနိုင်ပါတယ်။ Source code ကိုလည်း အောက်က link မှာ ရယူနိုင်ပါတယ်။

// Source Code Link:

<https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-5/01-insert-books-data.sql>

```
[school_db=# SELECT * FROM books;
 book_id | title | author
-----+-----+-----
      1 | Myanmar Language Grade 1 | U Maung Maung
      2 | Mathematics Guide | Daw Mya Mya
      3 | History Book | U Tin Oo
      4 | Introduction to Science | Dr. Aye Chan
(4 rows)
```

borrow_records table ထဲက **book_id** ဟာ **books** table ရှိ **book_id** ကို ရည်ညွှန်းတဲ့ foreign key ဖြစ်ပါတယ်။ ပြီးတော့ **book_id** တန်ဖိုး 1 ရှိတဲ့ စာအုပ်ကို ငှားထားတဲ့ လူနှစ်ယောက်ရှိနေတာကို အထူးသတိပြုရမှာပါ။

```
[school_db=# SELECT * FROM borrow_records;
 borrow_id | book_id | borrower_name | borrow_date
-----+-----+-----+-----
      1 | 1 | Maung Aung | 2025-05-01
      2 | 1 | Mi Mi Khine | 2025-05-02
      3 | 2 | Soe Moe | 2025-05-03
(3 rows)
```

2.1 INNER JOIN: Matching Records

INNER JOIN က table နှစ်ခုအကြား ဆက်စပ်တဲ့ column ပေါ် မူတည်ပြီး ကိုက်ညီတဲ့ data records တွေကိုသာ ရွေးထုတ်ပေးတာ ဖြစ်ပါတယ်။

```
SELECT books.title, borrow_records.borrower_name,
borrow_records.borrow_date FROM books INNER JOIN borrow_records ON
books.book_id = borrow_records.book_id;
```

ဒီ query က **books** table နဲ့ **borrow_records** table ရဲ့ **book_id** တန်ဖိုးတွေ တူညီတဲ့ record တွေကိုပဲ ပေါင်းပြီး ပြပေးမှာ ဖြစ်ပါတယ်။

```
school_db=# SELECT books.title, borrow_records.borrower_name, borrow_records.borrow_date
school_db=# FROM books
school_db=# INNER JOIN borrow_records ON books.book_id = borrow_records.book_id;
      title                | borrower_name | borrow_date
-----+-----+-----
 Myanmar Language Grade 1 | Maung Aung    | 2025-05-01
 Myanmar Language Grade 1 | Mi Mi Khine   | 2025-05-02
 Mathematics Guide        | Soe Moe       | 2025-05-03
(3 rows)
```

2.2 LEFT JOIN

LEFT JOIN (သို့) **LEFT OUTER JOIN** ဆိုတာက table နှစ်ခုကို ပေါင်းစပ်တဲ့အခါမှာ left table ထဲက record အကုန်လုံးကို ရယူပြီး right table ထဲကတော့ ဆက်စပ်မှုရှိတဲ့ record တွေကိုပဲ တွဲဖက်ပြသပေးတာပါ။ right table မှာ ဆက်စပ်မှုမရှိတဲ့ record တွေရှိရင်တော့ **NULL** တန်ဖိုးတွေနဲ့ ပြထားမှာ ဖြစ်ပါတယ်။

ဒီနေရာမှာ left table, right table ဆိုပြီး နားမလည်နိုင်စရာ ရှိကောင်းရှိနိုင်ပါတယ်။ အတိုချုပ်အားဖြင့် **JOIN** keyword ရဲ့ ရှေ့မှာရေးတဲ့ဟာကို left table နောက်မှာရေးတာကို right table လို့ မှတ်ရင်ရပါပြီ။

```
SELECT books.title, borrow_records.borrower_name,
borrow_records.borrow_date FROM books LEFT JOIN borrow_records ON
books.book_id = borrow_records.book_id;
```

ဒီ query မှာတော့ ရှေ့ကတည်ဆောက်ထားတဲ့ **books** နဲ့ **borrow_records** ဆိုတဲ့ table နှစ်ခုကို **LEFT JOIN** နဲ့ ချိတ်ဆက်ထားပါတယ်။ **JOIN** ကို သုံးထားတာကြောင့် **books** table ထဲက စာအုပ်အားလုံးကို ပြသမှာဖြစ်ပြီး ငှားထားတဲ့အချက်အလက်ရှိတဲ့ စာအုပ်တွေအတွက်သာ **borrow_records** table ထဲက data တွေပါ ပါလာမယ်။ ငှားမထားသေးတဲ့ စာအုပ်တွေအတွက်တော့ **borrower_name** နဲ့ **borrow_date** column တွေမှာ **NULL** တန်ဖိုးပဲ ပြသမှာ ဖြစ်ပါတယ်။

```
school_db=# SELECT books.title, borrow_records.borrower_name, borrow_records.borrow_date
school_db=# FROM books
school_db=# LEFT JOIN borrow_records ON books.book_id = borrow_records.book_id;
```

title	borrower_name	borrow_date
Myanmar Language Grade 1	Maung Aung	2025-05-01
Myanmar Language Grade 1	Mi Mi Khine	2025-05-02
Mathematics Guide	Soe Moe	2025-05-03
Introduction to Science		
History Book		

(5 rows)

2.3 RIGHT JOIN

RIGHT JOIN ကိုတော့ **LEFT JOIN** နဲ့ ပြောင်းပြန်လို့သာ မှတ်ထားရင် လုံလောက်ပါပြီ။ left table နေရာမှာ right table ကို အစာထိုးလိုက်ရင် အဆင်ပြေပါပြီ။

```
SELECT books.title, borrow_records.borrower_name,
borrow_records.borrow_date FROM books RIGHT JOIN borrow_records ON
books.book_id = borrow_records.book_id;
```

ရလဒ်ကို ဒီနေရာမှာ ထပ်မံပြောပြပါအောင်။ မိမိကိုယ်ကို စမ်းကြည့်နိုင်ပါတယ်။ ငှားထားတဲ့ record မရှိတဲ့ စာအုပ်တွေကိုတော့ ဘာမှပြပေးမှာ မဟုတ်ပါဘူး။ လော့လော့ဆယ်အားဖြင့်တော့ **INNER JOIN** တုန်းက ရလဒ်နဲ့တူနေမှာတော့ ဖြစ်ပါတယ်။

2.4 FULL JOIN

FULL JOIN သည် table နှစ်ခုလုံးမှ data records အားလုံးကို ထုတ်ယူပြီး ကိုက်ညီမှုမရှိရင်တော့ **NULL** ဖြစ်နေမှာပါ။

```
SELECT books.title, borrow_records.borrower_name,
borrow_records.borrow_date FROM books FULL JOIN borrow_records ON
books.book_id = borrow_records.book_id;
```

```
school_db=# SELECT books.title, borrow_records.borrower_name, borrow_records.borrow_date
school_db=# FROM books
school_db=# FULL JOIN borrow_records ON books.book_id = borrow_records.book_id;
      title      | borrower_name | borrow_date
-----+-----+-----
 Myanmar Language Grade 1 | Maung Aung   | 2025-05-01
 Myanmar Language Grade 1 | Mi Mi Khine  | 2025-05-02
 Mathematics Guide      | Soe Moe     | 2025-05-03
 Introduction to Science |              |
 History Book          |              |
(5 rows)
```

2.5 SELF JOIN for Same-Table Relationships

SELF JOIN ဆိုတာကတော့ table တစ်ခုတည်းကို ကိုယ့်ဟာကိုယ် **JOIN** လုပ်တာပါ။ အခြား table မလိုဘဲ အဲဒီ table ထဲမှာရှိတဲ့ records တွေကို သူ့ရဲ့ ကိုယ်ပိုင် relationship အရ ချိတ်ဆက်ပေးတာပါ။

ဒါကတော့ နမူနာ syntax ပါ။ ဆက်လက်ရှင်းပြပါမယ်။

```
SELECT column_name(s) FROM table_name AS t1 INNER JOIN table_name AS t2
ON t1.column_name = t2.column_name;
```

အောက်ပါ table record data တွေကိုတော့ ကိုယ်တိုင်ရေးကြည့်ပါ။

```
[company_db=# SELECT * FROM customers;
 customer_id | customer_name | age | city      | balance
-----+-----+-----+-----+-----
          1 | John          | 30  | Yangon    | 3000.00
          2 | Mary          | 27  | Mandalay  | 2500.00
          3 | Peter         | 22  | Naypyidaw | 3500.00
          4 | Lisa          | 27  | Taunggyi  | 2500.00
(4 rows)
```

```
SELECT t1.customer_name AS customer1, t2.customer_name AS customer2,
t1.balance
FROM customers AS t1
```

```
INNER JOIN customers AS t2
ON t1.balance = t2.balance
AND t1.customer_id < t2.customer_id;
```

t1 နဲ့ **t2** ဆိုတာကတော့ **customers** table ကို နှစ်ခုပြန်ခွဲပြီး ယာယီ alias နာမည်တွေ တပ်ပေးလိုက်တာပါ။ **t1.balance = t2.balance** က condition ဖြစ်ပြီး လက်ကျန်ငွေ တူညီတဲ့ customer တွေကို ရှာဖွေဖို့ပါ။ **t1.customer_id < t2.customer_id** ကို ထပ်ထည့်ထားတာကတော့ customer တစ်ဦးချင်းစီက သူ့ဟာသူ တူညီနိုင်တာမို့ ထပ်ပြီး result ထဲမှာ မပါလာအောင်လို့ပါ။ ဒီ query ရလဒ်မှာ လက်ကျန်ငွေ 2500.00 ရှိတဲ့ **Mary** နဲ့ **Lisa** တွဲပေါ်လာမှာ ဖြစ်ပါတယ်။

```
company_db=# SELECT t1.customer_name AS customer1, t2.customer_name AS customer2, t1.balance
company_db=# FROM customers AS t1
company_db=# INNER JOIN customers AS t2
company_db=# ON t1.balance = t2.balance
company_db=# AND t1.customer_id < t2.customer_id;
 customer1 | customer2 | balance
-----+-----+-----
    Mary   |    Lisa   | 2500.00
(1 row)
```

SELECT နောက်မှာပါတဲ့ column တွေကတော့ query ရလဒ်ထဲမှာ **t1** ရဲ့ **customer_name** ကို **customer1** လို့ ဖော်ပြမယ်။ **t2** ရဲ့ **customer_name** ကို **customer2** လို့ ဖော်ပြမယ်ဆိုတဲ့ အဓိပ္ပါယ်ပါ။ ပြီးတော့ **t1.balance** ကိုတစ်ခုတည်းပဲ ရွေးသုံးထားတာက **t1.balance = t2.balance** ဆိုတဲ့ condition ကြောင့်ဖြစ်ပါတယ်။ **balance** တန်ဖိုးက **t1** နဲ့ **t2** နှစ်ဖက်လုံးမှာ တူတာမို့ **t1.balance** (သို့) **t2.balance** တစ်ခုတည်းကို အသုံးပြုတာနဲ့တင် အဆင်ပြေပါတယ်။

2.6 CROSS JOIN

CROSS JOIN ဟာ row တွေအားလုံးကို တွဲပေးတဲ့ join အမျိုးအစားတစ်ခုပါ။ **CROSS JOIN** မှာ condition မလိုအပ်ပါဘူး။ ဆိုလိုတာက တစ်ဖက် table ရဲ့ row တစ်ခုနဲ့ တခြား table ရဲ့ row အကုန်လုံး ကို တူတူပေါင်းပေးတာပါ။ ဒါကြောင့် ဖြစ်နိုင်သမျှ row အားလုံးကို ထုတ်ပေးမှာပါ။

```
SELECT books.title, borrow_records.borrower_name, borrow_date
FROM books
CROSS JOIN borrow_records;
```


3. Practical Examples of Joins

3.1 Combining Data from Multiple Tables

ဒီတစ်ခါတော့ table နှစ်ခုထက် ပိုပြီး data တွေကို ပေါင်းစပ်ကြည့်ကြရအောင်ပါ။ ရှေ့မှာ တည်ဆောက်ခဲ့တဲ့ **books** နဲ့ **borrow_records** table တွေကို ဒီနေရာမှာ ဆက်လက်အသုံးပြုမှာပါ။ ပြီးတော့ **readers** table တစ်ခုကို ထပ်တိုးတည်ဆောက်ပါမယ်။

```
[school_db=# SELECT * FROM readers;
 reader_id | reader_name | reader_email
-----+-----+-----
          1 | Maung Aung | maungaung@example.com
          2 | Mi Mi Khine | mimikhine@example.com
          3 | Soe Moe    | soemoe@example.com
          4 | Aye Aye    | ayeaye@example.com
(4 rows)
```

INNER JOIN နှစ်ခါသုံးထားတာကို သတိပြုပါ။

```
SELECT books.title, readers.reader_name, readers.reader_email,
borrow_records.borrow_date
FROM books
INNER JOIN borrow_records ON books.book_id = borrow_records.book_id
INNER JOIN readers ON borrow_records.borrower_name =
readers.reader_name;
```

ရလဒ်က အခုလို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။

```
school_db=# SELECT books.title, readers.reader_name, readers.reader_email, borrow_records.borrow_date
school_db=# FROM books
school_db=# INNER JOIN borrow_records ON books.book_id = borrow_records.book_id
[school_db=# INNER JOIN readers ON borrow_records.borrower_name = readers.reader_name;
 title | reader_name | reader_email | borrow_date
-----+-----+-----+-----
 Myanmar Language Grade 1 | Maung Aung | maungaung@example.com | 2025-05-01
 Myanmar Language Grade 1 | Mi Mi Khine | mimikhine@example.com | 2025-05-02
 Mathematics Guide | Soe Moe | soemoe@example.com | 2025-05-03
(3 rows)
```

books နဲ့ **borrow_records** ကို **JOIN** တဲ့အခါမှာတော့ **book_id** 1 နဲ့ 2 အတွက်ပဲ ကိုက်ညီတဲ့အတွက် စုစုပေါင်း row သုံးခုပဲ ရလာပါတယ်။

ပြီးတော့ **borrow_records** နဲ့ **readers** ကို **JOIN** တဲ့အခါမှာလည်း **borrower_name** နဲ့ **reader_name** တို့ကိုက်ညီတဲ့ Maung Aung, Mi Mi Khine, Soe Moe တို့ပဲ ထွက်လာပါတယ်။ Aye Aye က **borrow_records** မှာ မပါတဲ့အတွက် result မှာ ပါလာမှာမဟုတ်ပါဘူး။

နောက်တစ်ခုကတော့ **books** table ထဲက **book_id** 3 (History Book) နဲ့ 4 (Introduction to Science) ဆိုတဲ့ စာအုပ်တွေလည်း **borrow_records** မှာ မပါတဲ့အတွက် result ထဲမှာ ပါမလာဘူးဆိုတာ သတိပြုရမှာပါ။

3.2 Debugging Join Queries

ဘယ်ဟာမဆို သူ့အားသာချက်၊ အာနည်းချက်တွေ ရှိတာမလို့ **JOIN** query တွေ အသုံးပြုတဲ့အခါ တချို့ပြဿနာလေးတွေတော့ ရှိပါတယ်။ Error လေးတွေ တွေ့ရင်တော့ ဒီအကြံပြုချက်လေးတွေကို သိထားတာက အသုံးဝင်မယ်ထင်ပါတယ်။

/* **ON** clause မှာသုံးထားတဲ့ column တွေဟာ တကယ်ပဲ table တွေအကြား ဆက်နွယ်မှုရှိစေဖို့ အရေးကြီးတဲ့ primary key နဲ့ foreign key တွေဖြစ်ရဲ့လားဆိုတာကို သေချာစစ်ဆေးဖို့လိုပါတယ်။ */

/* **JOIN** မလုပ်ခင်မှာတော့ table တစ်ခုချင်းစီကို သီးသန့် **SELECT** query နဲ့ စမ်းကြည့်ဖို့လိုပါတယ်။ မသေချာဘဲ **JOIN** လုပ်လိုက်ရင် error တွေ ဖြစ်ပေါ်လာနိုင်ပါတယ်။ */

/* တကယ်လို့ SQL error message တွေလာရင်လည်း အဲ့ message ကို သေချာဖတ်ပြီး ပြဿနာကို အဖြေရှာတာက ပိုထိရောက်ပါတယ်။ */

/* ပထမဆုံးမှာတော့ **INNER JOIN** လို ရိုးရှင်းတဲ့ **JOIN** နဲ့ စမ်းကြည့်ပါ။ အဲဒီကနေ သေချာပြီဆိုတော့မှ လိုအပ်သလို **LEFT JOIN**, **RIGHT JOIN** နဲ့ တိုးချဲ့အသုံးပြုသင့်ပါတယ်။ */

/* နောက်ဆုံးသတိပြုစရာလေးက **NULL** တန်ဖိုးတွေကို သတိထားပါ။ **LEFT JOIN** သို့မဟုတ် **RIGHT JOIN** အသုံးပြုတဲ့အခါ match မဖြစ်တဲ့ rows တွေကြောင့် **NULL** တန်ဖိုးတွေ ပါလာတတ်ပါတယ်။ အဲ့တာကြောင့် **NULL** တန်ဖိုး ပါဝင်နိုင်တဲ့ column တွေဆို သတိပြုပါ။ */

4. Exercises

အောက်မှာပေးထားတဲ့ table နှစ်ခုကို ကိုယ်တိုင်တည်ဆောက်ကြည့်ပြီး လေ့ကျင့်ခန်းတွေ လုပ်ကြည့်ပါ။
အဲ့တာမှလည်း ပိုနားလည်လာမှာပါ။

// Source Code Link:

<https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-5/12-create-and-insert-customers.sql>

```
[exercises_db=# SELECT * FROM customers;
 customer_id | customer_name | phone
-----+-----+-----
          1 | John Smith   | 09-11111111
          2 | Mary Johnson | 09-22222222
          3 | Michael Anderson | 09-33333333
(3 rows)
```

// Source Code Link:

<https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-5/13-create-and-insert-orders.sql>

```
[exercises_db=# SELECT * FROM orders;
 order_id | customer_id | product | order_date
-----+-----+-----+-----
          1 |          1 | rice    | 2025-05-01
          2 |          1 | oil     | 2025-05-02
          3 |          2 | sugar   | 2025-05-03
(3 rows)
```

Question 1:

Join **customers** and their **orders** using **INNER JOIN**, and retrieve customer name, product, and order date.

Question 2:

Retrieve all **customers** and join their **orders** using **LEFT JOIN**. For customers without orders, product and order date should be **NULL**.

Question 3:

Join all **orders** with **customers** using **RIGHT JOIN**.

Question 4:

Join **customers** and **orders** using **CROSS JOIN** and retrieve all possible combinations.

Question 5:

Write a SQL query using a **SELF JOIN** on the **orders** table to find all pairs of orders made by the same customer on different dates.

"Chapter 5" အဆုံးကို ရောက်ရှိလာပြီဆိုတော့ SQL အခြေခံတွေကို သေချာနားလည်သလောက် ဖြစ်နေပြီလို့ ယူဆပါတယ်။ နောက်အခန်းမှာတော့ data တွေကို ဘယ်လို aggregate လုပ်မလဲဆိုတာနဲ့ ပတ်သက်ပြီး ဆက်လက်လေ့လာသွားကြပါမယ်။ အထူးသဖြင့် **SUM**, **COUNT**, **AVG**, **MIN**, **MAX** စတဲ့ aggregate functions တွေအကြောင်းပါ။

Chapter 6: Aggregating Data

ယေဘုယျအားဖြင့် ပြောရရင် aggregation ဆိုတာက မတူညီတဲ့ objects တွေကို single entity (တစ်စုတစ်စည်း) အဖြစ် စုပေါင်းပေးတာပဲ ဖြစ်ပါတယ်။ SQL မှာတော့ table တစ်ခုထဲမှာရှိတဲ့ column တစ်ခုရဲ့ records အားလုံးကို တစ်ခုတည်းအဖြစ် စုပေါင်းထုတ်ပေးနိုင်တဲ့ aggregate functions တွေ ရှိပြီးသားဖြစ်ပါတယ်။

1. Introduction to Aggregate Functions

Aggregate function တွေကတော့ **SUM()**, **AVG()**, **COUNT()**, **MIN()**, **MAX()** စတဲ့ function တွေဖြစ်ပါတယ်။ နားလည်လွယ်အောင် ပြောရရင် ဒီ function တွေဟာ column တစ်ခုထဲမှာရှိတဲ့ row အားလုံးကို စုပေါင်းတွက်ချက်ပြီး result တစ်ခုတည်းပဲ ပြန်ထုတ်ပေးတာပါ။

1.1 COUNT, SUM, AVG, MIN, MAX

COUNT

COUNT ဆိုတာက row တွေရဲ့ အရေအတွက်ကို ရေတွက်ဖို့သုံးတဲ့ function ပါ။ ဥပမာ - ဆိုင်တစ်ခုမှာ တနေ့တာအတွင်း order ဘယ်နှစ်ခုရှိတယ်ဆိုတာ စစ်ချင်ရင် **COUNT()** ကိုသုံးလို့ရပါတယ်။

ဒီနေရာမှာတော့ အခန်း (၄) မှာ သုံးခဲ့တဲ့ **orders** table ကို ပြန်အသုံးပြုသွားမှာပါ။ [တစ်ခုရှိတာတော့ amount ဆိုတဲ့ column အသစ်ကို ထပ်တိုးထားပါတယ်။](#)

```
SELECT COUNT(*) AS total_orders FROM orders;
```

ဒီ query ကို run မယ်ဆိုရင် order မှာထားတဲ့အရေအတွက် သုံးခုရှိတယ်ဆိုတာ လွယ်လွယ်နဲ့ သိနိုင်ပါတယ်။

```
[exercises_db=# SELECT COUNT(*) AS total_orders FROM orders;
total_orders
-----
3
(1 row)
```

SUM

စုစုပေါင်းတန်ဖိုးတွေကို တွက်တဲ့နေရာမှာဆို **SUM** keyword ကို သုံးနိုင်ပါတယ်။

```
SELECT SUM(amount) AS total_amount FROM orders;
```

amount ဆိုတဲ့ column ထဲမှာရှိတဲ့ တန်ဖိုးအားလုံးကို ပေါင်းမှာဖြစ်တဲ့အတွက် **SUM** လို့ရေးပြီး parentheses () အထဲမှာ **amount** ဆိုတဲ့ column name ရေးပေးရတာကို သတိပြုပါ။

```
[exercises_db=# SELECT SUM(amount) AS total_amount FROM orders;
total_amount
-----
          52000
(1 row)
```

AVG

AVG က column တစ်ခုရဲ့ ပျမ်းမျှတန်ဖိုးကို တွက်တဲ့ function ပါ။ ဥပမာအနေနဲ့ order တစ်ခုချင်းစီရဲ့ ပျမ်းမျှတန်ဖိုးကို သိချင်ရင် **AVG** ကို သုံးနိုင်ပါတယ်။ ဒီမှာတော့ result တွေကို အသေးစိတ် မပြတော့ပါဘူး။ မိမိကိုယ်တိုင် SQL shell မှာ စမ်းကြည့်ပါ။

```
SELECT AVG(amount) AS avg_amount FROM orders;
```

MIN

Minimize ကနေလာတဲ့ အနည်းတန်ဖိုးကိုရှာဖို့ **MIN** keyword ပါ။ အထူးတလည် ထပ်မရှင်းပြတော့ပါဘူး။

```
SELECT MIN(amount) AS min_amount FROM orders;
```

MAX

အဲဒီလိုပဲ **MAX** ကလည်း အများဆုံးတန်ဖိုးကို ရှာဖို့ သုံးတဲ့ keyword ဖြစ်ပါတယ်။

```
SELECT MAX(amount) AS max_amount FROM orders;
```

1.2 Use Cases for Each Function

အပေါ်မှာအသုံးပြုခဲ့တဲ့ aggregate function တွေကို လိုတိုရှင်း ဖော်ပြပေးလိုက်ပါတယ်။

COUNT: မှတ်တမ်းအရေအတွက် သိချင်တဲ့အခါ။ ဥပမာ - ဆိုင်မှာ order စုစုပေါင်း ဘယ်လောက်ရှိလဲ။

SUM: စုစုပေါင်းပမာဏ တွက်ချင်တဲ့အခါ။ ဥပမာ - တစ်လအတွင်း ရောင်းရပမာဏ ဘယ်လောက်လဲ။

AVG: ပျမ်းမျှတန်ဖိုး တွက်ချင်တဲ့အခါ။ ဥပမာ - ကျောင်းသားတွေရဲ့ ပျမ်းမျှအမှတ် ဘယ်လောက်လဲ။

MIN: အနည်းဆုံးတန်ဖိုး ရှာချင်တဲ့အခါ။ ဥပမာ - အနည်းဆုံးဈေးနဲ့ ရောင်းတဲ့ ပစ္စည်း။

MAX: အများဆုံးတန်ဖိုး ရှာချင်တဲ့အခါ။ ဥပမာ - အမှတ်အများဆုံး ရရှိတဲ့ ကျောင်းသား။

2. GROUP BY Clause

GROUP BY ဟာဖြင့်ဆိုရင် data ကို တစ်ခု (သို့) တစ်ခုထက်ပိုတဲ့ column တွေအလိုက် အုပ်စုခွဲပေးတာပါ။ Aggregate function တွေနဲ့ တွဲပြီးသုံးရတာ များပါတယ်။ ဖောက်သည်တစ်ဦးချင်းစီအတွက် သူ့မှာထားတဲ့ order စုစုပေါင်းတန်ဖိုးကို တွက်ကြမယ်ဆိုပါစို့။

```
SELECT customer_id, SUM(amount) AS total_order_value FROM orders GROUP BY customer_id;
```

ဒီ query က **orders** table ထဲမှာရှိတဲ့ ဖောက်သည်တစ်ဦးစီရဲ့ **customer_id** အလိုက် order မှာထားတဲ့ **amount** တွေကို စုစုပေါင်းတွက်ပြီး **total_order_value** ဆိုတဲ့ နာမည်နဲ့ ပြပေးလိုက်တာပါ။

```
exercises_db=# SELECT customer_id, SUM(amount) AS total_order_value
exercises_db=# FROM orders
exercises_db=# GROUP BY customer_id;
 customer_id | total_order_value
-----+-----
           2 |          12000
           1 |          40000
(2 rows)
```


3. HAVING Clause

GROUP BY ကနေ သက်သက်စီဖြစ်အောင် ခွဲထုတ်ထားတဲ့ data တွေကို ထပ်ပြီး filter လုပ်ဖို့ဆိုရင် **HAVING** ကို သုံးလို့ရပါတယ်။ ဥပမာ - စုစုပေါင်း order တန်ဖိုးကို တွက်ပြီးနောက် 20000 ထက်ပိုပြီး order မှာထားတဲ့ ဖောက်သည်ကို တိတိကျကျ သိချင်တယ်ဆိုပါစို့။

3.1 Filtering Grouped Data

```
SELECT customer_id, SUM(amount) AS total_order_value FROM orders GROUP
BY customer_id HAVING SUM(amount) > 20000;
```

Order စုစုပေါင်းတန်ဖိုး 20000 ထက်များတာ **customer_id** 1 တစ်ယောက်ပဲ ဖြစ်ပါတယ်။

```
exercises_db=# SELECT customer_id, SUM(amount) AS total_order_value
exercises_db=# FROM orders
exercises_db=# GROUP BY customer_id HAVING SUM(amount) > 20000;
 customer_id | total_order_value
-----+-----
          1 |          40000
(1 row)
```

3.2 HAVING vs. WHERE

HAVING နဲ့ **WHERE** ဘာကွာလဲဆိုတာကို အရိုးရှင်းဆုံး ရှင်းပြရမယ်ဆိုရင် **WHERE** က **GROUP BY** မတိုင်ခင် မှာဘဲ ကြို filter လုပ်ပါတယ်။ **HAVING** ကျတော့ **GROUP BY** ပြီးတဲ့နောက်မှာ filter လုပ်ပါတယ်။

```
SELECT customer_id, SUM(amount) AS total_amount
FROM orders
WHERE order_date ≥ '2025-05-02'
GROUP BY customer_id
HAVING SUM(amount) > 10000;
```

WHERE order_date ≥ '2025-05-02' ဆိုတာကတော့ 2025-05-02 နေ့မှစပြီး လက်ခံရရှိတဲ့ orders တွေကိုသာ ဖော်ပြဖို့ filtered condition တစ်ခုပါ။ **GROUP BY customer_id** ကတော့ orders တွေကို customer တစ်ဦးစီအလိုက် အုပ်စုခွဲပေးတာဖြစ်ပါတယ်။ အဲ့ဒါပြီးမှ **SUM(amount)** ဟာ customer တစ်ယောက်ချင်းစီရဲ့ order မှာထားတဲ့ စုစုပေါင်းပမာဏကို တွက်ချက်ပေးတာဖြစ်ပါတယ်။ နောက်ဆုံး **HAVING SUM(amount) > 10000** ကတော့ **GROUP BY** နဲ့ အုပ်စုခွဲပြီးသား customer

တွေထဲက စုစုပေါင်း order တန်ဖိုး 10,000 ကျော်တဲ့သူတွေကိုပဲ ထပ်စစ်ထုတ်ပေးတာပါ။ ရလဒ်က အခုလို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။

```

[exercises_db=# SELECT customer_id, SUM(amount) AS total_amount
[exercises_db=# FROM orders
[exercises_db=# WHERE order_date >= '2025-05-02'
[exercises_db=# GROUP BY customer_id
[exercises_db=# HAVING SUM(amount) > 10000;
 customer_id | total_amount
-----+-----
          2 |         12000
          1 |         15000
(2 rows)

```

4. Nested Aggregations

Nested Aggregations ဆိုတာက aggregation function တစ်ခုအတွင်းမှာ နောက်ထပ် aggregation function တစ်ခုကို ထပ်ထည့်သုံးတာပါ။ ဥပမာ - ပထမအဆင့်မှာ customer တစ်ယောက်ချင်းစီရဲ့ order အရေအတွက်ကို **COUNT()** နဲ့တွက်မယ်။ အဲ့တာပြီးရင် သူတို့အကြားမှာ order အရေအတွက် အများဆုံးရှိ တဲ့ customer ကို **MAX()** နဲ့ထပ်ပြီး ရှာမယ်ဆိုရင် ဒါက nested aggregation ဖြစ်သွားပါတယ်။

```

SELECT MAX(order_count)
FROM (
    SELECT customer_id, COUNT(*) AS order_count
    FROM orders
    GROUP BY customer_id
) AS customer_orders;

```

4.1 Aggregates Within Aggregates

() ထဲကဟာက nested aggregation ပါ။

```

SELECT customer_id, COUNT(*) AS order_count
FROM orders

```

```
GROUP BY customer_id;
```

ဒီ query မှာဆိုရင်တော့ **orders** table ထဲက data တွေကို **customer_id** အလိုက် **GROUP BY** လုပ်ပြီး ဖောက်သည်တစ်ယောက်ချင်းစီအတွက် order အရေအတွက်ကို **COUNT(*)** နဲ့ အုပ်စုတစ်ခုချင်းစီအလိုက် တွက်ပေးတာပဲ ဖြစ်ပါတယ်။

```
exercises_db=# SELECT MAX(order_count)
exercises_db=# FROM (
exercises_db(#      SELECT customer_id, COUNT(*) AS order_count
exercises_db(#      FROM orders
exercises_db(#      GROUP BY customer_id
[exercises_db(# ) AS customer_orders;
max
-----
2
(1 row)
```

4.2 Practical Examples (e.g., Top-N Queries)

Top-N queries ဆိုတာကတော့ result တွေထဲက အမြင့်ဆုံး (သို့) အနိမ့်ဆုံး **N** ခုကိုပဲ ရွေးထုတ်ဖော်ပြချင်တဲ့ SQL query တစ်ခုပဲ ဖြစ်ပါတယ်။ ဥပမာ - order တန်ဖိုးအများဆုံး customer 2 ယောက်ကိုရှာမယ် ဆိုရင်တော့ ဒီလို **Top-N** query ကိုသုံးလို့ရပါတယ်။

```
SELECT customer_id, SUM(amount) AS total_amount
FROM orders
GROUP BY customer_id
ORDER BY total_amount DESC
LIMIT 2;
```

ဒီ query မှာ **total_amount** တွေကို ကြီးစဉ်ငယ်လိုက် စီပြီးနောက်မှာ **LIMIT 2** ဆိုတာကြောင့် အများဆုံး order တန်ဖိုးရှိတဲ့ ဖောက်သည်နှစ်ဦးကိုပဲ ရယူထားတာဖြစ်ပါတယ်။ ဒီလို top n (ဒီမှာတော့ top 2) ကိုရှာတဲ့ query ကိုပဲ **Top-N** query လို့ ခေါ်တာပဲ ဖြစ်ပါတယ်။

```

exercises_db=# SELECT customer_id, SUM(amount) AS total_amount
exercises_db=# FROM orders
exercises_db=# GROUP BY customer_id
exercises_db=# ORDER BY total_amount DESC
exercises_db=# LIMIT 2;
 customer_id | total_amount
-----+-----
            1 |         40000
            2 |         12000
(2 rows)

```

5. Common Pitfalls and Best Practices

5.1 Avoiding Ambiguous Column References

Aggregate functions တွေကို အသုံးပြုတဲ့အခါမှာ **GROUP BY** ထဲမှာ မထည့်ထားတဲ့ column ကို **SELECT** မှာ ထည့်ခေါ်လို့ မရပါဘူး။ အဲဒါက error ဖြစ်စေနိုင်ပါတယ်။ တစ်နည်းအားဖြင့် **SELECT** ထည့်ခေါ်ချင်ရင် **GROUP BY** မှာလည်း ထည့်ပေးရမှာ ဖြစ်ပါတယ်။

// Incorrect Syntax

```

SELECT customer_id, order_date, SUM(amount) AS total_amount FROM orders
GROUP BY customer_id;

```

ဒီ query မှာ error ဖြစ်ရတဲ့ အကြောင်းရင်းကတော့ **SELECT** မှာ **customer_id** နဲ့ **order_date** နှစ်ခုလုံးကို ဖော်ပြထားတယ်။ ဒါပေမဲ့ **GROUP BY** နောက်မှာ **customer_id** တစ်ခုပဲ ထည့်ထားတာဖြစ်တဲ့ အတွက် **order_date** ဟာ aggregate function မဟုတ်တဲ့ column တစ်ခုအနေနဲ့ **GROUP BY** ထဲမှာ မပါလို့ error ဖြစ်လာတာပါ။ ဆိုလိုတာက **order_date** ကိုပါ အုပ်စုထဲပါချင်ရင် အုပ်စုဝင်ဖြစ်ရမယ် ဆိုတဲ့ သဘောပါ။

SQL မှာ **GROUP BY** ကို သုံးတဲ့အခါမှာ **SELECT** မှာဖော်ပြမယ့် column တွေကို စဉ်းစားရပါတယ်။ အထူး သဖြင့် aggregate မဟုတ်တဲ့ column တွေကို **GROUP BY** ထဲမှာ ထည့်ပေးဖို့လိုပါတယ်။

// Correct Syntax

ပုံစံအမှန်ကတော့ အခုလို ရေးရမှာပဲဖြစ်ပါတယ်။

```
SELECT customer_id, order_date, SUM(amount) AS total_amount FROM orders
GROUP BY customer_id, order_date;
```

5.2 Optimizing Aggregate Queries

Aggregate queries တွေ ရေးတဲ့အခါ သတိထားရမယ့်အချက်လေးတွေပါ။

ရှေ့အခန်းတွေမှာ ပြောခဲ့သလိုပါပဲ။ လိုအပ်တဲ့ column တွေကိုပဲ ထုတ်ယူပါ။ မလိုအပ်တဲ့ column တွေပါ အကုန်ထုတ်မယ်ဆိုရင် performance ကျမှာပါ။ GROUP BY နှင့် JOIN တွင်အသုံးပြုတဲ့ column တွေ (ဥပမာ - **customer_id, book_id**) မှာ အညွှန်း (**INDEX**) များကို တစ်ခါတည်း ဖန်တီးယူပါ။ ဒီလိုပါ -

```
CREATE INDEX idx_customer_id ON orders(customer_id);
```

INDEX ပေးထားလိုက်ရင် query run ရတာ ပိုမြန်ပါတယ်။ ပြီးတော့ ရှုပ်ထွေးတဲ့ query တွေ တန်းမရေး ခင်မှာ လွယ်ကူရိုးရှင်းတဲ့ query နဲ့ အစပြုတာ ကောင်းပါတယ်။ ပြီးမှ တဖြည်းဖြည်း extend လုပ်ပါ။

6. Exercises

အောက်မှာပေးထားတဲ့ table နှစ်ခုကိုတော့ ထိုးစံအတိုင်း ကိုယ်တိုင်တည်ဆောက်ကြည့်ပြီး exercise question တစ်ခုချင်းစီကို သေချာလုပ်ကြည့်စေချင်ပါတယ်။ လက်တွေ့လုပ်မှ ပိုမိုနားလည်မှတ်မိလာမှာပါ။

// Source Code Link:

<https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-6/11-create-and-insert-students.sql>

```
[exercises_db=# SELECT * FROM students;
 student_id | student_name | class
-----+-----+-----
          1 | Ko Ko       | Grade5
          2 | Nyi Nyi     | Grade7
          3 | Su Lay      | Grade5
          4 | Hnin Eain   | Grade7
(4 rows)
```

// Source Code Link:

<https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-6/12-create-and-insert-scores.sql>

```
[exercises_db=# SELECT * FROM scores;
 score_id | student_id | subject | score | exam_date
-----+-----+-----+-----+-----
          1 |          1 | Myanmar |    85 | 2025-05-01
          2 |          1 | Math    |    90 | 2025-05-01
          3 |          2 | Myanmar |    78 | 2025-05-02
          4 |          3 | Math    |    92 | 2025-05-01
          5 |          4 | Myanmar |    80 | 2025-05-02
(5 rows)
```

Question 1:

Find the total number of records in the exam **scores** table using **COUNT**.

Question 2:

Calculate the total score for each student using **SUM** and **GROUP BY**.

Question 3:

Find students whose average score is greater than **80** using **AVG** and **HAVING**.

Question 4:

Find the highest score for each subject using **MAX**.

Question 5:

Find the top two students with the highest scores using a **Top-N** query.

Chapter 7: Subqueries and Nested Queries

SQL မှာ subquery ဆိုတာက main query တစ်ခုအတွင်းမှာ ထပ်ဆင့်ရေးသားထားတဲ့ query တစ်ခုပဲ ဖြစ်ပါတယ်။ ဒီအခန်းမှာတော့ subquery ရဲ့ အဓိပ္ပါယ်၊ ဘယ်လိုအသုံးပြုကြလဲ၊ ဘယ်လိုအမျိုးအစားတွေ ရှိလဲ ဆိုတာတွေနဲ့အတူ Joins နဲ့ကွဲ ဘယ်လိုကွဲပြားသလဲဆိုတာကိုပါ တပါတည်း နားလည်သွားအောင် ရှင်းပြသွားမှာ ဖြစ်ပါတယ်။ ဒါ့အပြင် performance ပိုင်းမှာ သတိထားစရာအချက်တွေ၊ လိုက်နာသင့်တဲ့ အကြံပြုချက်တွေကိုလည်း လိုရင်းတိုရှင်းနဲ့ လက်တွေ့အသုံးဝင်အောင် ဖော်ပြပေးသွားပါမယ်။

1. What are Subqueries?

1.1 Definition and Purpose

အလွယ်ဆုံးပြောရရင် subquery ဆိုတာက outer query တစ်ခုအတွင်းမှာ ထပ်ထည့်ရေးထားတဲ့ SQL inner query တစ်ခုပဲ ဖြစ်ပါတယ်။ အဓိကရည်ရွယ်ချက်ကတော့ data တွေကို စစ်ထုတ်ဖို့၊ တန်ဖိုးတွေ တွက်ချဖို့၊ ဒါမှမဟုတ် ရှုပ်ထွေးနေတဲ့ query တွေကို ပိုတိကျအောင် အလွယ်တကူ ဖြေရှင်းဖို့ပါ။ ရှေ့အခန်း မှာလည်း subquery တစ်ခုကို nested aggregation အနေနဲ့ သုံးခဲ့တာကို သတိထားမိမှာပါ။

1.2 Subquery Placement (SELECT, WHERE, FROM)

ထိုးစံအတိုင်းပါပဲ။

- **SELECT** ဆိုတာ column တစ်ခုရဲ့ တန်ဖိုးတွေကို ရွေးချယ်ဆွဲထုတ်ယူဖို့ပါ။
- **WHERE** ဆိုတာကတော့ data တွေထဲက မလိုအပ်တဲ့ row တွေကို စိစစ်ဖယ်ထုတ်ဖို့၊ condition တွေနဲ့ filter လုပ်ဖို့ပါ။
- **FROM** နောက်မှာပါတဲ့ subquery ဆိုတာ temporary table တစ်ခုအဖြစ် ထုတ်ပေးဖို့ပါ။

ရှေ့အခန်းမှာ သုံးခဲ့တဲ့ **orders** နဲ့ **customers** table နှစ်ခုကိုပဲ ပြန်အသုံးပြုမှာ ဖြစ်ပါတယ်။ နမူနာ query ကိုကြည့်ကြရအောင်ပါ။

```
SELECT  
    customers.customer_name,
```

```
(
    SELECT COUNT(*)
    FROM orders
    WHERE orders.customer_id = customers.customer_id
) AS order_count
FROM customers;
```

() အထဲက subquery ကို ရှင်းပြပါမယ်။

Row တစ်ခုချင်းစီအတွက် **orders** table ထဲကိုသွားပြီး စစ်ပါတယ်။ **orders.customer_id = customers.customer_id** ဆိုတာကတော့ **orders** table ထဲက **customer_id** နဲ့ **customers** table ထဲက **customer_id** တူတဲ့ record တွေကိုသာ ရွေးယူပေးအောင်လို့ပါ။ ပြီးတော့မှ ရလာတဲ့ result table ထဲမှာ **order_count** ဆိုတဲ့ column နဲ့အတူ **customer_name** ကို တွဲပြပေးတာ ဖြစ်ပါတယ်။

ဒီလိုနည်းနဲ့ customer တစ်ယောက်ချင်းစီ order ဘယ်နှစ်ခေါက်မှာထားတယ်ဆိုတာကို လွယ်လွယ်နဲ့ သိနိုင်ပါတယ်။

```
exercises_db=# SELECT customers.customer_name,
exercises_db=# (SELECT COUNT(*) FROM orders WHERE orders.customer_id = customers.customer_id) AS order_count
exercises_db=# FROM customers;
 customer_name | order_count
-----
John Smith    |          2
Mary Johnson  |          1
Michael Anderson |          0
(3 rows)
```

ဒီတစ်ခါ **WHERE** နောက်မှာ subquery ကို ရေးထားပါတယ်။

```
SELECT product, amount FROM orders WHERE amount > (SELECT AVG(amount)
FROM orders);
```

ဒီလို **WHERE** clause မှာ ရေးထားတဲ့ subquery ကို scalar subquery လို့ ခေါ်ပါတယ်။ Subquery က **amount** တွေ အားလုံးရဲ့ ပျမ်းမျှတန်ဖိုး (**AVG**) ကို ပထမဆုံးတွက်ပြီး တစ်ခုတည်းသောတန်ဖိုး (scalar value) အဖြစ် main query ဆီ ပြန်ပို့ပေးပါတယ်။

Main query **SELECT** ကတော့ **orders** table ထဲက average တန်ဖိုးထက် ကြီးတဲ့ record တွေကိုပဲ ဆွဲထုတ်ပေးတာ ဖြစ်ပါတယ်။

```

exercises_db=# SELECT product, amount FROM orders
exercises_db=# WHERE amount > (SELECT AVG(amount) FROM orders);
 product | amount
-----+-----
    rice  | 25000
(1 row)

```

```

SELECT customer_name, total_amount
FROM (
    SELECT c.customer_name, SUM(o.amount) AS total_amount
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    GROUP BY c.customer_name
) AS order_count
WHERE total_amount > 20000;

```

ဒီ query မှာ subquery ကို **FROM** နောက်မှာ အသုံးပြုထားပါတယ်။ အဲဒီ subquery က **customers** နဲ့ **orders** table နှစ်ခုကို **JOIN** လုပ်ပြီး customer တစ်ဦးစီအတွက် total order amount တွေကို **SUM()** နဲ့ တွက်ပါတယ်။ ပြီးမှ အဲဒီ subquery ကနေရလာတဲ့ result ကို temporary table တစ်ခုလို အသုံးပြုပြီး **total_amount > 20000** ဖြစ်သူတွေကို main query မှာ **WHERE** နဲ့ ထပ်စစ်ထုတ်တာ ဖြစ်ပါတယ်။ **o** နဲ့ **c** လို့ ပါတဲ့အတွက် မျက်စိရှုပ်ချင်စရာ ရှိပါတယ်။ Alias name တွေပါ။ Table name ကို အတိုကောက်ပုံစံနဲ့ အသုံးပြုလိုက်တာပါ။

2. Correlated vs. Non-Correlated Subqueries

ထပ်မံပြီးတော့ subquery တွေကို အမျိုးအစား နှစ်မျိုးခွဲခြားနိုင်ပါသေးတယ်။

2.1 Non-Correlated Subquery

ဆက်စပ်မှုမရှိတဲ့ subquery တွေကျပြန်တော့ main query နဲ့ လုံးဝမပတ်သက်ဘဲ သီးခြားအနေနဲ့ လုပ်ဆောင်နိုင်ပါတယ်။ တစ်ကြိမ်တည်းပဲ run လုပ်ပြီး result တန်ဖိုး (scalar value) တစ်ခုကို main query ဆီကို ပေးပို့ပါတယ်လို့ ရှေးမှာပြောခဲ့ပါတယ်။

```

SELECT product, amount FROM orders WHERE amount > (SELECT AVG(amount)
FROM orders);

```

စာနားက query ဖြစ်တဲ့အတွက် result က တူတူပဲရမှာ ဖြစ်ပါတယ်။

2.2 Correlated Subquery

Correlated subquery ဆိုတာက main query ထဲက record တစ်ခုချင်းစီအတွက် subquery ကို ထပ်ခါတလဲလဲ ပြန် run လုပ်ရတဲ့ subquery အမျိုးအစားပဲ ဖြစ်ပါတယ်။ Subquery ဟာ main query ထဲက တန်ဖိုးကို မှီခိုပြီးမှ အလုပ်လုပ်နိုင်တာကြောင့် main query ရဲ့ row တစ်ခုစီအပေါ် အခြေခံပြီး ရလဒ်တွေကို ပြန်ထုတ်ပေးပါတယ်။

```
SELECT c.customer_name
FROM customers c
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.customer_id = c.customer_id AND o.amount > 20000
);
```

customers table ထဲက customer တစ်ဦးချင်းစီအတွက် **orders** table ထဲမှာ သူတို့ရဲ့ order တစ်ခုခုဟာ **amount > 20000** ဖြစ်သလားဆိုတာ စစ်ပါတယ်။ တကယ်ဖြစ်နေရဲ့လားဆိုတာကို **WHERE EXISTS ()** နဲ့ စစ်တာပဲ ဖြစ်ပါတယ်။ **EXISTS** မှာပါတဲ့ subquery ထဲက result တစ်ခုခုဟာ ကိုက်ညီနေတယ်ဆိုရင် အဲဒီ condition က **TRUE** ဖြစ်သွားပါတယ်။ **SELECT 1** လို့ subquery ထဲမှာ ရေးတာက return ဖြစ်လာမယ့် value ကို အသုံးမပြုဘဲ ကိုက်ညီတဲ့ record တစ်ခုခုရှိမရှိကိုပဲ စစ်ချင်တာ ဖြစ်ပါတယ်။ **EXISTS** မှာ condition **FALSE** ဖြစ်သွားတဲ့ row တွေကိုတော့ ဖယ်ထားပြီး result ထဲမှာ မထည့်ဘူးဆိုတဲ့ သဘောပါ။

Subquery ထဲက **o.customer_id = c.customer_id** ဆိုတာကို subquery ထဲမှာ ရေးထားတာကြောင့် **customers** table ထဲက row တစ်ခုချင်းစီအတွက် subquery က တစ်ခါချင်းစီပြန်ပြန် run လုပ်ပါတယ်။ ဒါကြောင့်မို့လို့ correlated subquery လို့ ခေါ်တာပါ။

ရလဒ်က အခုလိုတွေ့ရမှာပါ။

```
exercises_db=# SELECT c.customer_name
exercises_db=# FROM customers c
exercises_db=# WHERE EXISTS (
exercises_db=#     SELECT 1
exercises_db=#     FROM orders o
exercises_db=#     WHERE o.customer_id = c.customer_id AND o.amount > 20000
exercises_db=# );
customer_name
-----
John Smith
(1 row)
```

2.3 Performance Considerations

- **Non-Correlated Subquery:** တစ်ကြိမ်တည်း လုပ်ဆောင်ပြီးရလာတဲ့ result ကို main query ထဲကို ပြန်ပေးတာမလို့ ပိုမြန်ပါတယ်။
- **Correlated Subquery:** Main query ထဲက record တစ်ခုချင်းစီအတွက် subquery ကို ထပ်ခါထပ်ခါ ပြန် run လုပ်ရတာမလို့ data အရမ်းများလာရင် နည်းနည်းတော့ နှေးတတ်ပါတယ်။
- **Suggestion:** ဖြစ်နိုင်ရင်တော့ correlated subquery အစား JOIN နဲ့ ပြောင်းသုံးတာက ပိုမြန်တတ်ပါတယ်။

3. Subqueries in Action

3.1 Filtering with Subqueries

Subquery ကို သုံးပြီး အနည်းဆုံး order တန်ဖိုးထက် ပိုများတဲ့ order တွေကို ရှာကြည့်ကြပါမယ်။ Query ကိုတော့ နားလည်ရလွယ်အောင် ဖော်ပြထားပါတယ်။

```
SELECT product, amount
FROM orders
WHERE amount > (SELECT MIN(amount) FROM orders);
```

```
exercises_db=# SELECT product, amount
exercises_db=# FROM orders
exercises_db=# WHERE amount > (SELECT MIN(amount) FROM orders);
 product | amount
-----+-----
    rice  |  25000
     oil  |  15000
(2 rows)
```

3.2 Subqueries for Calculations

အခုတစ်ခေါက်မှာတော့ subquery ကို အသုံးပြုပြီး customer တစ်ယောက်ချင်းစီအတွက် သူတို့မှာထားတဲ့ order အရေအတွက် ဘယ်လောက်ရှိလဲဆိုတာကို ပြန်တွက်ကြည့်ရအောင်ပါ။

```
SELECT c.customer_name,
       (SELECT COUNT(*) FROM orders o WHERE o.customer_id = c.customer_id)
AS order_count
```

```
FROM customers c;
```

Alias names တွေနဲ့ ရေထားတာတစ်ခုပဲ မတူတာပါ။ ဒီ chapter ရဲ့ အစဆုံးမှာ ရေးခဲ့တဲ့ query ပဲဖြစ်တာကြောင့် ရလဒ်က အခုလိုပဲ တူတူပြန်တွေ့ရမှာပါ။

```
exercises_db=# SELECT c.customer_name,
exercises_db=# (SELECT COUNT(*) FROM orders o WHERE o.customer_id = c.customer_id) AS order_count
exercises_db=# FROM customers c;
 customer_name | order_count
-----+-----
 John Smith    |          2
 Mary Johnson  |          1
 Michael Anderson |          0
(3 rows)
```

4. When to Use Subqueries vs. Joins

4.1 Trade-offs and Best Practices

→ Subquery:

- ရှိရင်းတဲ့ filtering နဲ့ small-scale calculation တွေအတွက် သင့်တော်ပါတယ်။
- Query ရဲ့ structure က ပိုရှင်းရှင်းပြီး ဖတ်ရလွယ်ပါတယ်။
- တခုပြောရရင်တော့ correlated subqueries တွေသုံးရင် performance ပိုင်းမှာ နည်းနည်းနှေးလာနိုင်ပါတယ်။

→ Joins:

- Multiple tables က data တွေကို combine လုပ်ချင်တဲ့အခါမှာ **JOIN** အထူးသင့်လျော်ပါတယ်။
- Execution speed က ပိုမြန်နိုင်ပါတယ်။
- ဒါပေမယ့် complex queries တွေမှာ **JOIN** statement တွေနဲ့ မှန်ကန်အောင် ရေးရခက်နိုင်ပါတယ်။

4.2 Rewriting Subqueries as Joins

Nested subqueries တွေကို မကြာခဏဆိုသလို **JOIN** statement တွေနဲ့ အစားထိုးနိုင်ပါတယ်။ Query performance ကို ပိုကောင်းစေမှာပါ။

```
SELECT c.customer_name
FROM customers c
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.customer_id = c.customer_id AND o.amount > 20000
);
```

JOIN နဲ့ ပြောင်းရေးကြည့်မယ်ဆိုရင် syntax က ပိုတိုပြီး logic ကလည်း ရှင်းရှင်းလင်းလင်းနဲ့ ထိထိမိမိ ဖြစ်နေတာကို အောက်မှာတွေ့ရမှာပါ။

```
SELECT c.customer_name
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.amount > 20000;
```

ရလဒ်ချင်းတော့ တူတူပဲ ဖြစ်ပါတယ်။

```
exercises_db=# SELECT c.customer_name
exercises_db=# FROM customers c
exercises_db=# JOIN orders o ON c.customer_id = o.customer_id
exercises_db=# WHERE o.amount > 20000;
 customer_name
-----
John Smith
(1 row)
```

5. Best Practices

- **Prioritize simplicity** - ဖြစ်နိုင်ရင် correlated subqueries ထက် **JOINS** ကို အသုံးပြုတာ ပိုရိုးရှင်းပြီး နားလည်ရလွယ်ပါတယ်။
- **Check performance** - Data ပမာဏများလာတဲ့အခါ subquery တစ်ခုချင်းစီရဲ့ execution time ကို စစ်ကြည့်ပါ။

- **Use indexes** - Subquery ထဲမှာ အသုံးပြုတဲ့ column တွေမှာ indexing လုပ်တာက query speed ကို ပိုမြန်ဆန်စေနိုင်ပါတယ်။
- **Write clearly** - Queries တွေကို readable ဖြစ်အောင်ရေးရင် တခြား developer တွေ ဖတ်တဲ့အခါ အဆင်ပြေပါလိမ့်မယ်။

6. Exercises

လေ့ကျင့်ခန်းတွေကို ထပ်မံပြီး ကိုယ်တိုင်လုပ်ကြည့်ပါ။ Subquery အကြောင်းကို သေချာနားလည်အောင် လို့ပါ။ Chapter 6 က table နှစ်ခုကိုပဲ ပြန်အသုံးပြုမှာပါ။

// students Table

```
[exercises_db=# SELECT * FROM students;
 student_id | student_name | class
-----+-----+-----
          1 | Ko Ko       | Grade5
          2 | Nyi Nyi     | Grade7
          3 | Su Lay      | Grade5
          4 | Hnin Eain   | Grade7
(4 rows)
```

// scores Table

```
[exercises_db=# SELECT * FROM scores;
 score_id | student_id | subject | score | exam_date
-----+-----+-----+-----+-----
          1 |          1 | Myanmar |    85 | 2025-05-01
          2 |          1 | Math    |    90 | 2025-05-01
          3 |          2 | Myanmar |    78 | 2025-05-02
          4 |          3 | Math    |    92 | 2025-05-01
          5 |          4 | Myanmar |    80 | 2025-05-02
(5 rows)
```

Question 1:

Find scores that are greater than the **average** score using a **subquery** in the **WHERE** clause.

Question 2:

Find the number of exam attempts for each student using a **correlated subquery** in the **SELECT** clause.

Question 3:

Find the student with the highest score in the Myanmar subject using a **subquery** in the **FROM** clause.

Question 4:

Find students who have at least one exam score using a **correlated subquery** with the **EXISTS** clause.

Question 5:

Rewrite the **subquery** from Question 4 as a **JOIN**.

Chapter 8: Window Functions

SQL မှာ window functions တွေကတော့ table ထဲက record တစ်ခုချင်းစီအတွက် တစ်ကြိမ်ချင်း ပြန်လည်တွက်ချက်ပေးတဲ့ function တွေပါပဲ။ အထူးသဖြင့် row တစ်ခုချင်းစီပေါ်မူတည်ပြီး တန်ဖိုးပြန် ထုတ်ပေးတဲ့အတွက် aggregate functions (**SUM**, **AVG**, **COUNT**) တွေနဲ့ မတူပါဘူး။ Aggregate functions တွေက group တစ်ခုလုံးအတွက် တန်ဖိုးတစ်ခုပဲ ထုတ်ပေးတာပါ။ ဒါပေမဲ့ Window functions တွေကျတော့ row တစ်ခုချင်းစီအတွက် ထပ်ဆင့်တွက်ချက်မှုတွေ လုပ်ပေးနိုင်တဲ့အတွက် flexible လဲ ဖြစ်သလို အသုံးဝင်မှုလဲ များပါတယ်။

ဒီအခန်းမှာတော့ window functions ဆိုတာ ဘာလဲဆိုတာကို အရင်ဆုံး နားလည်အောင်ရှင်းပြပါမယ်။ ဘယ်လို syntax structure နဲ့ ရေးလဲ၊ partition လုပ်တာကဘာလဲ၊ frame သတ်မှတ်တာဘယ်လိုလဲ၊ အရေးကြီးတဲ့ ranking functions တွေ (**ROW_NUMBER**, **RANK**, **DENSE_RANK**) ကို ဘယ်လိုသုံးလဲ ဆိုတာမျိုးတွေကို တစ်ခုချင်းစီ နားလည်သွားအောင် ရှင်းပြသွားမှာ ဖြစ်ပါတယ်။

နောက်ပြီးတော့ လက်တွေ့အသုံးချမှုအနေနဲ့ window functions တွေကို ဘယ်လို scenario တွေမှာ အသုံးပြုလို့ရနိုင်လဲဆိုတာကိုပါ နမူနာတွေနဲ့တကွ တစ်ပါတည်းဖော်ပြသွားပါမယ်။ Chapter နောက်ဆုံးမှာ တော့ ထုံးစံအတိုင်း ကိုယ်တိုင်စမ်းရေးကြည့်ဖို့ လေ့ကျင့်ခန်းလေးတွေလည်း ထည့်ပေးထားပါတယ်။

1. Introduction to Window Functions

1.1 What They Are and Why They're Useful

SQL မှာ window functions တွေကတော့ data record တွေကို အုပ်စုဖွဲ့ပြီး row တစ်ခုချင်းစီအလိုက် သီးခြားရလဒ်တွေ ထုတ်ပေးနိုင်တဲ့ function တွေလို့ ရှေ့မှာပြောခဲ့ပါတယ်။ အဓိကအားဖြင့်တော့ row တစ်ခုချင်းစီကိုအခြေခံပြီး တွက်ချက်မှုတွေ လုပ်ပေးတဲ့အတွက် subtotal တွေ၊ ranking သတ်မှတ်မှုတွေ၊ moving average တွေလို တခြားရှုပ်ထွေးတဲ့ လုပ်ဆောင်ချက်တွေမှာ အသုံးဝင်ပါတယ်။ Aggregate functions လိုမျိုး group တစ်ခုလုံးအတွက် တန်ဖိုးတစ်ခုတည်းကို ထုတ်ပေးတာမဟုတ်ဘဲ row တစ်ခုချင်းစီအတွက် ရလဒ်တွေကို သီးသန့်ထုတ်ပေးနိုင်တာက အဓိကအချက်ဖြစ်ပါတယ်။

ဥပမာ - ဝန်ထမ်းတွေရဲ့ လစာကို ဘဏ်ဌာနအလိုက် အစဉ်လိုက်စီပြီး အမြင့်ဆုံးလခအထိ rank ထားချင်တဲ့ အခါမှာ window function တွေထဲက **RANK()** ကို အသုံးပြုနိုင်ပါတယ်။ ဒီနည်းလမ်းကို payroll report

တွေ၊ employee dashboard တွေမှာ ဝန်ထမ်းတစ်ဦးချင်းစီရဲ့ လစာအဆင့်ကို သဲသဲကွဲကွဲဖြစ်အောင်ပြသဖို့ အသုံးပြုနိုင်ပါတယ်။

1.2 Syntax and Structure

Window function တွေသုံးတဲ့အခါမှာ **OVER** keyword ကို မဖြစ်မနေ သုံးရပါတယ်။ ဒီ **OVER** ထဲမှာတော့ အောက်ပါ အစိတ်အပိုင်းတွေကို ထည့်ရေးနိုင်ပါတယ်။

- **PARTITION BY** - ဒါကတော့ ဒေတာတွေကို အုပ်စုဖွဲ့ဖို့ သုံးတာ ဖြစ်ပါတယ်။
- **ORDER BY** - အုပ်စုအတွင်းမှာ row တွေကို အစဉ်လိုက်စီဖို့အတွက်ပါ။ သိပြီးသားပါ။
- **Frame Specification** - ဘယ် row ကနေ ဘယ် row အထိ တွက်မလဲဆိုတာ သတ်မှတ်ဖို့ သုံးတာ ပါ။

// Sample Syntax

```
SELECT column_name,
       FUNCTION() OVER (
         PARTITION BY column
         ORDER BY column
         ROWS BETWEEN ...
       ) AS result_column
FROM table_name;
```

SUM(), **AVG()**, **RANK()**, **ROW_NUMBER()** aggregate functions တွေဖြစ်ပေမယ့် **OVER** clause နဲ့ တွဲသုံးလိုက်တဲ့အခါ window function အနေနဲ့ အလုပ်လုပ်သွားပါတယ်။ ဒီ syntax structure ကတော့ window function တွေသုံးချင်တဲ့အခါ အခြေခံအဖြစ် မဖြစ်မနေ သိထားသင့်တဲ့ပုံစံပဲ ဖြစ်ပါတယ်။

ရှေ့အခန်း(၄) မှာ အသုံးပြုခဲ့တဲ့ **company_db** ထဲက **employees** table ကို ပြန်အသုံးပြုပါမယ်။

employees table ထဲက ဝန်ထမ်းတွေကို department (ဌာန) အလိုက် ခွဲပြီး အဲ့ဒီဌာနအတွင်း salary (လစာ) အရ အမြင့်ဆုံးကနေ အနိမ့်ဆုံးကို စီပါမယ်။ **RANK()** function ကို အသုံးပြုပြီး ဝန်ထမ်းတစ်ဦးချင်းစီအတွက် အဆင့်တွေကို တွက်ပေးပါမယ်။

```
SELECT
  name AS employee_name,
  department,
  salary,
  RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS
  salary_rank
FROM employees;
```

RANK() OVER (PARTITION BY department ORDER BY salary DESC) ဆိုတာကတော့ window function တစ်ခုဖြစ်ပြီး **PARTITION BY department** ဆိုတာက ဝန်ထမ်းတွေကို ဌာနအလိုက် တူရာတူရာ စုလိုက်တာပါ။ **ORDER BY salary DESC** ဆိုတာက သိတဲ့အတိုင်း လစာအမြင့်ဆုံးကနေ အနိမ့်ဆုံးအထိ စီပေးတာပါ။ နောက်ဆုံးမှာတော့ **RANK()** function က ဌာနတစ်ခုချင်းစီအတွင်း ဝန်ထမ်းတစ်ဦးချင်းစီရဲ့ လစာအဆင့် (salary rank) ကို တွက်ပေးတာ ဖြစ်ပါတယ်။ **MGMT** department မှာ **Imran Khan** လစာနည်းတဲ့အတွက် **salary_rank** 2 ရနေတာကို သတိပြုပါ။

```
company_db=# SELECT
company_db=#   name AS employee_name,
company_db=#   department,
company_db=#   salary,
company_db=#   RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS salary_rank
company_db=# FROM employees;
```

employee_name	department	salary	salary_rank
Devi Kumar	ANLY	5000.00	1
Eaint San	DSGN	5200.25	1
Hla Myint	HR	4700.50	1
Alice Smith	IT	5000.00	1
Chan Myae	IT	5000.00	1
Faisal Rahman	IT	5000.00	1
Bob Johnson	MGMT	7200.50	1
Imran Khan	MGMT	6800.00	2
Lin Htet	MKTG	5100.00	1
Khin Zaw	SUPP	3500.00	1

(10 rows)

2. Ranking Functions

Window functions ထဲမှာမှ ranking functions အမျိုးမျိုး ရှိပါသေးတယ်။

2.1 ROW_NUMBER()

ROW_NUMBER() function ဟာ row တစ်ခုချင်းစီအတွက် ထူးခြားတဲ့ unique နံပါတ် (serial number) ကိုပေးတဲ့ function ပါ။ ဒါပေမယ့် အဲ့ဒီ unique number ဟာ **ORDER BY** နဲ့ သတ်မှတ်ထားတဲ့

အစီအစဉ်ပေါ် မူတည်ပါတယ်။ ဆိုလိုတာက အမြင့်ဆုံးလစာရဲ့ row က 1 ဖြစ်ပြီး အောက်ကိုဆင်းလာလေ နံပါတ်ဟာ တိုးလာမှာပါ။

ဥပမာ - **employees** table ထဲမှ ဝန်ထမ်းတွေကို လစာအမြင့်ဆုံးကနေ အနိမ့်ဆုံးအထိ **DESC** နဲ့ စီမှာပါ။ အဲ့ဒါကို ထပ်ပြီးမှ **ROW_NUMBER()** ဖြင့် အဆင့်နံပါတ် ပေးလိုက်တာပါ။

```
SELECT
  name AS employee_name,
  salary,
  ROW_NUMBER() OVER (ORDER BY salary DESC) AS salary_rank
FROM employees;
```

ရလဒ်ဟာ အောက်ပါအတိုင်းပဲ ရမှာဖြစ်ပါတယ်။

```
company_db=# SELECT
company_db=#   name AS employee_name,
company_db=#   salary,
company_db=#   ROW_NUMBER() OVER (ORDER BY salary DESC) AS salary_rank
company_db=# FROM employees;
 employee_name | salary  | salary_rank
-----+-----+-----
 Bob Johnson   | 7200.50 |          1
  Imran Khan   | 6800.00 |          2
   Eaint San   | 5200.25 |          3
   Lin Htet    | 5100.00 |          4
 Faisal Rahman | 5000.00 |          5
   Chan Myae   | 5000.00 |          6
   Devi Kumar  | 5000.00 |          7
   Alice Smith | 5000.00 |          8
   Hla Myint   | 4700.50 |          9
   Khin Zaw    | 3500.00 |         10
(10 rows)
```

2.2 RANK()

RANK() function ဟာ သတ်မှတ်ထားတဲ့ အစီစဉ်အတိုင်း တန်ဖိုးတူတဲ့ row တွေရှိရင် တူညီတဲ့အဆင့် (rank) ဖြင့် သတ်မှတ်ပေးပါတယ်။ အဲ့တော့ တန်ဖိုးတူနေတဲ့ row တွေကြောင့် နောက်ထပ်လာမယ့် row တွေအတွက် rank number ဟာ skip ဖြစ်သွားတတ်ပါတယ်။ တစ်နည်းအားဖြင့် gap ဖြစ်သွားတာပါ။ ဒီလိုပါ တန်ဖိုးတူတဲ့ row နှစ်ခုမှာ rank 1 ဖြစ်ရင် နောက်လာတဲ့ row ဟာ rank 2 အစဉ်လိုက် မလာတော့ဘဲ rank 3 ဖြစ်သွားတာပါ။

```
SELECT
    employee_id,
    name,
    department,
    salary,
    RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS
    salary_rank
FROM employees;
```

IT department အုပ်စုကိုပဲ ကြည့်ပါ။ ဒုတိယမြောက်ကနေ စတုတ္ထအထိ salary တူတာကြောင့် rank number သုံးခုလည်း တူသွားပါတယ်။ ဒါကြောင့် နောက် employee row အတွက် 5 ကနေ ပြန်စပါတယ်။

```
company_db=# SELECT
company_db=#     employee_id,
company_db=#     name,
company_db=#     department,
company_db=#     salary,
company_db=#     RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS salary_rank
company_db=# FROM employees;
```

employee_id	name	department	salary	salary_rank
4	Devi Kumar	ANLY	5000.00	1
13	Thura Min	ANLY	4800.00	2
5	Eaint San	DSGN	5200.25	1
14	Moe Zaw	DSGN	5000.00	2
7	Hla Myint	HR	4700.50	1
15	Aye Thida	HR	4500.00	2
12	Su Su Aung	IT	5300.00	1
1	Alice Smith	IT	5000.00	2
3	Chan Myae	IT	5000.00	2
6	Faisal Rahman	IT	5000.00	2
11	Nyein Chan	IT	4500.00	5
2	Bob Johnson	MGMT	7200.50	1
8	Imran Khan	MGMT	6800.00	2
10	Lin Htet	MKTG	5100.00	1
9	Khin Zaw	SUPP	3500.00	1

(15 rows)

2.3 DENSE_RANK()

DENSE_RANK ကတော့ RANK နဲ့ ဆန့်ကျင်ဘက်ပါ။ တူညီတဲ့ တန်ဖိုးတွေအတွက် တူညီတဲ့ rank အဆင့်ကို သတ်မှတ်ပေးပေမယ့် နောက် row ရဲ့ rank number ဟာ အစဉ်လိုက်ပဲ လာပါမယ်။ gap မရှိတော့ပါ။

```
SELECT
    employee_id,
    name,
    department,
```

```
salary,
DENSE_RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS
salary_dense_rank
FROM employees;
```

ဒီနေရာမှာ ရလဒ်ကို ထပ်ပြီးမပြတော့ပါဘူး။

2.4 Handling Ties and Gaps

- **ROW_NUMBER():** တူညီတဲ့ တန်ဖိုးတွေရှိပေမယ့် row တစ်ခုချင်းစီကို မတူညီတဲ့နံပါတ်များနဲ့ သတ်မှတ်ပေးပါတယ်။ ထို့ကြောင့် တူညီတဲ့ တန်ဖိုးများရှိခဲ့ရင်လည်း နံပါတ်တွေက unique ဖြစ်နေမှာပါ။ ဥပမာ - 1, 2, 3 ။
- **RANK():** တန်ဖိုးတူညီတဲ့ row တွေအတွက် တူညီတဲ့အဆင့် (rank) ကိုပေးပေးမယ့် နောက်ထပ် row တွေမှာတော့ နံပါတ်ကို skip လုပ်လိုက်ပါတယ်။ ဥပမာ - 1, 1, 3 ။
- **DENSE_RANK():** တန်ဖိုးတူရင်လည်း တူညီတဲ့အဆင့်ကိုပဲ ပေးပါတယ်။ ဒါပေမယ့် RANK() လို skip မလုပ်ပါဘူး။ နံပါတ်တွေကို ဆက်တိုက်ပေးပါတယ်။ ဥပမာ - 1, 1, 2 ။

လိုအပ်ချက်ပေါ်မူတည်ပြီး အသုံးပြုရပါမယ်။

3. Partitioning and Framing

3.1 PARTITION BY for Grouping

PARTITION BY က data တွေကို အုပ်စုလိုက်ခွဲဖို့ သုံးတာပါ။ ဒါပေမယ့် aggregate functions လိုမျိုး table တစ်ခုလုံးအတွက် ရလဒ်တစ်ခုတည်း ထွက်လာတာမဟုတ်ပါဘူး။ အဲ့အစား **PARTITION BY** ဟာ data ကို window လို့ခေါ်တဲ့ အုပ်စုလေးတွေအလိုက် စီပြီး function တွေကို တွက်ပေးတာပါ။

```
SELECT
employee_id,
name,
department,
salary,
AVG(salary) OVER (PARTITION BY department) AS avg_department_salary
```

```
FROM employees;
```

ဒီ query မှာ **AVG(salary)** ဆိုတာက employees တွေရဲ့ လစာတွေကို ပျမ်းမျှတွက်ပေးတာပါ။ ဒါပေမယ့် **OVER (PARTITION BY department)** ပါတဲ့အတွက် table တစ်ခုလုံးရဲ့ စုစုပေါင်းပျမ်းမျှထက် department အလိုက်အုပ်စုခွဲပြီး ပျမ်းမျှအသီးသီးကို တွက်ပေးတာပါ။ ဆိုလိုတာက HR department ရဲ့ ပျမ်းမျှလစာတစ်ခု၊ IT department ရဲ့ ပျမ်းမျှလစာတစ်ခုဆိုပြီး သက်သက်စီ ထွက်လာပါမယ်။

ပုံမှန်ဆို **AVG** လို aggregate function က table တစ်ခုလုံးအတွက် ရလဒ်တစ်ခုပဲ ထုတ်ပေးပေမယ့် **PARTITION BY** နဲ့တွဲလိုက်တဲ့အခါ row တစ်ခုချင်းစီအတွက် သူ့အုပ်စု (department) ရဲ့ ပျမ်းမျှကို ပြပေးတာ ဖြစ်ပါတယ်။ ဒါက window function ရဲ့ အလုပ်လုပ်ပုံပါ။

```
company_db=# SELECT
company_db=#   employee_id,
company_db=#   name,
company_db=#   department,
company_db=#   salary,
company_db=#   AVG(salary) OVER (PARTITION BY department) AS avg_department_salary
company_db=# FROM employees;
```

employee_id	name	department	salary	avg_department_salary
13	Thura Min	ANLY	4800.00	4900.0000000000000000
4	Devi Kumar	ANLY	5000.00	4900.0000000000000000
14	Moe Zaw	DSGN	5000.00	5100.1250000000000000
5	Eaint San	DSGN	5200.25	5100.1250000000000000
7	Hla Myint	HR	4700.50	4600.2500000000000000
15	Aye Thida	HR	4500.00	4600.2500000000000000
1	Alice Smith	IT	5000.00	4960.0000000000000000
3	Chan Myae	IT	5000.00	4960.0000000000000000
6	Faisal Rahman	IT	5000.00	4960.0000000000000000
11	Nyein Chan	IT	4500.00	4960.0000000000000000
12	Su Su Aung	IT	5300.00	4960.0000000000000000
2	Bob Johnson	MGMT	7200.50	7000.2500000000000000
8	Imran Khan	MGMT	6800.00	7000.2500000000000000
10	Lin Htet	MKTG	5100.00	5100.0000000000000000
9	Khin Zaw	SUPP	3500.00	3500.0000000000000000

(15 rows)

3.2 Window Frames for Sliding Calculations

Window frame ဆိုတာက window function တွေမှာ ဘယ် row တွေကိုယူပြီး တွက်မလဲဆိုတာကို သတ်မှတ်ပေးတဲ့ အပိုင်းပါ။ ဥပမာ - **ROWS BETWEEN** ကိုသုံးပြီး လက်ရှိ row နဲ့ သူ့အရင် row တချို့ကို တွဲယူပြီး တွက်ချင်တဲ့အခါ အသုံးဝင်ပါတယ်။ ဒီလိုနဲ့ data တစ်ခုချင်းစီအတွက် dynamic calculation တွေ ဖန်တီးလို့ရတယ်။ ဒီလို calculation ကို moving average လို့လည်း ခေါ်ပါတယ်။

```
SELECT
  employee_id,
```

```

name,
salary,
AVG(salary) OVER (
    ORDER BY employee_id
    ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
) AS avg_salary_last_3
FROM employees;

```

ဒီ query မှာ **AVG(salary)** ဟာ **ROWS BETWEEN 2 PRECEDING AND CURRENT ROW** ကို သုံးပြီး လက်ရှိ row နဲ့ အတူ အပေါ်က row 2 ခုအထိ ပေါင်းပြီး average salary ကို တွက်ပေးတာပါ။ ဒါကြောင့် row တစ်ခုချင်းစီအတွက် sliding window တစ်ခုပေါ်လာပြီး အဲ့ဒီ window မှာပါဝင်တဲ့ row 3 ခု (ယခင် 2 ခု + လက်ရှိ 1 ခု) ကိုယူပြီး average salary ကို တွက်ပေးတာ ဖြစ်ပါတယ်။ ထိပ်ဆုံး row အတွက်တော့ သူ့အပေါ်မှာ ဘာမှမရှိတော့တာကြောင့် average salary က ပြောင်းမသွားပါဘူး။

```

company_db=# SELECT
company_db=#     employee_id,
company_db=#     name,
company_db=#     salary,
company_db=#     AVG(salary) OVER (
company_db=#         ORDER BY employee_id
company_db=#         ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
company_db=#     ) AS avg_salary_last_3
company_db=# FROM employees;
 employee_id |      name      | salary | avg_salary_last_3
-----
1 | Alice Smith    | 5000.00 | 5000.0000000000000000
2 | Bob Johnson    | 7200.50 | 6100.2500000000000000
3 | Chan Myae     | 5000.00 | 5733.5000000000000000
4 | Devi Kumar     | 5000.00 | 5733.5000000000000000
5 | Eaint San      | 5200.25 | 5066.7500000000000000
6 | Faisal Rahman  | 5000.00 | 5066.7500000000000000
7 | Hla Myint      | 4700.50 | 4966.9166666666666667
8 | Imran Khan     | 6800.00 | 5500.1666666666666667
9 | Khin Zaw       | 3500.00 | 5000.1666666666666667
10 | Lin Htet       | 5100.00 | 5133.3333333333333333
11 | Nyein Chan     | 4500.00 | 4366.6666666666666667
12 | Su Su Aung     | 5300.00 | 4966.6666666666666667
13 | Thura Min      | 4800.00 | 4866.6666666666666667
14 | Moe Zaw        | 5000.00 | 5033.3333333333333333
15 | Aye Thida      | 4500.00 | 4766.6666666666666667
(15 rows)

```

4. Practical Applications

4.1 Running Totals

Running total ကတော့ လက်ရှိ row တစ်ခုချင်းစီအတွက် ယခင် row တွေအားလုံးနဲ့အတူ စုပေါင်းတန်ဖိုး ကို တွက်ပေးတဲ့ calculation တစ်ခုပဲ ဖြစ်ပါတယ်။


```

SELECT
  name,
  salary,
  SUM(salary) OVER (
    ORDER BY employee_id
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
  ) AS running_total
FROM employees;

```

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW - "UNBOUNDED" ပါတာကြောင့် လက်ရှိ row ကနေပြီး အပေါ်မှာရှိတဲ့ ထိပ်ဆုံး row အထိ salary များကို ပေါင်းပြီး (running total) တစ်ခု ထုတ်ပေးတာပါ။ အောက်ကို row တွေ ဆင်းလာလေလေ ပေါင်းထားတဲ့တန်ဖိုးတွေဟာ ပိုပြီးကြီးလာလေပဲ ဖြစ်ပါတယ်။

```

company_db=# SELECT
company_db=#   name,
company_db=#   salary,
company_db=#   SUM(salary) OVER (
company_db=#     ORDER BY employee_id
company_db=#     ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
company_db=#   ) AS running_total
company_db=# FROM employees;

```

name	salary	running_total
Alice Smith	5000.00	5000.00
Bob Johnson	7200.50	12200.50
Chan Myae	5000.00	17200.50
Devi Kumar	5000.00	22200.50
Eaint San	5200.25	27400.75
Faisal Rahman	5000.00	32400.75
Hla Myint	4700.50	37101.25
Imran Khan	6800.00	43901.25
Khin Zaw	3500.00	47401.25
Lin Htet	5100.00	52501.25
Nyein Chan	4500.00	57001.25
Su Su Aung	5300.00	62301.25
Thura Min	4800.00	67101.25
Moe Zaw	5000.00	72101.25
Aye Thida	4500.00	76601.25

(15 rows)

4.2 Moving Averages

Moving average ဆိုတာက သတ်မှတ်ထားတဲ့ row အရေအတွက်အတွင်းမှာ ပျမ်းမျှတန်ဖိုးကို တွက်တာ ပါ။ လက်ရှိ row နဲ့အတူ ယခင် 2 row တို့ပေါင်းပြီး အဲ့ဒီ 3 row ရဲ့ average ကို ထုတ်ပြတာမျိုးဖြစ် ပါတယ်။


```
SELECT
    employee_id,
    name,
    salary,
    AVG(salary) OVER (
        ORDER BY employee_id
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS moving_avg
FROM employees;
```

3.2 က query နဲ့ ပုံစံတူဖြစ်တာကြောင့် ရလဒ်ကို ထပ်မံပြောဆိုပါ။

4.3 Top-N per Group

Group တစ်ခုစီမှာ ထိပ်ဆုံး N ရောက်တဲ့ record တွေကို ရှာဖွေဖို့ window function တွေကို အသုံးပြုနိုင်ပါတယ်။ Group ဆိုတာ department လည်း ဖြစ်နိုင်ပါတယ်။ ကြိုက်တာဖြစ်နိုင်ပါတယ်။

```
SELECT *
FROM (
    SELECT
        employee_id,
        name,
        department,
        salary,
        RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS
        salary_rank
    FROM employees
) AS ranked_employees
WHERE salary_rank ≤ 2;
```

Subquery ရဲ့ ရလဒ်ကို **ranked_employees** ဆိုပြီး alias အနေနဲ့ ထပ်ပေးထားပါတယ်။ ပြီးမှ department တိုင်းက ထိပ်ဆုံး employee 2 ယောက်ကိုဖော်ပြမှာပါ။ ဒါပေမယ့် rank တူရင် တူတဲ့ အလျှောက် **salary_rank 1** ရတဲ့သူက နှစ်ယောက်လည်း ဖြစ်နိုင်ပါတယ်။ သုံးယောက်လည်း ဖြစ်နိုင်ပါတယ်။ ပိုပြီးလည်း ဖြစ်နိုင်ပါတယ်။ ကိုယ်တိုင်ရေးကြည့်ရင်း အပြောင်းအလဲတွေကို သတိပြုပါ။

5. Exercises

အောက်မှာပေးထားတဲ့ link ကိုနှိပ်ပြီး ပေးထားတဲ့ table တွေကို အသစ်တည်ဆောက်ပြီး မေးခွန်းတစ်ခုချင်းစီကို သေချာလေ့ကျင့်ကြည့်ပါ။

// Source Code Link:

[https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-8/10-exercises-data-records.s
ql](https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-8/10-exercises-data-records.sql)

Question 1: Order Ranking per Customer

For each customer, rank their orders by **order amount** (highest to lowest) using **ROW_NUMBER()**. This helps identify their biggest purchases uniquely, even if amounts tie.

Question 2 : Salesperson Performance Ranking with Ties

Rank **salespersons** based on **monthly sales** using **RANK()**. If two salespersons have the same **sales amount**, they share the same **rank**, and gaps appear in the ranking. Useful for leaderboard displays.

Question 3: Total Revenue per Region

Calculate **total revenue** generated per **sales region** using **SUM()** with **PARTITION BY**. This helps generate regional sales reports showing how much each **region** contributes.

Question 4: Moving Average of Daily Website Visitors

For each **day**, calculate the **moving average** of **website visitors** over the last 3 days using **AVG()** with a sliding window frame (**ROWS BETWEEN 2 PRECEDING AND CURRENT ROW**). This can help track traffic trends smoothly.

Question 5: Top 3 Best-Selling Products per Category

Find the top **3 products** by **sales volume** in each **product category** using **ROW_NUMBER()**. Useful for creating dashboards that highlight popular products by **category**.

Chapter 9: Common Table Expressions - CTEs

SQL မှာ CTE (Common Table Expressions) ကို ရှုပ်ထွေးတဲ့ query တွေကို ရှိရင်းအောင်ရေးနိုင်ဖို့ အသုံးပြုပါတယ်။ CTE ကို WITH နဲ့ စရေးပြီး temporary result table တစ်ခု ဖန်တီးပါတယ်။ ပြီးရင်တော့ main query ထဲမှာပဲ အဲဒီ CTE ကို တခြား table တစ်ခုလိုမျိုး ထပ်တလဲလဲ အစားထိုးသုံးနိုင်ပါတယ်။ ပိုပြီးဖတ်ရလွယ်ကူသလို နားလည်လွယ်တာကြောင့် အသုံးများပါတယ်။

ဒီအခန်းမှာတော့ CTE ရဲ့ အဓိပ္ပါယ်၊ အားသာချက်တွေ၊ ရေးပုံရေးနည်းနဲ့ Recursive CTE အပါအဝင် တခြားနမူနာတွေကိုပါ အသေးစိတ် လေ့လာသွားပါမယ်။

1. What are CTEs?

1.1 Definition and Benefits

ထပ်ဖော်ပြပါမယ်။ CTE ဆိုတာ SQL မှာ WITH ဆိုတဲ့ keyword ကိုအသုံးပြုပြီး ယာယီ result set တစ်ခုကို အမည်ပေးရေးသားခြင်းဖြစ်ပါတယ်။ အဲဒီယာယီဇယားကိုပဲ main query အတွင်းမှာ လွယ်လွယ်ကူကူ ပြန်လည်ခေါ်သုံးနိုင်ပါတယ်။ Subquery တွေနဲ့ နှိုင်းယှဉ်ရင်တော့ CTE တွေဟာ logic ကို ခွဲခြားဖော်ပြနိုင်တာကြောင့် query ကို ဖတ်ရလွယ်ကူသလို လိုအပ်ရင် ပြန်ပြင်ဖို့လည်း ပိုအဆင်ပြေစေ ပါတယ်။

သူ့အားသာချက်တွေကတော့-

- **Readability:** ရှုပ်ထွေးတဲ့ query logic ကို ခွဲထုတ်ရေးနိုင်လို့ နားလည်ရလွယ်ကူပါတယ်။
- **Repeated Usability:** တစ်ကြိမ်ပဲဖန်တီးထားတဲ့ CTE ကို တူညီတဲ့ query အတွင်း နေရာအမျိုးမျိုးမှာ ပြန်သုံးနိုင်ပါတယ်။
- **Recursive CTEs:** Hierarchical data (ဥပမာ – manager/employee structure) ကို ကိုင်တွယ်တဲ့နေရာမျိုးမှာ အထူးအသုံးဝင်ပါတယ်။

1.2 Improving Query Readability

ရှေ့မှာဖော်ပြခဲ့သလို CTEs ဟာ ရှုပ်ထွေးတဲ့ query တွေကို အဆင့်လိုက်ခွဲပြီး ဖော်ပြပေးတာကြောင့် ပိုမိုနားလည်ရလွယ်အောင် ကူညီပေးပါတယ်။

ဥပမာ - ရှေ့အခန်းမှာ သုံးခဲ့တဲ့ **customers** နဲ့ **orders** table နှစ်ခုကိုသုံးပြီး customer တစ်ဦးစီအတွက် order စုစုပေါင်းတန်ဖိုးကို တွက်ပြီး 20000 ထက်များတဲ့ သူတွေကိုရှာကြရအောင်ပါ။

// With subquery

```
SELECT customer_name, total_amount
FROM (
    SELECT c.customer_name, SUM(o.amount) AS total_amount
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    GROUP BY c.customer_name
) AS sub
WHERE total_amount > 20000;
```

// With CTE

```
WITH customer_totals AS (
    SELECT c.customer_name, SUM(o.amount) AS total_amount
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    GROUP BY c.customer_name
)
SELECT customer_name, total_amount
FROM customer_totals
WHERE total_amount > 20000;
```

ရလဒ်က တူတူပဲရမှာ ဖြစ်ပါတယ်။ ဒါပေမယ့် CTE version မှာ inner subquery ကို သီးခြားခွဲထုတ်လိုက် တာကြောင့် ပိုရှင်းပြီး ဖတ်ရလွယ်ကူပါတယ်။

```
exercises_db=# WITH customer_totals AS (
exercises_db=#     SELECT c.customer_name, SUM(o.amount) AS total_amount
exercises_db=#     FROM customers c
exercises_db=#     JOIN orders o ON c.customer_id = o.customer_id
exercises_db=#     GROUP BY c.customer_name
exercises_db=# )
exercises_db=# SELECT customer_name, total_amount
exercises_db=# FROM customer_totals
exercises_db=# WHERE total_amount > 20000;
 customer_name | total_amount
-----+-----
 John Smith    |         40000
(1 row)
```

2. Writing CTEs

2.1 Basic CTE Syntax

CTE syntax structure ကို ပြန်လေ့လာကြည့်ကြရအောင်ပါ။ **WITH** keyword နဲ့စပြီး ရလာတာက temporary result set တစ်ခုပါ။

// With CTE

```
WITH cte_name AS (
    SELECT column1, column2
    FROM table_name
    WHERE condition
)
SELECT column1, column2
FROM cte_name
WHERE condition;
```

cte_name ဟာ CTE ရဲ့ နာမည် (alias) ဖြစ်ပြီး မိမိကြိုက်သလို နာမည်ပေးလို့ရပါတယ်။ **SELECT column1, column2 FROM cte_name WHERE condition;** ဒီအပိုင်းကတော့ အပေါ်မှာဖန်တီးထားတဲ့ CTE (**cte_name**) ကို အသုံးပြုပြီး final result ကို ရယူတဲ့အပိုင်းဖြစ်ပါတယ်။ CTE ထဲက ရလာတဲ့ **column1** နဲ့ **column2** တို့ကို ထပ်မံ **WHERE condition** နဲ့ filter လုပ်နိုင်ပါတယ်။

ဥပမာ - CTE ကိုသုံးပြီး ရှေ့အခန်းတွေမှာလို ပျမ်းမျှတန်ဖိုးထက်ကြီးတဲ့ customer ရဲ့ order တွေကို ရှာကြမယ်ဆိုရင်။ ဒီတစ်ခါတော့ table name ရဲ့ first letter ကို capital letter နဲ့ ရေးလိုက်ပါတယ်။

```
WITH avg_amount AS (
    SELECT AVG(amount) AS avg_value
    FROM Orders
)
SELECT product, amount
FROM Orders, avg_amount
WHERE amount > avg_amount.avg_value;
```

CTE ၏ရလဒ် **avg_amount** ကို main query တွင် တိုက်ရိုက် reference လုပ်ထားတာမလို့ **CROSS JOIN** လို ဖြစ်နေပါတယ်။ ဒီလိုပါ **FROM Orders, avg_amount** ကတော့ **Orders CROSS JOIN avg_amount** သဘောတရားနဲ့ ဆင်တူပါတယ်။ ဒါပေမယ့် ရလာမယ့် **avg_amount** မှာ row တစ်ခုပဲရှိလို့ အဲ့လိုပုံစံ ရေးလိုက်တာပါ။ Logic အရ ပိုရှင်းပါတယ်။

```

exercises_db=# WITH avg_amount AS (
exercises_db=#     SELECT AVG(amount) AS avg_value
exercises_db=#     FROM Orders
exercises_db=# )
exercises_db=# SELECT product, amount
exercises_db=# FROM Orders, avg_amount
exercises_db=# WHERE amount > avg_amount.avg_value;
 product | amount
-----+-----
    rice  | 25000
(1 row)

```

2.2 Chaining Multiple CTEs

CTE တစ်ခုရဲ့ ရလဒ်ကို နောက်ထပ် CTE တစ်ခုမှာ reference လုပ်ပြီး ရှုပ်ထွေးတဲ့ query တွေကို အဆင့်ဆင့် ခွဲထုတ်ရေးသားနိုင်ပါသေးတယ်။

```

WITH order_counts AS (
    SELECT
        c.customer_id,
        c.customer_name,
        COUNT(*) AS order_count
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    GROUP BY c.customer_id, c.customer_name
),
order_totals AS (
    SELECT
        c.customer_id,
        c.customer_name,
        SUM(o.amount) AS total_amount
    FROM customers c
    JOIN orders o ON c.customer_id = o.customer_id
    GROUP BY c.customer_id, c.customer_name
)
SELECT
    oc.customer_name,
    oc.order_count,
    ot.total_amount
FROM order_counts oc

```



```
JOIN order_totals ot ON oc.customer_id = ot.customer_id
WHERE oc.order_count > 1;
```

`order_counts` နှင့် `order_totals` ဆိုတဲ့ CTE နှစ်ခုကို သီးခြားဖန်တီးပြီးမှ JOIN နဲ့ ပြန်ပေါင်းတာပါ။

3. Recursive CTEs

3.1 Handling Hierarchical Data (e.g., Org Charts)

Recursive CTEs ကို organization table သို့မဟုတ် employee-manager relationships တို့လို အဆင့်လိုက် ဆက်စပ်နေတတ်တဲ့ data တွေကို manage လုပ်ဖို့ အသုံးပြုနိုင်ပါတယ်။

```
[postgres=# SELECT * FROM organization;
 employee_id | employee_name | manager_id
-----+-----+-----
          1 | Alice         |
          2 | Bob           |          1
          3 | Charlie       |          2
          4 | Diana         |          3
(4 rows)
```

3.2 Recursive Query Examples

ဒီ organization table ကို recursive CTE နဲ့ အသုံးပြုပြီး ပင်မ manager တစ်ဦးအောက်မှာရှိတဲ့ လုပ်သားအားလုံးကို အဆင့်အလိုက် ခွဲထုတ်ဖော်ပြနိုင်ပါတယ်။

```
-- Recursive CTE to build the organization chart with hierarchy levels
WITH RECURSIVE org_chart AS (
  -- Base case: Select top-level employees (those who have no manager)
  SELECT
    employee_id,
    employee_name,
    manager_id,
    1 AS level -- Top level starts at 1
  FROM organization
```

```

WHERE manager_id IS NULL
UNION ALL

-- Recursive step: Get employees managed by the previous level
SELECT
    o.employee_id,
    o.employee_name,
    o.manager_id,
    oc.level + 1 -- Increment level for each subordinate
FROM organization o
JOIN org_chart oc ON o.manager_id = oc.employee_id
)

-- Final result: Show employee names and their level in the hierarchy
SELECT
    employee_name,
    level
FROM org_chart;

```

ဒီတစ်ခါ comment တွေနဲ့ပါ ထည့်ရှင်းပြထားပါတယ်။ **Recursive CTE** ကို အသုံးပြုပြီး organization chart ကို အဆင့်လိုက်ဖော်ပြတာ ဖြစ်ပါတယ်။ အရင်ဆုံး **manager_id** မရှိတဲ့ဝန်ထမ်း (ဥပမာ - CEO တို့လို) ကို **base case** အဖြစ်ရွေးပြီး **level = 1** အဖြစ် သတ်မှတ်ပါတယ်။ ထို့နောက် **UNION ALL** ကို သုံးပြီး recursive step မှာတော့ **manager_id** က base case ထဲက **employee_id** နဲ့ တူတဲ့ ဝန်ထမ်းတွေကို ရွေးထုတ်ပါတယ်။ ဒီလိုနဲ့ level တစ်ခုစီတိုးပြီး (e.g., 1 → 2 → 3...) hierarchy ကို ဆက်လက်ဖော်ပြနိုင်ပါတယ်။

ဥပမာအားဖြင့် - **Alice** ရဲ့ **employee_id = 1** ဖြစ်ပြီး တခြားဝန်ထမ်းတစ်ဦးရဲ့ **manager_id = 1** ဆိုလျှင် အဲ့ဒီဝန်ထမ်းဟာ **Alice** ရဲ့ အောက်အဆင့်ဖြစ်တဲ့ level 2 ဝန်ထမ်းအဖြစ် သတ်မှတ်ပေးပါတယ်။ ဒီလိုနဲ့ recursion တစ်ဆင့်ချင်း ဆင်းသွားပါတယ်။ နောက်ဆုံးအဆင့်မှာ **Diana** ရဲ့ **employee_id = 4** ကို **manager_id** အဖြစ် အသုံးပြုထားတဲ့ ဝန်ထမ်းမရှိတော့တဲ့အချိန်မှာတော့ recursion ပြီးဆုံးသွားပါပြီ။

employee_name	level
Alice	1
Bob	2
Charlie	3
Diana	4

(4 rows)

4. CTEs vs. Subqueries

CTEs:

- ဖတ်ရှုရလွယ်ကူပြီး ရိုးရှင်းပါတယ်။
- တူညီသော ရလဒ်ဇယားကို အကြိမ်များစွာ ပြန်အသုံးပြုနိုင်ပါတယ်။
- Recursive Queries တွေအတွက် ပိုအသုံးဝင်ပါတယ်။

Subqueries:

- ရိုးရှင်းတဲ့ query တွေအတွက်ဆို သင့်တော်ပါတယ်။
- ရှုပ်ထွေးလာရင် ဖတ်ရှုရခက်ခဲနိုင်ပါတယ်။
- Recursive Queries အတွက် အဆင်မပြေပါ။

4.1 When to Choose CTEs

- Queries ကို ပိုမိုရှင်းလင်းပြီး စနစ်တကျရေးချင်တဲ့အခါ။
- တူညီတဲ့ result set ကို query တစ်ခုအတွင်း အကြိမ်များစွာအသုံးပြုရမည့်အခါ။
- Hierarchical သို့ recursive data များကို ကိုင်တွယ်ရမည့်အခါ။

4.2 Performance Considerations

- **CTEs** က temporary result set တွေကို ဖန်တီးတဲ့အတွက် data ပမာဏကြီးတဲ့အခါ performance မှာ နှေးကွေးမှု ဖြစ်တတ်ပါတယ်။
- ရိုးရှင်းတဲ့ queries တွေအတွက်တော့ simple subqueries တွေဟာ ပိုမြန်နိုင်ပါတယ်။
- Data ပမာဏကြီးတဲ့အခါ **CTEs** နဲ့ **subqueries** အပြန်အလှန် performance ဘယ်ဟာပိုကောင်းလဲ ဆိုတာကို စမ်းသပ်ကြည့်ဖို့ လိုပါတယ်။ Data amount နဲ့ query ပုံစံအပေါ်မူတည်ပြီး တစ်ခါတစ်ရံ **CTEs** ပိုမြန်နိုင်ပြီး တစ်ခါတစ်ရံ **subqueries** ပိုထိရောက်နိုင်ပါတယ်။ ဒါကြောင့် **Indexes** တွေကိုသုံးပြီး query performance ပိုကောင်းအောင် ပြုလုပ်နိုင်ပါတယ်။

5. Exercises

အောက်ပါလေ့ကျင့်ခန်းတွေကို ကိုယ်တိုင်လုပ်ကြည့်ဖို့ ရှေ့အခန်းတွေမှာ အသုံးပြုခဲ့တဲ့ **students**, **scores** table တွေကိုပဲ ပြန်အသုံးပြုလို့ရပါတယ်။ **teachers** table ကိုတော့ ထပ်ဖြည့်ရပါမယ်။

// Source Code Link:

<https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-9/06-create-insert-teachers.sql>

```
[exercises_db=# SELECT * FROM teachers;
 teacher_id | position | supervisor_id
-----+-----+-----
          1 | Headmaster |
          2 | Senior Teacher |          1
          3 | Junior Teacher |          2
          4 | Assistant |          3
(4 rows)
```

Question 1:

Use a **CTE** to find exam scores that are higher than the average score.

Question 2:

Using **two CTEs**, calculate the number of exams taken and the total score for each student, then find students who have taken more than one exam.

Question 3:

Use a **recursive CTE** to find the hierarchical relationships among teachers from the **teachers** table.

Question 4:

Rewrite the **CTE query** from **Question 1** as a **subquery**.

Question 5:

Use **CTEs** to find the student with the highest score in each subject.

Chapter 10: Database Design Basics

Database design ဆိုတာကတော့ data တွေကို ထိရောက်အောင် သိမ်းဆည်းနိုင်ဖို့၊ စီမံခန့်ခွဲနိုင်ဖို့နဲ့ လိုအပ်တဲ့အခါ ထုတ်ယူနိုင်ဖို့အတွက် table အချင်းချင်းဆက်နွှယ်မှုတွေကို စနစ်တကျ ဖန်တီးခြင်း ဖြစ်ပါတယ်။ ဒီအခန်းမှာတော့ database design နဲ့ဆိုင်တဲ့ Fundamental Principles၊ Normalization၊ Denormalization၊ Keys and Constraints၊ Indexes နှင့် Real-world examples of table structures တွေကို ထပ်လေ့လာသွားပါမယ်။

1. Principles of Database Design

Database design ကောင်းမွန်ဖို့ဆိုရင် အောက်ပါအချက်တွေကို ထည့်သွင်းစဉ်းစားဖို့ လိုပါတယ်။

- **Data Integrity:** Data ဟာ တိကျမှန်ကန်ဖို့ လိုပါတယ်။
- **Minimize Redundancy:** မလိုအပ်ဘဲ data duplication ဖြစ်တာတွေ ရှောင်ကြဉ်ဖို့ လိုပါတယ်။
- **Performance:** Query တွေ run လုပ်တဲ့အခါ မြန်ဆန်ပြီး efficiency ရှိဖို့ လိုပါတယ်။
- **Maintainability:** Data structure ဟာ လိုအပ်ရင် ပြောင်းလွယ်ပြင်လွယ် ဖြစ်နေရပါမယ်။

1.2 Normalization (1NF, 2NF, 3NF)

Normalization ဆိုတာကတော့ table design ကို စနစ်တကျ ပြင်ဆင်ပြီး data ထပ်နေတဲ့အပိုင်းတွေကို လျှော့ချဖို့ ဖြစ်ပါတယ်။ Data တစ်ခုပြောင်းချင်ရင် တစ်နေရာမှာပဲ ပြင်လိုက်တာနဲ့ အားလုံးကို အဆင်ပြေ ပြေ ပြင်နိုင်ပါတယ်။ ဒါကြောင့် data တွေဟာ တိကျမှုရှိပြီး storage ကိုလည်း ထိထိရောက်ရောက် အသုံးပြု နိုင်ပါတယ်။

First Normal Form (1NF)

- Table မှာ record တစ်ခုစီဟာ unique ဖြစ်ရပါမယ်။
- Column တစ်ခုမှာ တန်ဖိုးတစ်ခုပဲ ပါရပါမယ်။

Denormalized Table

customer_id	customer_name	products_ordered
1	John Smith	oil, rice
2	Mary Johnson	sugar

products_ordered column ထဲမှာ value တစ်ခုတည်းမဟုတ်ဘဲ comma-separated values တွေ ပါလာတယ်။ အဲ့တာကြောင့် denormalized ဖြစ်တယ်လို့ ပြောရမှာပါ။ အဲ့တာကို normalized ဖြစ်တဲ့ စံနှုန်းနဲ့ ရေးချင်တယ်ဆိုရင် အောက်က table တွေအတိုင်းဖြစ်အောင် ရေးရပါမယ်။

Convert to 1NF (Split into multiple rows)

// Orders Table

order_id	customer_id	product	amount
1	1	rice	25000
2	1	oil	15000
3	2	sugar	12000

// Customers Table

customer_id	customer_name
2	Mary Johnson
1	John Smith

products_ordered column ကိုဖယ်ပြီး သူ့အတွင်းထဲမှာပါတဲ့ product တွေကို တစ်ခုချင်းစီ row အသစ်အဖြစ် ခွဲထုတ်ရေးတာပါ။

Second Normal Form (2NF)

- ပထမဦးဆုံး 1NF ဖြစ်ပြီးသား ဖြစ်ရပါမယ်။
- Non-key columns တွေဟာ **primary key** ပေါ်မှာ လုံးဝမှီခိုရပါမယ်။

Denormalized Table

order_id	customer_id	customer_name	product	amount
1	1	John Smith	rice	25000
2	1	John Smith	oil	15000

(2 rows)

customer_name ဟာ **customer_id** နဲ့ ဆက်နွှယ်တာဖြစ်တဲ့အတွက် **orders** table မှာ ထပ်ပြီး ထည့်မနော့ဘဲ customer info ကို သီးခြား **customers** table ထဲမှာ ထားသင့်ပါတယ်။ ဒီလိုခွဲခြားခြင်းက data redundancy ကိုလျော့ချပြီး maintain လုပ်ရတာလွယ်စေပါတယ်။ ဥပမာအနေနဲ့ဖော်ပြထားတာပဲ ဖြစ်ပါတယ်။

Convert to 2NF

// Orders Table

order_id	customer_id	product	amount
1	1	rice	25000
2	1	oil	15000

(2 rows)

// Customers Table

customer_id	customer_name
1	John Smith

(1 row)

customer_name က non-key column တစ်ခုဖြစ်တဲ့အတွက် သူ့အနေနဲ့ **primary key** ဖြစ်တဲ့ **customer_id** ပေါ်မှာပဲ မှီခိုနေရပါတယ်။ ဆိုလိုတာက customer name တစ်ခုချင်းစီဟာ customer ID တစ်ခုတည်းနဲ့ ဆက်နွှယ်နေတာမို့ အဲ့ဒီ **customer_name** ကို **orders** table မှာ မထားပဲ **customers** table ထဲကို သီးသန့်ထုတ်ပေးလိုက်တာပါ။

Third Normal Form (3NF)

- 2NF ဖြစ်ရပါမယ်။
- Non-key columns များဟာဖြင့် အခြားသော non-key columns များပေါ်တွင် မမှီခိုရပါ။ ဥပမာ - column A ဟာ column B ရဲ့ အခြေအနေအရ ဖြစ်ပေါ်လာတာဆိုရင် column B ကို သီးခြား table တစ်ခုထဲ ခွဲထုတ်ပေးသင့်ပါတယ်။

Denormalized Table

customer_id	customer_name	email	email_domain
1	John Smith	john@example.com	example.com
2	Marry Johnson	marry@example.com	example.com

email_domain ဟာ **email** column ထဲမှာပါတဲ့တန်ဖိုးကို အခြေခံပြီး ပြန်ထုတ်ထားတာ ဖြစ်ပါတယ်။ ဒါကြောင့် **email** ကို base column လို့ခေါ်နိုင်ပြီး **email_domain** ကတော့ အခြား non-key column ဖြစ်တဲ့ **email** ကို မှီခိုနေတဲ့ dependent column တစ်ခု ဖြစ်ပါတယ်။

Conversion to 3NF

// Customers Table

customer_id	customer_name	email_id
1	John Smith	1
2	Mary Johnson	2

// Emails Table

email_id	email	email_domain
1	john@example.com	example.com
2	mary@example.com	example.com

email_domain ဟာ **email** column ကို မှီခိုနေတဲ့အတွက် ဒီလိုအချင်းချင်း မှီခိုနေတဲ့ non-key columns တွေကို ခွဲထုတ်ဖို့လိုပါတယ်။ အဲဒီအတွက် **emails** ဆိုတဲ့ table အသစ်တစ်ခုထဲကို **email** နဲ့ **email_domain** ကို သီးခြားထုတ်ထားတာပါ။

1.2 Denormalization for Performance

Denormalization ဆိုတာက performance ပိုကောင်းဖို့အတွက် data တွေ duplicated ဖြစ်နေတာကို လက်ခံပေးပြီး tables တွေကို ပေါင်းစပ်ပေးတာပဲ ဖြစ်ပါတယ်။ ဥပမာအနေနဲ့ **orders** table နဲ့ **customers** table တို့ကို သီးသန့်ခွဲမနေတော့ဘဲ **customer_name**, **customer_email** စတဲ့ customer info တွေကို **orders** table ထဲမှာ တပါတည်းဖြစ်အောင် ထည့်သွင်းလိုက်တာမျိုးပါ။ အဲ့လိုလုပ်ခြင်းက data redundancy ဖြစ်စေရင်လည်း performance ပိုကောင်းဖို့အတွက်ပဲ တချို့ application use-cases မှာ အသုံးဝင်ပါတယ်။

// Denormalized Table

order_id	customer_id	customer_name	product	amount
1	1	John Smith	rice	25000
2	1	John Smith	oil	15000

(2 rows)

Denormalization ရဲ့ အားသာချက်တွေကတော့ data query လုပ်တဲ့အခါ ပိုမြန်ဆန်ပြီး ရှုပ်ထွေးတဲ့ **JOIN** တွေကို လျော့ချနိုင်ခြင်းပါ။ ဒါပေမယ့် အားနည်းချက်ကတော့ data redundancy ကြောင့် သိမ်းဆည်းရမယ့် storage ပိုလိုအပ်လာပြီး data ပြောင်းလဲမှုတွေလုပ်ရတာ ခက်ခဲတတ်ပါတယ်။ ဒါကြောင့် denormalization နည်းကို အသုံးပြုမယ်ဆိုရင် သင့်တော်မှုရှိမရှိ ကြိုတင်စဉ်းစားသင့်ပါတယ်။

2. Keys and Constraints

2.1 Primary Keys and Foreign Keys

Primary key နဲ့ Foreign Key ထပ်ပြီးရှင်းပြချင်ပါတယ်။

Primary Key ဆိုတာက table တစ်ခုထဲမှာရှိတဲ့ record တစ်ခုချင်းစီကို သေချာခွဲခြားနိုင်အောင် သတ်မှတ်ပေးတဲ့ key တစ်ခုဖြစ်ပါတယ်။ ဥပမာ - **customer_id** ဟာ **customers** table ရဲ့ **primary key** လို့ ပြောရမှာပါ။

Foreign Key ကတော့ table တစ်ခုနဲ့တစ်ခုကို ချိတ်ဆက်ပေးတဲ့ key ဖြစ်ပြီး တခြား table ရဲ့ **primary key** ကို ရည်ညွှန်းပါတယ်။ **orders** table ထဲက **customer_id** ဟာ **customers** table ထဲက **customer_id** ကို reference လုပ်တဲ့ **foreign key** လို့ ပြောရမှာပါ။

```
CREATE TABLE customers (  
    customer_id INT PRIMARY KEY,  
    customer_name VARCHAR(50),  
    email VARCHAR(100)  
);
```

```
CREATE TABLE orders (  
    order_id INT PRIMARY KEY,  
    customer_id INT,  
    product VARCHAR(50),  
    amount INT,  
    order_date DATE,  
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)  
);
```

2.2 Unique Constraints and Check Constraints

Unique Constraint ဆိုတာက column တစ်ခုထဲမှာ တန်ဖိုးတူတာမျိုးတွေ မဖြစ်ရအောင် သတ်မှတ်ပေးရတဲ့ စည်းမျဉ်းတစ်ခုပါ။ ဥပမာ - **email** column ထဲမှာ တူညီတဲ့ email တန်ဖိုးတွေ မရှိသင့်ပါဘူး။

Check Constraint ကတော့ column ထဲ ထည့်မယ့်တန်ဖိုးတွေဟာ သတ်မှတ်ထားတဲ့ condition နဲ့ ကိုက်ညီဖို့လိုတဲ့ စည်းမျဉ်းတစ်ခုပါ။ ဥပမာ - **amount** ဟာ 0 ထက်များတဲ့ တန်ဖိုးဖြစ်ရပါမယ်။

email ကို **UNIQUE** လို့ ပေးထားပါတယ်။ **age** ကို **18** နှစ်ထက်ကြီးရမယ်လို့ **CHECK** ထားပါတယ်။

```
CREATE TABLE customers (  
    customer_id INT PRIMARY KEY,  
    customer_name VARCHAR(50),  
    email VARCHAR(100) UNIQUE,  
    age INT CHECK (age ≥ 18)  
);
```

3. Indexes

3.1 How Indexes Work

Index တွေကို ဘာအတွက် အသုံးပြုသလဲဆိုတာနဲ့ ပတ်သက်ပြီး ထပ်ရှင်းပြပါမယ်။ Index တွေဟာ database ထဲက data တွေကို မြန်မြန်ဆန်ဆန် ရှာဖွေထုတ်ယူနိုင်အောင် အထောက်အကူပြုတဲ့ စနစ်တစ်ခုပါ။ ဒီအရာကို စာအုပ်တွေမှာပါတဲ့ မာတိကာနမူနာနဲ့ ချိန်ထိုးပြီး စဉ်းစားနိုင်ပါတယ်။ စာအုပ်တစ်အုပ်ထဲက မာတိကာလိုပဲ **INDEX** အညွှန်းတစ်ခုရှိနေရင် ဘယ် data ကိုမဆို တိတိကျကျနဲ့ လွယ်လွယ်ကူကူ ရှာဖွေနိုင်အောင် လုပ်ပေးပါတယ်။ ဒါကြောင့် large dataset တွေမှာ database performance ကောင်းမွန်ဖို့အတွက် Indexing ပြုလုပ်ထားဖို့ လိုအပ်ပါတယ်။

ဥပမာအားဖြင့် customer_id column ပေါ်မှာ index တစ်ခု ဖန်တီးချင်တယ်ဆိုပါစို့။

```
CREATE INDEX idx_customer_id ON orders(customer_id);
```

ဒီလို index တစ်ခုထည့်လိုက်ရင် **orders** table ထဲက **customer_id** ကိုအခြေခံပြီး ရှာဖွေတဲ့ query တွေဟာ ပိုမိုမြန်ဆန်လာပါမယ်။ အဓိကရည်ရွယ်ချက်ကတော့ searching speed ကို တိုးမြှင့်ပေးဖို့ပဲ ဖြစ်ပါတယ်။

3.2 When to Create Indexes

- မကြာခဏအသုံးပြုရတဲ့ column တွေ - ဥပမာ **WHERE**၊ **JOIN**၊ **ORDER BY** စတဲ့နေရာတွေမှာ ထပ်ခါထပ်ခါသုံးနေရတဲ့ column တွေအတွက် indexing လုပ်ခြင်းက query performance ကို ပိုမိုကောင်းမွန်လာစေပါတယ်။
- ပြီးတော့ data အရေအတွက်များတဲ့ table တွေမှာလည်း index တွေက ရှာဖွေတဲ့ process ကို ပိုမြန်လာအောင် ကူညီပေးနိုင်ပါတယ်။
- အကြံပြုချက်အနေနဲ့တော့ index တွေဟာ storage ပိုလိုအပ်ပြီး data ကို insert/update/delete လုပ်တဲ့အခါ index ကိုပါ ပြန် update လုပ်ရလို့ performance ကို နှေးကွေးစေနိုင်ပါတယ်။ ဒါကြောင့် လိုအပ်တဲ့နေရာမှာပဲ ရွေးချယ်အသုံးပြုသင့်ပါတယ်။

4. Real-World Schema Examples

4.1 E-Commerce Database Design

ဒီ database design က online store တစ်ခုမှာ customer တွေမှာယူတဲ့ order တွေနဲ့ products တွေကို စနစ်တကျသိမ်းဖို့ ပြုလုပ်ထားတာပါ။

```
CREATE TABLE customers (  
    customer_id INT PRIMARY KEY,  
    customer_name VARCHAR(50),  
    email VARCHAR(100) UNIQUE,  
    phone VARCHAR(15)  
);  
  
CREATE TABLE products (  
    product_id INT PRIMARY KEY,  
    product_name VARCHAR(50),  
    price INT CHECK (price > 0)  
);  
  
CREATE TABLE orders (  
    order_id INT PRIMARY KEY,  
    customer_id INT,  
    order_date DATE,  
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)  
);  
  
CREATE TABLE order_items (  
    order_item_id INT PRIMARY KEY,  
    order_id INT,  
    product_id INT,  
    quantity INT CHECK (quantity > 0),  
    amount INT,  
    FOREIGN KEY (order_id) REFERENCES orders(order_id),  
    FOREIGN KEY (product_id) REFERENCES products(product_id)  
);
```

customers နဲ့ orders table တွေကို foreign key နဲ့ ချိတ်ဆက်ထားပြီး order_items မှာတော့ orders နဲ့ products တို့ကို ထပ်မံချိတ်ဆက်ထားပါတယ်။ ဒီလိုချိတ်ဆက်ခြင်းက order တစ်ခုစီအတွက် သက်ဆိုင်တဲ့ product_id၊ quantity နဲ့ amount တို့ကို တိတိကျကျ သိမ်းဆည်းနိုင်ဖို့ အထောက်အကူ ပြုပါတယ်။

4.2 Blog Platform Schema

ဒီတစ်ခါ blog platform အတွက် နမူနာ schema တွေကို ကြည့်ကြရအောင်ပါ။

```
CREATE TABLE users (  
    user_id INT PRIMARY KEY,  
    username VARCHAR(50),  
    email VARCHAR(100) UNIQUE  
);  
  
CREATE TABLE posts (  
    post_id INT PRIMARY KEY,  
    user_id INT,  
    title VARCHAR(100),  
    content TEXT,  
    post_date DATE,  
    FOREIGN KEY (user_id) REFERENCES users(user_id)  
);  
  
CREATE TABLE comments (  
    comment_id INT PRIMARY KEY,  
    post_id INT,  
    user_id INT,  
    comment_text TEXT,  
    comment_date DATE,  
    FOREIGN KEY (post_id) REFERENCES posts(post_id),  
    FOREIGN KEY (user_id) REFERENCES users(user_id)  
);  
  
CREATE TABLE categories (  
    category_id INT PRIMARY KEY,  
    category_name VARCHAR(50) NOT NULL UNIQUE  
);  
  
CREATE TABLE post_categories (  
    post_id INT NOT NULL,  
    category_id INT NOT NULL,  
    PRIMARY KEY (post_id, category_id),  
    FOREIGN KEY (post_id) REFERENCES posts(post_id),  
    FOREIGN KEY (category_id) REFERENCES categories(category_id)  
);
```

```

CREATE TABLE tags (
    tag_id INT PRIMARY KEY,
    tag_name VARCHAR(50) NOT NULL UNIQUE
);

CREATE TABLE post_tags (
    post_id INT NOT NULL,
    tag_id INT NOT NULL,
    PRIMARY KEY (post_id, tag_id),
    FOREIGN KEY (post_id) REFERENCES posts(post_id),
    FOREIGN KEY (tag_id) REFERENCES tags(tag_id)
);

CREATE TABLE likes (
    like_id INT PRIMARY KEY,
    user_id INT NOT NULL,
    post_id INT NOT NULL,
    like_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(user_id),
    FOREIGN KEY (post_id) REFERENCES posts(post_id)
);

```

Foreign key တွေကိုပဲ အဓိကထားပြီး ရှင်းပြပေးပါမယ်။

posts table မှာရှိတဲ့ **user_id** ကတော့ ဘယ် user က post တင်ထားတာလဲဆိုတာပြသဖို့ **users** table ထဲက **user_id** ကို foreign key အဖြစ် ကိုးကားထားပါတယ်။

comments table ထဲမှာတော့ **post_id** နဲ့ **user_id** တို့ကို foreign keys အဖြစ်သုံးထားပြီး ဘယ် post ပေါ်မှာ ဘယ် user က comment ရေးထားတာကို ပြသနိုင်အောင်လို့ပါ။

post_categories table မှာတော့ post တစ်ခုကို category တစ်ခု (သို့မဟုတ်) များစွာနဲ့ ချိတ်ဆက်နိုင်အောင် design ဆွဲထားပြီး **post_id** နဲ့ **category_id** တို့ကို foreign key အဖြစ်သုံးထားပါတယ်။

post_tags table က post တစ်ခုမှာ tags တွေ တပ်နိုင်အောင် **posts** နဲ့ **tags** table တွေရဲ့ primary key တွေဖြစ်တဲ့ **post_id** နဲ့ **tag_id** ကို foreign key အဖြစ် ချိတ်ဆက်ထားတာပါ။

ဒီနေရာမှာ **post_categories** နဲ့ **post_tags** တို့က many-to-many relationship ကို ဖော်ပြပေးတဲ့ association tables တွေ ဖြစ်ပါတယ်။ posts တစ်ခုဟာ categories တစ်ခုထက်ပိုပြီး

ပတ်သက်နိုင်သလို category တစ်ခုဟာလည်း posts များစွာနဲ့ ဆက်နွှယ်နေနိုင်ပါတယ်။ ထိုနည်းတူပင် post တစ်ခုမှာ tags များစွာ တပ်နိုင်သလို tag တစ်ခုကို posts များစွာမှာ သုံးနိုင်ပါတယ်။

ဒါကြောင့် **post_categories** နဲ့ **post_tags** ဆိုတဲ့ သီးခြား **connector tables** မရှိဘဲနဲ့ တိုက်ရိုက် posts နဲ့ categories (သို့) posts နဲ့ tags တို့ကို **JOIN** လုပ်ဖို့ နည်းလမ်းမရှိပါဘူး။ ဒါကြောင့် ဒီလိုနေရာမှာ **many-to-many relationship design** ကို မဖြစ်မနေလိုအပ်တာ ဖြစ်ပါတယ်။

likes table မှာတော့ ဘယ် user ကဖြင့် ဘယ် post ကို like လုပ်ထားသလဲဆိုတာကို database ထဲမှာ တိတိကျကျ သိမ်းဆည်းနိုင်ဖို့ **user_id** နဲ့ **post_id** တို့ကို foreign key အဖြစ် အသုံးပြုထားပါတယ်။

5. Exercises

လေ့ကျင့်ခန်းလေးတွေ ထည့်ပေးတယ်ဆိုတာက database design ဆွဲတာကို ကိုယ်တိုင်လုပ်ကြည့်ပြီး ပိုနားလည်သွားအောင်လို့ပါ။ မေးခွန်းတစ်ခုချင်းစီကို သေချာအချိန်ပေးပြီး လေ့ကျင့်ဖြေဆိုကြည့်ပါ။

// Source Code Link:

<https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-10/16-students-table-setup.sql>

```
exercises_db=# SELECT * FROM students;
 student_id | student_name | class
-----+-----+-----
          1 | Ko Ko       | Grade5
          2 | Nyi Nyi    | Grade7
          3 | Su Lay     | Grade5
          4 | Hnin Eain  | Grade7
(4 rows)
```

// Source Code Link:

<https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-10/17-scores-table-setup.sql>

```
exercises_db=# SELECT * FROM scores;
 score_id | student_id | subject | score | exam_date
-----+-----+-----+-----+-----
          1 |          1 | Myanmar |    85 | 2025-05-01
          2 |          1 | Math   |    90 | 2025-05-01
          3 |          2 | Myanmar |    78 | 2025-05-02
          4 |          3 | Math   |    92 | 2025-05-01
          5 |          4 | Myanmar |    80 | 2025-05-02
(5 rows)
```

Question 1:

Write the SQL **CREATE TABLE** statements for the **students** and **scores** tables, including primary keys and foreign keys.

Question 2:

Add a **CHECK constraint** to the score column in the **scores** table (The score must be between 0 and 100).

Question 3:

Add a **UNIQUE constraint** to the **student_name** column in the **students** table.

Question 4:

Write the SQL statement to create an **index** on the **student_id** column in the **scores** table.

Question 5:

Design a **database schema** for a school library, which must include the following tables: **books**, **borrowers**, and **loans**.

Chapter 11: Modifying Data

ရှေ့ကအခန်းတွေမှာ data တွေကို ဘယ်လို modify လုပ်ရမလဲဆိုတာနဲ့ ပတ်သက်ပြီး မသိခဲ့ကြသေးပါဘူး။ ဒီအခန်းမှာတော့ မိမိလိုချင်သလို data တွေကို ဘယ်လိုထည့်မလဲ၊ ဘယ်လိုပြင်မလဲ၊ ပြီးတော့ ဘယ်လိုဖျက်ကြမလဲ ဆိုတာတွေကို လေ့လာကြမှာပါ။

SQL မှာတော့ data တွေကို modify လုပ်ဖို့ **INSERT**, **UPDATE**, **DELETE** ဆိုတဲ့ command တွေကို သုံးရပါတယ်။ ဒါပေမဲ့ data တွေ မှန်မှန်ကန်ကန် သိမ်းထားနိုင်ဖို့အတွက်တော့ transactions နည်းလမ်းကို အသုံးပြုကြပါတယ်။

1. INSERT: Adding New Data

INSERT command ဟာ database tables တွေထဲမှာ new records တွေကိုထည့်ဖို့ အသုံးပြုပါတယ်။

1.1 Single-Row Inserts

Database table ထဲကို မှတ်တမ်းတစ်ခုတည်း (single record) ကို တစ်ကြိမ်တည်းနဲ့ ထည့်ချင်တဲ့အခါ **INSERT** command ကို သုံးပါတယ်။

ဥပမာ – အရင်ကတည်ဆောက်ထားပြီးသား **customers** table ထဲကို customer အသစ်တစ်ဦး ထည့်ချင်တယ်ဆိုပါစို့။

```
INSERT INTO customers (customer_id, customer_name, phone)
VALUES (4, 'John Doe', '09-44444444');
```

INSERT INTO ဆိုတာကတော့ data ထည့်မယ့် table နာမည်နဲ့ column name တွေကို ဖော်ပြဖို့ သုံးတာဖြစ်ပါတယ်။

VALUES ကတော့ အဲ့ဒီ column တွေအတွက် ထည့်မယ့်တန်ဖိုး (data) တွေကို ရေးဖို့ဖြစ်ပါတယ်။

```
[exercises_db=# SELECT * FROM customers;
 customer_id | customer_name | phone
-----+-----+-----
          1 | John Smith   | 09-11111111
          2 | Mary Johnson | 09-22222222
          3 | Michael Anderson | 09-33333333
          4 | John Doe     | 09-44444444
(4 rows)
```

1.2 Bulk Inserts

Data record များစွာကို တစ်ခါတည်း database ထဲကို ထည့်ချင်တဲ့အခါမှာ **Bulk Insert** ကို အသုံးပြုပါတယ်။

ဒီနည်းမှာတော့ **INSERT INTO** command တစ်ခုပဲရေးပြီး **VALUES** အတွင်းမှာ record တစ်ခုချင်းစီကို comma (,) နဲ့ ချိတ်ဆက်ပေးရတာဖြစ်ပါတယ်။

ဥပမာ - **orders** table ထဲကို အသစ်မှာထားတဲ့ order တွေကို တစ်ခါတည်းထည့်မယ်ဆိုရင် ဒီလိုပဲလုပ်နိုင်ပါတယ်။

```
INSERT INTO orders (order_id, customer_id, product, amount, order_date)
VALUES
  (4, 3, 'noodles', 18000, '2025-06-10'),
  (5, 3, 'soap', 8000, '2025-06-11'),
  (6, 4, 'milk', 10000, '2025-06-12');
```

ယခင်ရှိပြီးသား table ထဲကို row အသစ်သုံးခု တစ်ခါတည်းထည့်သွင်းပြီးသား ဖြစ်သွားတာကို တွေ့ရမှာပါ။

```
[exercises_db=# SELECT * FROM orders;
 order_id | customer_id | product | amount | order_date
-----+-----+-----+-----+-----
          1 |          1 | rice    | 25000 | 2025-05-01
          2 |          1 | oil     | 15000 | 2025-05-02
          3 |          2 | sugar   | 12000 | 2025-05-03
          4 |          3 | noodles | 18000 | 2025-06-10
          5 |          3 | soap   | 8000  | 2025-06-11
          6 |          4 | milk    | 10000 | 2025-06-12
(6 rows)
```

2. UPDATE: Modifying Existing Data

UPDATE keyword ကို သုံးပြီး table ထဲမှာ ရှိပြီးသား record တွေကို ပြန်ပြင်နိုင်ပါတယ်။

2.1 Updating Specific Rows

ဥပမာ - **customer_id** 1 ဖြစ်တဲ့ customer ရဲ့ ဖုန်းနံပါတ်ကို ပြောင်းချင်တဲ့အခါမှာ ဒီလိုရေးနိုင်ပါတယ်။

```
UPDATE customers
SET phone = '09-10101010'
WHERE customer_id = 1;
```

UPDATE customers ဆိုတာက **customers** table ထဲက data ကို ပြင်ဖို့သုံးတဲ့ command ပါ။ **SET** မှာ ပြင်ချင်တဲ့ column နဲ့ တန်ဖိုးအသစ်တွေကို ထည့်ပေးရပါတယ်။ **WHERE** ကတော့ ဘယ် record (row) ကို ပြင်မလဲဆိုတာ သတ်မှတ်ဖို့ အသုံးပြုတာ ဖြစ်ပါတယ်။

WHERE မထည့်ပဲ **UPDATE** လုပ်လိုက်ရင် table ထဲက record အားလုံးကို ပြောင်းပေးသွားသလို ဖြစ်သွားမှာ ပါ။ ဒါကြောင့် အရမ်းသတိထားဖို့ လိုပါတယ်။

2.2 Using Conditions Safely

WHERE keyword ကို မထည့်ဘဲ **UPDATE** လုပ်လိုက်ရင် table ထဲမှာရှိတဲ့ data record အကုန်လုံး ပြောင်းလဲသွားတာကို လက်တွေ့ပြပါမယ်။

```
UPDATE customers
SET phone = '09-99999999';
```

အခုလိုမျိုး **phone** column ထဲက record အကုန်လုံးရဲ့တန်ဖိုးတွေဟာ တူတူဖြစ်သွားပါတယ်။

```
[exercises_db=# SELECT * FROM customers;
 customer_id | customer_name | phone
-----+-----+-----
          1 | John Smith   | 09-99999999
          2 | Mary Johnson | 09-99999999
          3 | Michael Anderson | 09-99999999
          4 | John Doe     | 09-99999999
(4 rows)
```


3. DELETE: Removing Data

DELETE ဆိုတဲ့အတိုင်း table ထဲက data record တွေကို ဖယ်ရှားပစ်ဖို့ အသုံးပြုတဲ့ keyword ပါ။

3.1 Deleting Specific Rows

တချို့ record တွေကိုသာ ဖျက်ချင်ရင် **UPDATE** မှာလိုပဲ **WHERE** ကိုသုံးရပါမယ်။ ဥပမာ - **order_id** က 2 ဖြစ်နေတဲ့ row ကို delete လုပ်ချင်တယ်ဆိုရင်။

```
DELETE FROM orders
WHERE order_id = 2;
```

order_id 2 ဖြစ်တဲ့ row မရှိတော့တာ တွေ့ရမှာပဲ ဖြစ်ပါတယ်။

```
[exercises_db=# SELECT * FROM orders;
 order_id | customer_id | product | amount | order_date
-----+-----+-----+-----+-----
      1 |          1 | rice   |  25000 | 2025-05-01
      3 |          2 | sugar  |  12000 | 2025-05-03
      4 |          3 | noodles|  18000 | 2025-06-10
      5 |          3 | soap  |   8000 | 2025-06-11
      6 |          4 | milk   |  10000 | 2025-06-12
(5 rows)
```

3.2 Truncating Tables

TRUNCATE က table တစ်ခုလုံးထဲက record အားလုံးကို ဖျက်ပစ်လိုက်တာပဲ ဖြစ်ပါတယ်။ ဒါပေမဲ့ table structure ကိုတော့ မဖျက်ပါဘူး။ ကွာခြားချက်တစ်ခုအနေနဲ့တော့ **DELETE** နဲ့ **TRUNCATE** တို့မှာ performance ပိုင်းအရ **TRUNCATE** က ဖျက်တာပိုမြန်ပါတယ်။

ဥပမာ - **orders** table ထဲက data အားလုံးကို ဖျက်ချင်ရင်။

```
TRUNCATE TABLE orders;
```

တစ်ခုသတိပြုရမှာက **TRUNCATE** ကို once run လုပ်လိုက်ရင် **undo** ပြန်လုပ်ဖို့ မလွယ်ကူတော့ပါဘူး။

```
[exercises_db=# SELECT * FROM orders;
 order_id | customer_id | product | amount | order_date
-----+-----+-----+-----+-----
(0 rows)
```

Data အားလုံးကို ဖျက်လိုက်တာကြောင့် ဘာမှမကျန်တော့တာ တွေ့ရမှာပဲ ဖြစ်ပါတယ်။

4. Transactions and Data Integrity

4.1 ACID Properties

Database transactions တွေဟာ data integrity ကောင်းမွန်ဖို့အတွက် **ACID** လို့ခေါ်တဲ့ measurement လေးခုနဲ့ ပြည့်စုံဖို့လိုပါတယ်။

→ **Atomicity:** Database ပေါ်မှာ အပြောင်းအလဲ (transaction) တစ်ခုလုံးအားဖြင့် ပြီးမြောက်ရပါ မယ်။ တချို့တဝက်လောက်သာ ပြီးမြောက်တာမျိုး မဟုတ်ဘဲ အကုန်ပြီးမြောက်ရပါမယ်။ မဟုတ် ဘူးဆိုရင် အကုန်လုံးပြန်ဖျက်သွားရမယ်ဆိုတဲ့ သဘောပါ။

→ **Consistency:** Transaction တစ်ခု ပြီးတဲ့အခါမှာ database က original rules (constraints, schema) တွေနဲ့အညီ မှန်ကန်နေဖို့ လိုပါတယ်။

→ **Isolation:** Transaction တစ်ခုချင်းစီကို သီးခြားစီမံလုပ်ဆောင်ရပါမယ်။ တစ်ခုလုပ်နေစဉ်မှာ တခြား transaction တွေကို သက်ရောက်မှုမရှိအောင် ထိန်းထားရပါမယ်။

→ **Durability:** Transaction တစ်ခု ပြီးမြောက်သွားတာနဲ့ ပြောင်းလဲမှုတွေဟာ permanent ဖြစ်နေဖို့ လိုပါတယ်။ System crash ဖြစ်သွားလည်း data တွေဟာ မပျက်သင့်တော့ပါဘူး။

4.2 BEGIN, COMMIT, ROLLBACK

Transaction အပြောင်းအလဲတွေကို စနစ်တကျလုပ်ဆောင်နိုင်ဖို့ **BEGIN**, **COMMIT**, နဲ့ **ROLLBACK** စတဲ့ keyword တွေကို အသုံးပြုပါတယ်။

ဥပမာ - နောက်ထပ် customer တစ်ဦးနဲ့ order အသစ်ကို database ထဲ ထည့်သွင်းတဲ့လုပ်ဆောင်မှုကို စဉ်းစားကြည့်ပါစို့။

Case 1 – Transaction Success (BEGIN + COMMIT)

```
BEGIN;  
  
INSERT INTO customers (customer_id, customer_name, phone)  
VALUES (8, 'Anna', '09-88888888');  
  
INSERT INTO orders (order_id, customer_id, product, amount, order_date)  
VALUES (7, 8, 'books', 10000, '2025-06-05');  
  
COMMIT;
```

အထက်က example မှာ **BEGIN** နဲ့ စပြီး အောင်မြင်စွာပြီးဆုံးတာနဲ့ **COMMIT** လုပ်လိုက်ပါတယ်။

Case 2 – Transaction Failure (BEGIN + ROLLBACK)

```
BEGIN;  
  
INSERT INTO customers (customer_id, customer_name, phone)  
VALUES (9, 'Alex Wan', '09-35353535');  
  
INSERT INTO orders (order_id, customer_id, product, amount, order_date)  
VALUES (8, 999, 'butter', 5000, '2025-06-13');  
  
ROLLBACK;
```

တကယ်တော့ **customer_id 999** ဟာ **customers** table မှာ မရှိပါဘူး။ Foreign key constraint မှားတဲ့အတွက် error ဖြစ်လာပါမယ်။ ဒီတော့ **ROLLBACK** လုပ်ပြီး transaction တစ်ခုလုံးကို ပြန်ဖျက်သိမ်းမှာ ဖြစ်ပါတယ်။

5. Exercises

ယခင်ရှိပြီးသား table နှစ်ခုပေါ်မှာပဲ အခြေခံပြီး လေ့ကျင့်ခန်းမေးခွန်းတွေ ဖြေဆိုကြည့်နိုင်ပါတယ်။

```
[exercises_db=# SELECT * FROM students;
 student_id | student_name | class
-----+-----+-----
          1 | Ko Ko       | Grade5
          2 | Nyi Nyi     | Grade7
          3 | Su Lay      | Grade5
          4 | Hnin Eain   | Grade7
(4 rows)
```

```
[exercises_db=# SELECT * FROM scores;
 score_id | student_id | subject | score | exam_date
-----+-----+-----+-----+-----
          1 |          1 | Myanmar |    85 | 2025-05-01
          2 |          1 | Math    |    90 | 2025-05-01
          3 |          2 | Myanmar |    78 | 2025-05-02
          4 |          3 | Math    |    92 | 2025-05-01
          5 |          4 | Myanmar |    80 | 2025-05-02
(5 rows)
```

Question 1:

Write an SQL **INSERT** command to add a new student (**student_id**: 5, **student_name**: Mya Mya, **class**: Grade6) to the **students** table using the **VALUES** clause.

Question 2:

Write an SQL **INSERT** command to insert two records into the **scores** table at once:

(**score_id**: 6, **student_id**: 4, **subject**: Math, **score**: 92, **date**: 2025-05-01)

(**score_id**: 7, **student_id**: 5, **subject**: Myanmar, **score**: 80, **date**: 2025-05-02)

Question 3:

Write an SQL **UPDATE** command to change the class of the student with **student_id** = 1 to "Grade6".

Question 4:

Write an SQL **DELETE** command to remove the record with **score_id** = 3 from the **scores** table.

Question 5:

Write SQL commands using **BEGIN** and **COMMIT** to insert a new student and their score into the **students** and **scores** tables, respectively.

Chapter 12: Performance Optimization

SQL မှာ database queries တွေကို ပိုပြီးမြန်မြန်ဆန်ဆန်နဲ့ ထိရောက်စွာလုပ်ဆောင်နိုင်ဖို့ရယ်၊ ပြီးတော့ performance ကောင်းအောင်လုပ်ပေးဖို့ အရေးကြီးတယ်ဆိုတာကိုလည်း ရှေ့အခန်းတွေကတည်းက သိလာခဲ့ပါပြီ။ ဒီအခန်းမှာတော့ query execution plans ကို နားလည်သဘောပေါက်ဖို့၊ indexing strategies ကို စနစ်တကျ အသုံးပြုနည်း၊ ထိရောက်တဲ့ query ရေးသားနည်းနဲ့ performance ကို ကျဆင်းစေနိုင်တဲ့ bottlenecks အကြောင်းတွေကို ထဲထဲဝင်ဝင် လေ့လာသွားမှာ ဖြစ်ပါတယ်။

1. Understanding Query Execution Plans

1.1 Reading and Interpreting Plans

Query Execution Plan ဆိုတာက SQL query တစ်ခုကို database engine က ဘယ်လိုအဆင့်ဆင့်နဲ့ တိတိကျကျ လုပ်ဆောင်သွားမလဲ ဆိုတာကို ပြသပေးတဲ့ process flow တစ်ခုလိုပါပဲ။ ဒီ plan ထဲမှာတော့ data ကို ဘယ်လိုရှာမလဲ (**data lookup**), table တွေကို ဘယ်လို **JOIN** မလဲ၊ ဘယ်လို **sorting** လုပ်မလဲ စတဲ့အဆင့်တွေကို တစ်ခုချင်းစီ သီးသန့်ပြပေးပါတယ်။ ဒါကြောင့် **Query Execution Plan** ကို ဖတ်တတ်လာတဲ့နဲ့ query performance ကို မြှင့်တင်နိုင်မယ့် နည်းလမ်းတွေကို သိလာနိုင်ပါတယ်။

ဥပမာ - customer နှင့် ၎င်းတို့ရဲ့ order တွေကို ရှာဖွေမယ်ဆိုပါစို့။

```
EXPLAIN SELECT c.customer_name, o.product, o.amount
FROM Customers c
JOIN Orders o ON c.customer_id = o.customer_id
WHERE o.amount > 10000;
```

```
exercises_db=# EXPLAIN SELECT c.customer_name, o.product, o.amount
exercises_db=# FROM Customers c
exercises_db=# JOIN Orders o ON c.customer_id = o.customer_id
exercises_db=# WHERE o.amount > 10000;
               QUERY PLAN
-----
Hash Join  (cost=15.85..32.55 rows=167 width=340)
  Hash Cond: (o.customer_id = c.customer_id)
    -> Seq Scan on orders o  (cost=0.00..16.25 rows=167 width=126)
        Filter: (amount > 10000)
    -> Hash  (cost=12.60..12.60 rows=260 width=222)
        -> Seq Scan on customers c  (cost=0.00..12.60 rows=260 width=222)
(6 rows)
```

တစ်ဆင့်ချင်း နားလည်အောင် ရှင်းပြပါမယ်။

အခန်း(10)မှာတုန်းက **CREATE INDEX idx_customer_id ON orders(customer_id);** ဆိုပြီး index တစ်ခု ဖန်တီးခဲ့ပေမယ့် အခုကျန်တော့ **JOIN** လုပ်တဲ့ table တွေဟာ အရွယ်အစားသေးငယ်လို့ PostgreSQL ရဲ့ **EXPLAIN** ဟာ **Index Scan** ကို မသုံးဘဲ **Seq Scan** နဲ့ လုပ်ဆောင်ပေးတာ ဖြစ်ပါတယ်။

→ Seq Scan on customers c

customers table ပေါ်မှာ **Sequential Scan** လုပ်နေတယ်။ ဆိုလိုတာက **customers** table ထဲမှာ ရှိတဲ့ row အကုန်လုံးကို တစ်ခုချင်းစီ စစ်နေတယ်ဆိုတဲ့ သဘောပါ။ **Index** မသုံးပဲ တစ်ကြောင်းချင်း စစ်တဲ့နည်း ဖြစ်ပါတယ်။

→ Hash Cond: (customer_id = c.customer_id)

Table တွေကို **JOIN** လုပ်မယ့် condition ကို ရည်ညွှန်းပါတယ်။ **orders** table ရဲ့ **customer_id** ကို **customers** table ရဲ့ **customer_id** နဲ့ ကိုက်ညီမှုရှိမရှိ စစ်ပါတယ်။ **Index** ကိုဘယ်အချိန်မှာ အသုံးပြု နိုင်မလဲဆိုတာကို ဒီနေရာမှာ ကြည့်လို့ရပါတယ်။

→ Filter: (amount > 10000)

orders table ထဲမှာ **amount > 10000** ဖြစ်တဲ့ row တွေကိုသာ ပြန်ပေးမှာဖြစ်တယ်။ ဒီ filter က **WHERE** condition ကိုဖြစ်ပြီး scan ပြီးတဲ့နောက်မှာ result တွေကို စစ်ထုတ်ပေးတာပါ။

→ (cost=0.00..16.25 rows=167 width=126)

cost ဆိုတာက PostgreSQL ရဲ့ internal ခန့်မှန်းချက်ဖြစ်ပြီး query တစ်ခုကို run လုပ်ဖို့ CPU, disk I/O စတဲ့ resource များကို ဘယ်လောက် လိုမလဲဆိုတာ ခန့်မှန်းပြထားတာ ဖြစ်ပါတယ်။

0.00 က **startup cost** ဖြစ်ပြီး operation ကိုစတင်ဖို့ လိုအပ်တဲ့ resource ပမာဏပါ။

16.25 က **total cost** ဖြစ်ပြီး operation တစ်ခုလုံးကို ပြီးမြောက်အောင်လုပ်ဆောင်ဖို့ PostgreSQL ခန့်မှန်းထားတဲ့ resource စုစုပေါင်းပါ။

rows=167 ဆိုတာက PostgreSQL ရဲ့ ခန့်မှန်းချက်အရ ဒီ operation ကနေ 167 row လောက်ကို result အနေနဲ့ return လုပ်မယ်လို့ ယူဆထားတာပါ။

width=126 ဆိုတာက row တစ်ခုချင်းမှာ average ဖြစ်နိုင်တဲ့ data size 126 bytes လောက်ရှိမယ်လို့ PostgreSQL က ခန့်မှန်းထားတာ ဖြစ်ပါတယ်။

1.2 Identifying Bottlenecks

- **Full Table Scan: Index** မရှိတဲ့ column တစ်ခုမှာ ရှာဖွေမှုလုပ်တဲ့အခါ table တစ်ခုလုံးကို မပြီးမချင်း တစ်ခုပြီးတစ်ခုစစ်ရတာကြောင့် query speed နှေးလာနိုင်ပါတယ်။
- **Unnecessary Joins:** မဖြစ်မနေ လိုအပ်တဲ့အချိန်ကျမှ joins ကို သုံးတာမျိုးမဟုတ်ဘဲ မလိုအပ်ဘဲ joins တွေ ထည့်သုံးတာကလည်း query speed ကို နည်းစေပါတယ်။
- **Sorting Large Data:** Data ပမာဏအများကြီးကို အစီစဉ်ချရတာသည်လည်း query performance ကို ထိခိုက်စေနိုင်ပါတယ်။

ဥပမာ - **orders** table ထဲမှာ **amount > 10000** ဆိုတဲ့ condition နဲ့ query လုပ်ချင်တဲ့အခါ **amount** column ပေါ်မှာ **index** မရှိဘူးဆိုရင် database က **orders** table တစ်ခုလုံးကို row အနေနဲ့ အစဉ်လိုက်စစ်ရပါမယ်။ အဲ့ဒီလိုဆက်သွားနေရင် **Full Table Scan** ဖြစ်ပြီး query execution time လည်း ကြာမှာဖြစ်သလို performance လည်း ကောင်းလာမှာ မဟုတ်ပါဘူး။

2. Indexing Strategies

2.1 Choosing Columns to Index

Index တွေဟာ data ရှာဖွေမှုကို မြန်ဆန်စေရင်လည်း storage space ပိုလိုအပ်တာမို့ data manipulation ကို နှေးကွေးစေနိုင်တယ်ဆိုတာကို ရှေ့အခန်းမှာလည်း ဖော်ပြခဲ့ပြီး ဖြစ်ပါတယ်။

Indexing လုပ်ဖို့ သင့်လျော်တဲ့ column များ -

- **WHERE** clause မှာ မကြာခဏ အသုံးပြုရတဲ့ column (ဥပမာ - **customer_id**, **order_date**)။
- **JOIN** condition များတွင် ပါဝင်တဲ့ column (ဥပမာ - **customer_id**)။
- **ORDER BY** သို့မဟုတ် **GROUP BY** တွင် ပါဝင်တဲ့ column များ။

ဥပမာ - **orders** table အတွက် **order_date** column ကို indexing လုပ်မယ်ဆိုရင်။

```
CREATE INDEX idx_order_date ON orders(order_date);
```

2.2 Composite Indexes

Composite indexes ဆိုတာကတော့ columns နှစ်ခု သို့မဟုတ် ထို့ထက်ပိုများတဲ့ columns တွေကို တွဲဖက်ပြီး တစ်ခုတည်းဖြစ်အောင် ပြုလုပ်ထားတဲ့ index ပါ။

ဥပမာ - **orders** table မှာ **customer_id** နဲ့ **order_date** ကို တွဲပြီး index ဖန်တီးချင်ရင် ဒီလိုရေးလို့ ရပါတယ်။

```
CREATE INDEX idx_customer_date ON orders(customer_id, order_date);
```

ဒီလို composite index တစ်ခု ဖန်တီးလိုက်ရင်တော့ **customer_id** နဲ့ **order_date** နှစ်ခုလုံးကို အသုံးပြုတဲ့ query တွေမှာ ရှာဖွေမှု ပိုမြန်လာမှာ ဖြစ်ပါတယ်။

```
SELECT product, amount
FROM orders
WHERE customer_id = 1 AND order_date ≥ '2025-05-01';
```

ဖန်တီးပြီးသား index key တွေကို ပြန်စစ်ဆေးချင်တယ်ဆိုရင် ဒီ query ကို အသုံးပြုလို့ရပါတယ်။

```
SELECT indexname, indexdef
FROM pg_indexes
WHERE tablename = 'orders';
```

အခုလိုတွေ့ရမှာပဲ ဖြစ်ပါတယ်။

```
exercices_db=# SELECT indexname, indexdef
exercices_db=# FROM pg_indexes
exercices_db=# WHERE tablename = 'orders';
 indexname          | indexdef
-----+-----
orders_pkey         | CREATE UNIQUE INDEX orders_pkey ON public.orders USING btree (order_id)
idx_customer_id     | CREATE INDEX idx_customer_id ON public.orders USING btree (customer_id)
idx_customer_date   | CREATE INDEX idx_customer_date ON public.orders USING btree (customer_id, order_date)
idx_order_date      | CREATE INDEX idx_order_date ON public.orders USING btree (order_date)
(4 rows)
```

3. Query Optimization Tips

3.1 Writing Efficient Queries

SELECT * ကို မလိုအပ်ဘဲ သုံးတာမျိုး ရှောင်ပါ။ ဘာလို့လဲဆိုတော့ * သုံးတဲ့အခါ database table ထဲမှာ ရှိတဲ့ column အားလုံးကိုဖတ်ဖို့ လိုလာပါတယ်။ ဒါကြောင့် လိုအပ်တဲ့ column တွေလောက်ပဲ ရွေးချယ်ပါ။

```
SELECT * FROM customers;
```

အပေါ်ကလိုမျိုး * နဲ့ရေးခြင်းဟာ တကယ်တမ်း မထိရောက်ပါဘူး။ အဲ့ဒီအစား အောက်ကလိုမျိုး လိုအပ်တဲ့ column တွေကိုသာ ရွေးပြီး ရေးသင့်ပါတယ်။

```
SELECT customer_name, phone FROM customers;
```

ပြီးတော့ **WHERE** clause ကို တိတိကျကျ သုံးပါ။ မလိုအပ်တဲ့ data တွေ ပါမလာအောင် ထိထိရောက်ရောက် filtering လုပ်တာက scanning workload ကို လျှော့ချပြီး query ပိုမြန်စေမှာပါ။

```
SELECT product FROM orders WHERE order_date = '2025-05-01';
```

ဒီလို query ရေးတာက database ကို မလိုအပ်တဲ့ row အကုန် စစ်ခိုင်းဖို့မလိုဘဲ တကယ်လိုအပ်တဲ့ data တွေကိုသာ ရှာဖွေနိုင်စေပါတယ်။ ဒါကြောင့် system ရော speed ရော ပိုကောင်းလာပါမယ်။

3.2 Avoiding Unnecessary Joins

မလိုအပ်တဲ့ **JOIN** တွေကလည်း query performance ကို ထိခိုက်စေနိုင်ပါတယ်။

// Inefficient query

```
SELECT c.customer_name
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE c.customer_id = 3;
```

// Efficient query

```
SELECT customer_name
FROM customers
WHERE customer_id = 3;
```

orders table က data မသုံးဘူးဆိုရင် **JOIN** မလုပ်တာပဲ ကောင်းပါတယ်။ မလိုအပ်တဲ့ **JOIN** တစ်ခုကြောင့် query ပိုနှေးနိုင်တယ်ဆိုတာ အမြဲသတိရပါ။

4. Avoiding Common Performance Bottlenecks

4.1 Overuse of Wildcards

ရှေ့မှာပြောခဲ့သလို ***** ကို အသုံးပြုပြီး column အားလုံးကို **SELECT** လုပ်တာဟာ မလိုအပ်တဲ့ data တွေကို လည်း ရယူနေရလို့ query performance ကို နှေးစေတဲ့အပြင် server နဲ့ network ပေါ်မှာ workload ကို တိုးစေပါတယ်။

```
SELECT * FROM orders WHERE customer_id = 1;
```

တကယ်လိုအပ်တာက data အနည်းငယ်ပဲဆိုရင် **SELECT *** နဲ့ကို မရေးသင့်တာပါ။ ဒါပေမယ့် development လုပ်တဲ့အချိန် ဒါမှမဟုတ် testing လုပ်တဲ့အခါတွေမှာတော့ ရေးလည်း ပြဿနာမရှိပါဘူး။

```
SELECT product, amount FROM orders WHERE customer_id = 1;
```

လိုအပ်တဲ့ columns တွေကိုပဲ ရွေးရေးတာက query ကို ပိုမြန်ဆန်စေပြီး server နဲ့ network ပေါ်မှာ load နည်းစေပါတယ်။ ထပ်ပြောပါမယ်။ ဒီလိုလုပ်ခြင်းက database performance ကောင်းမွန်စေပြီး resource အသုံးပြုမှုကိုလည်း လျော့နည်းစေပါတယ်။

4.2 Suboptimal Subqueries

Subquery တွေကို မလိုအပ်ဘဲ သုံးနေမယ်ဆိုရင် query performance လည်း ပိုနှေးလာပါမယ်။ System ပေါ်မှာလည်း extra load ဖြစ်လာပါမယ်။ ဒါကြောင့် subquery အစား **CTE (Common Table Expressions)** သုံးတာ၊ ဒါမှမဟုတ် **JOIN** နဲ့ ပြန်ရေးတာက ပိုထိရောက်ပါတယ်။

// Inefficient query

```
SELECT customer_name
FROM customers
WHERE customer_id IN (
    SELECT customer_id
    FROM orders
    WHERE amount > 20000
);
```

// Efficient query

```
SELECT c.customer_name
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.amount > 20000;
```

5. Exercises**// students table**

```
[exercises_db=# SELECT * FROM students;
 student_id | student_name | class
-----+-----+-----
          1 | Ko Ko       | Grade5
          2 | Nyi Nyi     | Grade7
          3 | Su Lay      | Grade5
          4 | Hnin Eain   | Grade7
(4 rows)
```

// scores table

```
[exercises_db=# SELECT * FROM scores;
 score_id | student_id | subject | score | exam_date
-----+-----+-----+-----+-----
          1 |          1 | Myanmar |    85 | 2025-05-01
          2 |          1 | Math    |    90 | 2025-05-01
          3 |          2 | Myanmar |    78 | 2025-05-02
          4 |          3 | Math    |    92 | 2025-05-01
          5 |          4 | Myanmar |    80 | 2025-05-02
(5 rows)
```

Question 1:

Write an SQL statement to create a **composite index** on the **scores** table for the **student_id** and **exam_date** columns.

Question 2:

Rewrite the following query to make it more efficient by selecting only the necessary columns.

```
SELECT * FROM scores WHERE student_id = 1;
```

Question 3:

Rewrite the following **subquery** using a **JOIN**.

```
SELECT student_name
FROM students
WHERE student_id IN (
    SELECT student_id
    FROM scores
    WHERE score > 80
);
```

Question 4:

Identify the performance issue in the following query and optimize it for better efficiency.

```
SELECT s.student_name, s.class
FROM students s
JOIN scores sc ON s.student_id = sc.student_id
WHERE s.class = 'Grade5';
```

Question 5:

Create an **index** on the **score** column in the **scores** table, and test the following query using **EXPLAIN**.

```
SELECT student_id, subject, score
```

```
FROM scores  
WHERE score > 85;
```


Chapter 13: SQL in the Real World

SQL ကို database management အပြင် data analysis, web development, နဲ့ business intelligence လုပ်ငန်းတွေမှာလည်း ကျယ်ကျယ်ပြန့်ပြန့်အသုံးပြုကြပါတယ်။ ဒီအခန်းမှာတော့ SQL ကို လက်တွေ့အသုံးချနိုင်မယ့် ဥပမာတွေနဲ့အတူ data analysis, web development, နဲ့ business decision making တို့မှာ ဘယ်လိုအသုံးဝင်လဲဆိုတာကို နားလည်အောင် ရှင်းပြသွားမှာပါ။

// Source Code Link:

[https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-13/01-create-products-table.s](https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-13/01-create-products-table.sql)
[ql](#)

```
[exercises_db=# SELECT * FROM products;
 product_id | product_name | category | price  | stock_quantity
-----+-----+-----+-----+-----
          1 | Rice 5kg    | Food    | 25000.00 |          100
          2 | Cooking Oil | Food    |  8000.00 |           50
          3 | Soap        | Personal |  3000.00 |          200
          4 | Shampoo     | Personal | 12000.00 |           75
(4 rows)
```

// Source Code Link:

[https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-13/02-create-employees-table.](https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-13/02-create-employees-table.sql)
[sql](#)

```
[exercises_db=# SELECT * FROM employees;
 employee_id | employee_name | department | salary  | hire_date
-----+-----+-----+-----+-----
          1 | Aye Aye      | Sales      | 500000.00 | 2024-01-15
          2 | Ko Ko        | IT          | 700000.00 | 2023-06-01
          3 | Su Su        | HR          | 450000.00 | 2024-03-10
          4 | Tun Tun      | Sales      | 550000.00 | 2023-09-20
(4 rows)
```

// Source Code Link:

<https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-13/03-transactions-table.sql>

```
[exercises_db=# SELECT * FROM transactions;
 transaction_id | order_id | payment_method | amount  | transaction_date
-----+-----+-----+-----+-----
              1 |         1 | Cash           | 25000.00 | 2025-05-01
              2 |         2 | Credit Card    | 15000.00 | 2025-05-02
              3 |         3 | Mobile Pay     | 15000.00 | 2025-05-03
              4 |         4 | Cash           |  8000.00 | 2025-05-04
(4 rows)
```

// Source Code Link:

<https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-13/04-create-and-insert-reviews.sql>

```
(exercises_db=# SELECT * FROM reviews;
 review_id | product_id | customer_id | rating |      comment      | review_date
-----+-----+-----+-----+-----+-----
          1 |          1 |           1 |       5 | Great quality rice! | 2025-05-02
          2 |          2 |           1 |       4 | Good oil, but small bottle | 2025-05-03
          3 |          1 |           2 |       3 | Average quality     | 2025-05-04
          4 |          3 |           3 |       5 | Love this soap!    | 2025-05-05
(4 rows)
```

1. SQL for Data Analysis

1.1 Exploratory Data Analysis

EDA ဆိုတာ SQL queries တွေကို သုံးပြီး data ထဲက patterns တွေ anomalies တွေနဲ့ valuable insights တွေကို ရှာဖွေတဲ့နည်းလမ်းပါ။ ဒီနည်းက data ရဲ့ သဘောသဘာဝကို နားလည်အောင် ကူညီပေးပြီး စီးပွားရေးဆိုင်ရာ ဆုံးဖြတ်ချက်တွေ ချမှတ်တဲ့အခါ အခြေခံအဓိကအချက်အဖြစ် ဆောင်ရွက်ပေးပါတယ်။ SQL နဲ့ဆိုရင် data ကို aggregate လုပ်တာ၊ filter လုပ်တာ၊ နဲ့ summarize လုပ်တာတွေကို လွယ်လွယ်ကူကူ လုပ်နိုင်ပြီး data ရဲ့ အသွားအလာပုံစံတွေကို မြန်မြန်ဆန်ဆန် ဖော်ထုတ်နိုင်ပါတယ်။

```
(exercises_db=# SELECT * FROM products;
 product_id | product_name | category | price  | stock_quantity
-----+-----+-----+-----+-----
          1 | Rice 5kg     | Food     | 25000.00 |          100
          2 | Cooking Oil  | Food     | 8000.00  |           50
          3 | Soap         | Personal | 3000.00  |          200
          4 | Shampoo     | Personal | 12000.00 |           75
(4 rows)
```

Practical Applications

Data Analysis

Product category အလိုက် ရောင်းအားပမာဏကို analyze လုပ်နိုင်ပါတယ်။ ဥပမာအားဖြင့် -

```
SELECT category, SUM(price * stock_quantity) AS total_stock_value FROM
products GROUP BY category;
```

အခုလို ရေးလိုက်တာနဲ့ သက်ဆိုင်တဲ့ data တွေကို လွယ်လင့်တကူ ဖော်ထုတ်နိုင်တာပါ။

Web Development

Website ပေါ်မှာ products list ကို ပြသချင်တယ်ဆိုရင်လည်း database ထဲက data ကို query လုပ်ပြီး အသုံးပြုနိုင်ပါတယ်။ ဒီလိုနဲ့ user တွေအတွက် တိတိကျကျ product display တစ်ခု ဖန်တီးလို့ ရသွားပါတယ်။

Business Intelligence

Low stock ဖြစ်ပြီး ရောင်းအားနည်းနေတဲ့ products တွေရှိရင်လည်း အဲ့ဒါတွေကို identify လုပ်ပြီး restock strategy တွေကို ပြန်စီစဉ်ရေးဆွဲနိုင်ပါတယ်။ ဒါကြောင့် data တွေဟာ business decision-making အတွက် ဘယ်လောက်အထောက်အကူပြုတယ်ဆိုတာ သိနိုင်ပါတယ်။

1.2 Generating Reports

အစီရင်ခံစာ (reports) ရေးဖို့ လိုအပ်တဲ့ information တွေကို database ကနေ SQL နဲ့ ဆွဲထုတ်နိုင်ပါတယ်။ ဥပမာ - **employees** table မှာရှိတဲ့ **department**, **salary**, နှင့် **hire_date** စတဲ့ data တွေကို အသုံးပြုပြီး လိုအပ်သလို employee report တွေ ထုတ်နိုင်ပါတယ်။

```

(exercises_db=# SELECT * FROM employees;
 employee_id | employee_name | department | salary | hire_date
-----+-----+-----+-----+-----
          1 | Aye Aye      | Sales      | 500000.00 | 2024-01-15
          2 | Ko Ko        | IT         | 700000.00 | 2023-06-01
          3 | Su Su        | HR         | 450000.00 | 2024-03-10
          4 | Tun Tun      | Sales      | 550000.00 | 2023-09-20
(4 rows)

```

Practical Applications

Data Analysis

Department အလိုက် ဝန်ထမ်းတွေရဲ့ ပျမ်းမျှလစာ (average salary) ကို တွက်ချက်ချင်တာမျိုးမှာ SQL ကို အသုံးပြုလို့ရပါတယ်။ ဥပမာ -

```
SELECT department, AVG(salary) FROM employees GROUP BY department;
```

ဆိုပြီး query ရေးလိုက်တာနဲ့ department အလိုက် ပျမ်းမျှလစာတွေကို သိနိုင်ပါတယ်။

Web Development

HR web app တစ်ခုတည်ဆောက်တဲ့အခါ ဝန်ထမ်းစာရင်းကို ပြချင်ရင် **employees** table ထဲက data ကို query လုပ်ပြီး website ပေါ်မှာ ပြနိုင်ပါတယ်။ အဲလိုလုပ်ရင် admin နဲ့ HR team ကနေ employee နဲ့ ပတ်သက်တဲ့ စာရင်းဇယားတွေလုပ်တဲ့အခါ ပိုမြန်ဆန်တိကျလာပါမယ်။

Business Intelligence

Turnover rate ခန့်မှန်းချင်ရင် **hire_date** ကိုသုံးပြီး ဝန်ထမ်းတစ်ဦးချင်းစီရဲ့ အလုပ်သက်တမ်းကို တွက်နိုင်ပါတယ်။ ဒီလို insights တွေက HR team က အရေးကြီးဆုံး ဆုံးဖြတ်ချက်တွေ ချတဲ့အခါမှာ တကယ်အသုံးဝင်ပါတယ်။

2. SQL in Web Development

2.1 Connecting Databases to Web Apps

Web apps တွေကနေ SQL databases နဲ့ ချိတ်ဆက်ပြီး data ရှာဖွေ၊ ပြင်ဖို့၊ ဖျက်ဖို့ စတဲ့ လုပ်ဆောင်ချက် တွေကို လုပ်လို့ရတဲ့အတွက် UI (User Interface) ကနေတစ်ဆင့် real-time data တွေကို တိုက်ရိုက်မြင်နိုင် သလို ကြိုက်သလို ပြန်ပြင်နိုင်ဖို့လည်း အဆင်ပြေသွားတာပါ။

```
[exercises_db=# SELECT * FROM transactions;
 transaction_id | order_id | payment_method | amount | transaction_date
-----+-----+-----+-----+-----
              1 |         1 | Cash           | 25000.00 | 2025-05-01
              2 |         2 | Credit Card    | 15000.00 | 2025-05-02
              3 |         3 | Mobile Pay     | 15000.00 | 2025-05-03
              4 |         4 | Cash           | 8000.00  | 2025-05-04
(4 rows)
```

Practical Applications

Data Analysis

Payment method အလိုက် total amount ကို တွက်ချင်တယ်ဆိုရင် backend ပိုင်းကနေ အခုလို ရေးပေးလိုက်လို့ ရပါတယ်။

```
SELECT payment_method, SUM(amount) FROM transactions GROUP BY
payment_method;
```

Cash နဲ့ ပေးတာ ဘယ်လောက်လဲ၊ Credit Card နဲ့ ဝယ်သွားတာက ဘယ်လောက်လဲဆိုတာ ခွဲသိနိုင်ပါတယ်။

Web Development

Customer dashboard တစ်ခုမှာ ငွေပေးချေမှုမှတ်တမ်း (payment history) ကို ပြချင်ရင် **transactions** table ထဲက data တွေကို database နဲ့ ချိတ်ဆက်ပြီး query ဆွဲယူ၊ website ပေါ်မှာ display လုပ်နိုင်ပါတယ်။ အရိုးရှင်းဆုံး ပြောလိုက်တာပါ။

Business Intelligence

Business ဘက်ကကြည့်မယ်ဆိုရင် transaction date ကို သုံးပြီး payment delays ဖြစ်တာတွေကို တိတိကျကျ ရှာဖွေနိုင်ပါမယ်။ ဒီလို analytics တွေက financial ဆုံးဖြတ်ချက်တွေ ချရာမှာ အထောက်အကူ ဖြစ်ပါတယ်။

2.2 ORM vs. Raw SQL

ORM (Object-Relational Mapping)

ORM လို့ပြောရင် SQL query တွေကို ကိုယ်တိုင်မရေးဘဲ database နှင့် objects (models) ကို ချိတ်ဆက်ပေးနိုင်တဲ့ tool တစ်ခုပဲ ဖြစ်ပါတယ်။

ဥပမာ - Sequelize, Prisma (JavaScript/TypeScript), SQLAlchemy (Python), Hibernate (Java), Entity Framework (C#) စသဖြင့် programming languages တွေအလိုက် အမျိုးမျိုးရှိပါတယ်။

ORM ရဲ့ အားသာချက်တွေက ရိုးရှင်းလွယ်ကူစွာ အသုံးပြုနိုင်ပြီး development process ကိုလည်း မြန်မြန် အပြီးသတ်နိုင်ပါတယ်။ ဒါ့အပြင် code readability ကို မြှင့်တင်ပေးလို့ အခြား developer တွေအတွက် လည်း ဖတ်တဲ့အခါ နားလည်ရလွယ်ကူစေပါတယ်။

ဒါပေမယ့် complex queries တွေမှာ performance ပိုလိုအပ်ပြီး customization အတွက် ထိန်းချုပ်မှု ကန့်သတ်ချက်တွေ ရှိနိုင်တာကို ဂရုစိုက်ရပါမယ်။ ဒီလိုအားနည်းချက်တွေရှိပေမယ့် မိမိ project အတွက် ကောင်းမွန်တဲ့ နည်းပညာတစ်ခုဖြစ်တဲ့ **ORM** ကို အသုံးပြုသင့်ပါတယ်။

Raw SQL

Raw SQL ဆိုတာကတော့ SQL queries ကို တိုက်ရိုက်ရေးသားတာကို ဆိုလိုပါတယ်။ Database ထဲမှာ ဘာကိုရှာဖွေချင်တာလဲ၊ ဘာကိုပြောင်းလဲချင်တာလဲဆိုတာကို ကိုယ်တိုင် instructions ပေးရတာပါ။

Raw SQL ရဲ့ အားသာချက်တွေက performance ပိုကောင်းပြီး query logic ပေါ်မှာ မိမိလိုချင်သလောက် ထိန်းချုပ်နိုင်တာပါ။ ဒါ့အပြင် complex joins၊ subqueries စတာတွေကို တိတိကျကျ ရေးနိုင်ပါတယ်။

အားနည်းချက်တွေကတော့ code ရေးသားရတာ အချိန်ပိုကြာနိုင်ပြီး error handling နဲ့ security (ဥပမာ - SQL Injection) တွေကို ကိုယ်တိုင်စောင့်ကြည့်ရမှာ ဖြစ်ပါတယ်။

// Raw SQL

```
UPDATE transactions
SET amount = 20000.00
WHERE transaction_id = 2;
```

// Prisma ORM

```
await prisma.transactions.update({
  where: { transaction_id: 2 },
  data: { amount: 20000.00 },
});
```

ORM ဟာ ရှိရှိလွယ်ကူတဲ့ queries တွေအတွက် သင့်တော်ပေမယ့် ရှုပ်ထွေးတဲ့ queries တွေအတွက် တော့ **Raw SQL** ကို အသုံးပြုတာကမှ data ကို ပိုမိုထိရောက်စွာ စိတ်ကြိုက် စီမံခန့်ခွဲနိုင်မှာပါ။

3. SQL for Business Intelligence

3.1 Dashboards and Metrics

SQL ဟာ စီးပွားရေးဆိုင်ရာ dashboards တွေမှာလည်း အရေးကြီးတဲ့ metrics တွေကိုရယူဖို့ အထောက်အကူပြုပါတယ်။ ဥပမာ - sales growth, customer activities လို မိမိလုပ်ငန်းနဲ့ သက်ဆိုင်တဲ့ data တွေကို လွယ်လင့်တကူ analyze လုပ်နိုင်ပါတယ်။ Dashboards တွေက strategic data တွေကို clear and visual way နဲ့ ပြသပေးတာကြောင့် business decision-making မှာ အကောင်းဆုံးအကူအညီ ပေးပါတယ်။


```

(exercises_db=# SELECT * FROM reviews;
 review_id | product_id | customer_id | rating | comment | review_date
-----+-----+-----+-----+-----+-----
          1 |          1 |          1 |       5 | Great quality rice! | 2025-05-02
          2 |          2 |          1 |       4 | Good oil, but small bottle | 2025-05-03
          3 |          1 |          2 |       3 | Average quality | 2025-05-04
          4 |          3 |          3 |       5 | Love this soap! | 2025-05-05
(4 rows)

```

Practical Applications

Data Analysis

Product အလိုက် ရရှိတဲ့ ပျမ်းမျှ rating ကို တွက်ချင်ရင် –

```
SELECT product_id, AVG(rating) FROM reviews GROUP BY product_id;
```

အခုလိုမျိုး query ကို အသုံးပြုပြီး customer satisfaction ကို ခန့်မှန်းနိုင်ပါတယ်။ ဒီလို insights တွေကတော့ နောက်ဆောင်မှာ product quality ပိုကောင်းမွန်လာစေဖို့ တကယ်အထောက်အကူဖြစ်ပါတယ်။

Web Development

E-commerce site တစ်ခုအတွက် product page တစ်ခုထဲမှာ ဖောက်သည်တွေရဲ့ သုံးသပ်ချက်တွေ (reviews) ကို database မှ data တွေကို fetch လုပ်ပြီး ပြသနိုင်ပါတယ်။ ဒါက အသုံးပြုသူတွေရဲ့ ယုံကြည်မှုရရှိဖို့နဲ့ user experience ကို ပိုမိုကောင်းမွန်စေဖို့ အထောက်အကူဖြစ်ပါတယ်။

Business Intelligence

Review data တွေကို သေချာ analyze လုပ်ပြီး ဖောက်သည်တွေရဲ့ စိတ်ကျေနပ်မှု (customer satisfaction level) ကို တိုင်းတာနိုင်ပါတယ်။ အဲ့ဒီ data တွေကတော့ market strategy ပြင်ဆင်ရာမှာ အထောက်အကူဖြစ်ပြီး ထုတ်ကုန်အရည်အသွေး မြှင့်တင်ရာမှာလည်း အကျိုးဝင်မှာဖြစ်ပါတယ်။

3.2 Working with Large Datasets

Data ပမာဏမြောက်များစွာနဲ့ အလုပ်လုပ်တဲ့အခါမှာ query performance ကို အထူးဂရုစိုက်ဖို့ လိုပါတယ်။ ရှေ့က chapter တွေမှာလည်း ဖော်ပြခဲ့ပြီး ဖြစ်ပါတယ်။ ဒါပေမယ့် ထပ်ဖော်ပြတာပါ။

- **Indexes:** Data ကို မြန်မြန်ဆွဲထုတ်ရယူနိုင်ဖို့ index တွေကို အသုံးပြုပါ။

- **Partitioning:** Data ကို logical partitions အနေနဲ့ ခွဲခြားထားခြင်းက query speed ကို တိုးလာစေပါတယ်။ ဥပမာ - E-commerce site မှာ orders ကို လအလိုက် partition လုပ်ထားရင် လတ်တလော order တွေကို မြန်မြန် ရှာဖွေနိုင်ပါတယ်။
- **WHERE Clauses:** မလိုအပ်တဲ့ rows တွေကို စစ်ထုတ်ဖို့တော့ WHERE condition ကို အသုံးပြုသင့်ပါတယ်။

Partitioning ဆိုတာက data တွေကို logical အရ သူကဏ္ဍအလိုက် အပိုင်းပိုင်းခွဲခြားထားတာ ဖြစ်ပါတယ်။ Table တစ်ခုမှာ millions of rows ရှိနေတဲ့အခါ table တစ်ခုလုံးကို scan လုပ်ရမယ်ဆိုရင် query time က အတော်ကြာနိုင်ပါတယ်။ ဥပမာ - 2025 ခုနှစ်ရဲ့ reviews တွေကိုသာ ရှာချင်တယ်ဆိုပါစို့။ အဲဒီအခါမှာ row အားလုံးကို စစ်ဖို့မလိုဘဲ 2025 အတွက်သီးသန့်ခွဲထားတဲ့ partition ကိုပဲ query လုပ်လို့ရတာမို့ ပိုမြန်ဆန်တဲ့ performance ရနိုင်ပါတယ်။

```
-- Child partition: 2023
CREATE TABLE reviews_2023 PARTITION OF reviews
    FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');

-- Child partition: 2024
CREATE TABLE reviews_2024 PARTITION OF reviews
    FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');

-- Child partition: 2025
CREATE TABLE reviews_2025 PARTITION OF reviews
    FOR VALUES FROM ('2025-01-01') TO ('2026-01-01');
```

ဒါကြောင့် query ကို run လုပ်တဲ့အချိန်မှာ မလိုအပ်တဲ့ data အားလုံးကို scan မလုပ်ပဲ လိုအပ်တဲ့ partition portion တစ်ခုကိုပဲ ရွေးပြီး မြန်မြန်ဆန်ဆန် လုပ်ဆောင်နိုင်ပါတယ်။ အဲဒီလိုနဲ့ query speed တိုးလာပြီး database performance လည်း ပိုမိုကောင်းမွန်လာပါတယ်။

4. Case Studies

4.1 Retail Sales Analysis

လက်လီဆိုင်တစ်ခုမှာ ရောင်းအားနဲ့ပတ်သက်တဲ့ data တွေကို SQL နဲ့ analyze လုပ်ပြီး ရောင်းအားတက်မတက်၊ ထိရောက်မှုရှိမရှိ၊ ဖောက်သည်အခြေအနေစတာတွေကို နားလည်နိုင်ပါတယ်။

// Source Code Link:

<https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-13/11-sales-table.sql>

```

(exercises_db=# SELECT * FROM sales;
 sale_id | sale_date | product_id | quantity | unit_price | customer_id
-----+-----+-----+-----+-----+-----
      1 | 2025-06-10 |          1 |         3 |      10.00 |          1
      2 | 2025-06-10 |          2 |         1 |      25.50 |          2
      3 | 2025-06-11 |          1 |         2 |      10.00 |          3
      4 | 2025-06-11 |          3 |         5 |       7.20 |          1
      5 | 2025-06-12 |          2 |         4 |      25.00 |          2
(5 rows)

```

ဒီ SQL query က စတိုးဆိုင်ရဲ့ နေ့စဉ်ရောင်းအားအတက်အကျကို သိနိုင်ဖို့ ရေးထားတာပါ။

```

SELECT
    sale_date,
    SUM(quantity * unit_price) AS daily_sales,
    AVG(unit_price) AS average_price
FROM sales
GROUP BY sale_date
ORDER BY sale_date;

```

4.2 User Activity Tracking

User Activity Tracking ဆိုတာက web app ကိုသုံးတဲ့ user တွေ ဘယ်လိုလုပ်ဆောင်ချက်တွေ (actions) လုပ်ခဲ့တယ်ဆိုတာကို စုစည်းထားပြီး နောက်ဆုံးမှာ လေ့လာသုံးသပ်တာပါ။ ဥပမာ - user တစ်ယောက်ရဲ့ login/logout မှတ်တမ်းတွေ၊ product တွေကြည့်ခဲ့တဲ့ history၊ ဝယ်ယူခဲ့တဲ့ history စတာတွေကို သိမ်းဆည်းထားတာပါ။ အောက်မှာ **user_activities** ဆိုတဲ့ table နမူနာတစ်ခုရှိပြီး သုံးစွဲသူတွေရဲ့ လုပ်ဆောင်ချက်တွေကို log လုပ်ထားတဲ့ ပုံစံဖြစ်ပါတယ်။

// Source Code Link:

<https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-13/12-user-activities.sql>

```

(exercises_db=# SELECT * FROM user_activities;
 activity_id | customer_id | activity_type | activity_time | description
-----+-----+-----+-----+-----
      1 |          1 | login        | 2025-06-18 16:22:08.139287 | User logged in
      2 |          1 | view_product | 2025-06-18 16:22:08.139287 | Viewed product ID 5
      3 |          2 | purchase     | 2025-06-18 16:22:08.139287 | Purchased product ID 3
      4 |          1 | logout       | 2025-06-18 16:22:08.139287 | User logged out
(4 rows)

```

ဒီ query မှာတော့ တစ်ဦးချင်းသုံးစွဲသူအလိုက် activity တွေ ဘယ်လောက်ရှိတယ်ဆိုတာကို track လုပ်ထားတာပါ။

```
SELECT
customer_id,
COUNT(*) AS total_activities
FROM user_activities
GROUP BY customer_id
ORDER BY customer_id;
```

5. Exercises

// Source Code Link:

<https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-13/13-customers.sql>

```
[exercises_db=# SELECT * FROM customers;
 customer_id | customer_name | membership_level
-----+-----+-----
          1 | John Doe      | Silver
          2 | Jane Smith    | Gold
          3 | Alice Johnson | Silver
(3 rows)
```

// Source Code Link:

<https://github.com/rubenhtun/dive-into-sql/blob/main/chapter-13/14-orders.sql>

```
[exercises_db=# SELECT * FROM orders;
 order_id | customer_id | product_category | order_amount | order_date
-----+-----+-----+-----+-----
          1 |          1 | Electronics      |         250.00 | 2025-06-01
          2 |          1 | Books            |          35.00 | 2025-06-01
          3 |          2 | Electronics      |          40.00 | 2025-06-02
          4 |          3 | Clothing         |          75.00 | 2025-06-01
(4 rows)
```

Question 1:

Write an SQL query to find the **average order amount** for each product category from the **orders** table and create a **report** for the business dashboard.

Question 2:

Write an SQL query to find the **total order amount** and the **number of orders** placed by each customer from the **customers** and **orders** tables.

Question 3:

Write an SQL query to find customers who placed orders on **2025-06-01** from the **orders** table (to be displayed in a web app).

Question 4:

Rewrite the query to find customers whose orders exceed **\$100** as a **JOIN** query **between** the **customers** and **orders** tables.

Question 5:

Write a SQL query to calculate the **average order amount** grouped by **membership level** (e.g., Silver, Gold) using the **customers** and **orders** tables for inclusion in a business intelligence dashboard.

Chapter 14: Advanced Topics

Advanced topics လို့ ခေါင်းစဉ်ရွေးချယ်လိုက်တာက database တွေကို ပိုမိုထိရောက်စွာ စီမံခန့်ခွဲနိုင်ဖို့နဲ့ ရှုပ်ထွေးတဲ့ operations တွေကို လွယ်ကူစွာ လုပ်ဆောင်နိုင်ဖို့ သိသင့်တဲ့အရာတွေမို့ပါ။ ဒီအခန်းမှာ stored procedures နဲ့ functions, triggers, JSON နဲ့ XML data types တွေကို ဘယ်လိုသုံးရမလဲ၊ ပြီးတော့ရော NoSQL နဲ့ SQL ကြားက မတူညီတဲ့အချက်တွေကိုပါ သေချာလေ့လာသွားမှာ ဖြစ်ပါတယ်။

1. Stored Procedures and Functions

1.1 Writing and Calling Procedures

Stored procedures ဆိုတာကတော့ SQL code တချို့ကို database ထဲမှာ သိမ်းဆည်းထားနိုင်ပြီး နောက်ပိုင်းမှာ မကြာခဏ ပြန်လည်အသုံးပြုလို့ရအောင် စနစ်တကျရေးထားတာပါ။ Function တစ်ခုရေးတဲ့ programming concept နဲ့ဆင်တူပါတယ်။ တစ်ခါရေးပြီးသား code ကို တခြားနေရာတွေမှာလည်း ထပ်ခါတလဲလဲ ခေါ်သုံးနိုင်တာမျိုးပါ။

Stored procedure တစ်ခုရေးဖို့ **CREATE PROCEDURE** ဆိုတဲ့ command ကို အသုံးပြုရပြီး အကယ်၍ parameter တွေ လိုအပ်တယ်ဆိုရင်လည်း တစ်ခါတည်း ထည့်သွင်းဖော်ပြနိုင်ပါတယ်။ Procedure ကို သိမ်းဆည်းပြီးသွားတဲ့နောက်မှာတော့ **CALL** ဆိုတဲ့ command နဲ့ ကြိုက်တဲ့အချိန် ခေါ်သုံးနိုင်ပါတယ်။

Stored procedures အသုံးပြုခြင်းရဲ့ အားသာချက်တချို့ကို ဖော်ပြလိုက်ပါတယ်။

- မကြာခဏ အသုံးပြုဖို့လိုအပ်တဲ့ SQL လုပ်ဆောင်ချက်တွေအတွက် အရမ်းအဆင်ပြေပါတယ်။
- အခြားအသုံးပြုသူတွေက underlying tables ကို တိုက်ရိုက်မတွေ့ရဘဲ procedure ကိုသာ ခေါ်သုံးနိုင်တာမို့ security အတွက် ကောင်းပါတယ်။
- Logic တစ်ခုကို တစ်နေရာတည်းမှာ သိမ်းထားပေးတာကြောင့် ထိန်းသိမ်းရလွယ်ကူပါတယ်။
- Procedure တွေဟာ database ထဲမှာ ချက်ချင်း compile လုပ်ထားတဲ့အတွက် ထပ်ပြီးခေါ်တဲ့အခါ query တွေဟာ ပိုမြန်ဆန်စွာ အလုပ်လုပ်ပါတယ်။

ဥပမာ - ဖောက်သည်တစ်ဦးအတွက် အော်ဒါစုစုပေါင်းတန်ဖိုးကို တွက်ချက်ဖို့ stored procedure တစ်ခု ရေးမယ်ဆိုပါစို့။

```

CREATE PROCEDURE GetCustomerTotalOrderAmount(
    IN cust_id INT,
    OUT total_amount NUMERIC
)
LANGUAGE plpgsql
AS $$
BEGIN
    SELECT SUM(o.amount) INTO total_amount
    FROM orders o
    WHERE o.customer_id = cust_id;
END;
$$;

CALL GetCustomerTotalOrderAmount(1, NULL);

```

ရေးထုံးတွေကတော့ SQL အမျိုးမျိုးပေါ်မူတည်ပြီး ကွဲပြားနိုင်ပါတယ်။ ဒီနေရာမှာတော့ Postgres SQL ကို သုံးထားပါတယ်။ အပေါ်ကရေးထားတာတွေကို နားလည်သွားအောင် ပြန်ရှင်းပြပါမယ်။

CREATE PROCEDURE GetCustomerTotalOrderAmount(...) ဆိုတာက PostgreSQL မှာ stored procedure တစ်ခုကို ဖန်တီးတာဖြစ်ပါတယ်။ **IN cust_id INT** ဆိုတာက procedure ကို ခေါ်သုံးတဲ့အချိန်မှာ ပြောပြပေးရမယ့် input parameter ပါ။ ဒီနေရာမှာ customer ရဲ့ ID (**cust_id**) ကို **INT (integer)** အဖြစ် ရယူပါတယ်။

OUT total_amount NUMERIC ဆိုတာက ဒီ procedure ကနေ ရလာမယ့် output parameter တစ်ခု ဖြစ်ပြီး customer ရဲ့ စုစုပေါင်း order တန်ဖိုးကို **NUMERIC** အဖြစ် ပြန်ပေးမှာဖြစ်ပါတယ်။ **LANGUAGE plpgsql** ဆိုတာက PostgreSQL ရဲ့ procedural language ကို သတ်မှတ်တာဖြစ်ပြီး ဒီနေရာမှာ **PL/pgSQL** ကိုသုံးပြီး procedure ကိုရေးထားတာ ဖြစ်ပါတယ်။

AS \$\$... \$\$; ဆိုတာက procedure ရဲ့ body အပိုင်းကို dollar-quote နဲ့ wrap လုပ်ထားတာဖြစ်ပြီး **BEGIN ... END;** ကြားမှာတော့ procedure ရဲ့ လုပ်ဆောင်မယ့် SQL logic ကို ရေးသားပါတယ်။

ရလာတဲ့ **SUM(o.amount)** ကို **INTO total_amount** ထဲ ထည့်လိုက်တာကို သတိပြုပါ။ ပြီးမှ အဲ့ procedure ကို ပြန်ခေါ်တဲ့အခါ **()** ထဲက **1** ဆိုတာက **cust_id** အနေနဲ့ customer ID 1 ကို ပေးတာ ဖြစ်ပါတယ်။ PostgreSQL မှာ **CALL** statement သုံးတဲ့အခါ **OUT** parameter တွေအတွက် value မပေးချင်ရင် **NULL** လို့ placeholder အနေနဲ့ ပေးဖို့လိုပါတယ်။ Procedure ရဲ့ parameter အားလုံး (**IN, OUT, INOUT**) ကို တိတိကျကျ ထည့်ပေးဖို့လိုပါတယ်။ **OUT** parameter ဖြစ်တယ်ဆိုပြီး မဖြည့်ဘဲချန်ထား လို့ မရပါ။

ဒါကတော့ `customer_id 3` ကို ခေါ်သုံးလိုက်တဲ့အခါ ရရှိလာတဲ့ output ပဲ ဖြစ်ပါတယ်။

```
[exercises_db=# CALL GetCustomerTotalOrderAmount(3, NULL);
total_amount
-----
          8000
(1 row)
```

1.2 Reusability and Security

- ဒါကြောင့် code ထပ်နေတာတွေကို လျော့ချပေးပြီး database operations ကို တသမတ်တည်းနဲ့ စနစ်တကျ ဆောင်ရွက်နိုင်စေပါတယ်။
- Security အတွက် ကောင်းတယ်ဆိုတာက အသုံးပြုသူတွေဟာ procedure ကိုသာ ခေါ်သုံးခွင့်ရှိပြီး root tables တွေကို တိုက်ရိုက် access မလုပ်နိုင်ပါဘူး။ ဒါကြောင့် data တွေဟာ security အရ အားကောင်းသွားပြီး မလိုလားဘဲ ပြန်ပြင်တာတို့၊ ဖျက်တာတို့ကနေ ကာကွယ်နိုင်တာပါ။

ဥပမာ - Stored procedure အသုံးပြုပြီး order အသစ်တစ်ခု ပြန်ပြောင်းထည့်ကြည့်ကြပါမယ်။

```
CREATE OR REPLACE PROCEDURE AddOrder(
    odr_id INT,
    cust_id INT,
    prod_name VARCHAR,
    amt INT,
    ord_date DATE
)
LANGUAGE plpgsql
AS $$
BEGIN
    INSERT INTO orders (order_id, customer_id, product, amount,
order_date)
    VALUES (odr_id, cust_id, prod_name, amt, ord_date);
END;
$$;

CALL AddOrder(7, 3, 'coffee', 10000, '2025-06-18');
```

Procedure နဲ့ function တွေကို နာမည်ပေးတဲ့အခါ PascalCase (ရှေ့စာလုံးအကြီး) ရေးလည်း ရပါတယ်။

တကယ်လို့ **AddOrder** ဆိုတဲ့ procedure (သို့) function ဟာ ရှိပြီးသားဆိုရင် ပြန်ပြင်ချင်တဲ့အခါ **OR REPLACE** command ကို အသုံးပြုနိုင်ပါတယ်။ လုံးဝအသစ် ဖန်တီးမယ်ဆိုရင်တော့ **OR REPLACE** မထည့်ဘဲ **CREATE** နဲ့ ရေးလည်းရပါတယ်။

အရင်က **orders** table မှာ auto increment လုပ်ပေးတဲ့ serial key ကို မသုံးထားတဲ့အတွက် procedure ကို ပြန်ခေါ်တဲ့အခါ order id ကို ရှေ့ဆုံးမှာထားပြီး တစ်ခါတည်း ထည့်ခေါ်လိုက်တာပါ။

orders table ကို ပြန်စစ်ကြည့်တဲ့အခါ နောက်ထပ် row အသစ်တစ်ခု တိုးလာတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။

2. Triggers and Events

2.1 Automating Database Actions

Triggers ဆိုတာကတော့ database အတွင်းမှာ အပြောင်းလဲတစ်ခု (ဥပမာ - data ထည့်တာ၊ ဖျက်တာ၊ ပြင်တာ) ဖြစ်ပေါ်တဲ့အခါ အလိုအလျောက် လုပ်ဆောင်တဲ့ mechanism တစ်ခုပဲ ဖြစ်ပါတယ်။

ဥပမာ - **orders** table ထဲမှာ order အသစ်တစ်ခု ထည့်တဲ့အခါ trigger ကို အသုံးပြုပြီး အဲ့ဒီ order ရဲ့ **order_id** နဲ့ **amount** ကို ယူကာ **order_history** table ထဲကို **recorded_at** အချိန်အပါအဝင် အလိုအလျောက် မှတ်တမ်းတင်ပေးနိုင်ပါတယ်။

```
CREATE TABLE order_history (
    history_id SERIAL PRIMARY KEY,
    order_id INT NOT NULL,
    total_amount INT NOT NULL,
    recorded_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

SERIAL PRIMARY KEY လို့ ရေးထားတဲ့အတွက် **history_id** ကို PostgreSQL က auto increment လုပ်ပေးမှာ ဖြစ်ပါတယ်။ **recorded_at** column ကိုလည်း **CURRENT_TIMESTAMP** အဖြစ် default လုပ်ထားတဲ့အတွက် order အသစ်တစ်ခုစီ ထည့်လိုက်တိုင်းမှာ အချိန်နှင့်တပြေးညီ history မှတ်တမ်းကို အလိုအလျောက် စနစ်တကျ မှတ်ပေးသွားမှာ ဖြစ်ပါတယ်။

```

-- 1. Create the trigger function
CREATE OR REPLACE FUNCTION after_order_insert_fn()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO order_history (order_id, total_amount, recorded_at)
    VALUES (NEW.order_id, NEW.amount, CURRENT_TIMESTAMP);
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

-- 2. Create the trigger
CREATE TRIGGER after_order_insert
AFTER INSERT ON orders
FOR EACH ROW
EXECUTE FUNCTION after_order_insert_fn();

-- 3. Test it with an insert
INSERT INTO orders (order_id, customer_id, product, amount, order_date)
VALUES (8, 8, 'facial cleanser', 10000, '2025-06-19');

```

ဒီလိုပါ။ စတင်မှာ order record အသစ်ကို **orders** table ထဲက row တစ်ခုချင်းစီမှာ ထည့်ပါတယ်။ ထည့်ပြီးချိန်မှာတော့ နောက်ထပ် trigger အနေနဲ့ **after_order_insert_fn()** function ကို အလိုအလျောက် ဆောင်ရွက်ပေးပြီး new datasets အနေနဲ့ **order_history** table ထဲမှာ record အသစ်တွေကို automatic စနစ်တကျ သိမ်းဆည်းပေးသွားတာပါ။

Events ဆိုတာ ရှိပါသေးတယ်။ သူကျတော့ သတ်မှတ်ထားတဲ့ အချိန်ဇယား (schedule) အတိုင်း အလိုအလျောက် လုပ်ဆောင်ပေးတဲ့ database task တွေကို ပြောတာပါ။ ဥပမာ - တစ်နေ့တစ်ကြိမ် backup လုပ်တာ၊ expired data တွေကို cleanup လုပ်တာစသဖြင့် automation အလုပ်တွေမှာ အသုံးဝင်ပါတယ်။

ဥပမာ - နေ့စဉ် အော်ဒါစုစုပေါင်း (total orders) ကို အလိုအလျောက် တွက်ချက်ပြီး **order_summary** table ထဲကို ထည့်ပေးတဲ့ event တစ်ခုကို အခုလို ဖန်တီးနိုင်ပါတယ်။

```

CREATE FUNCTION daily_order_summary()
RETURNS void AS $$
BEGIN
    INSERT INTO order_summary (summary_date, total_amount)
    SELECT CURRENT_DATE, SUM(amount)
    FROM orders

```

```

    WHERE order_date = CURRENT_DATE;
END;
$$ LANGUAGE plpgsql;

SELECT daily_order_summary();

SELECT * FROM order_summary;

```

ခြင်းချက်အနေနဲ့ PostgreSQL မှာ MySQL လို **CREATE EVENT** နဲ့ event တစ်ခုကို တိုက်ရိုက်ဖန်တီးယူလို့ မရပါဘူး။ ဒါကြောင့် PostgreSQL မှာ event တစ်ခုကို သတ်မှတ်ထားတဲ့ အချိန်ရောက်တာနဲ့ သူ့အလိုလို လုပ်ဆောင် သွားဖို့ဆိုရင် external tools တွေကို အသုံးပြုဖို့ လိုပါတယ်။

1. cron job (Linux/macOS)

- Linux, Ubuntu, macOS စနစ်တွေမှာ သုံးတတ်တဲ့ task scheduler ပါ။
- cron job ဆိုတာကတော့ OS ကနေ PostgreSQL query (function or statement) တစ်ခုကို တိတိကျကျ သတ်မှတ်ထားတဲ့အချိန်မှာ automatic run ပေးတဲ့ system command ပါ။

2. Task Scheduler (Windows)

- Windows အသုံးပြုသူတွေအတွက် built-in scheduler tool ဖြစ်ပါတယ်။
- PostgreSQL function ကို သတ်မှတ်ထားတဲ့အချိန်မှာ execute လုပ်နိုင်ပါတယ်။

3. pgAgent (pgAdmin scheduler tool)

- pgAdmin GUI မှာ သုံးနိုင်တဲ့ PostgreSQL job scheduling tool ဖြစ်ပါတယ်။
- Scheduled jobs တွေကို visual interface နဲ့ပဲ set လုပ်လို့ရပါတယ်။

ဥပမာ - Linux မှာ cron job ကို အသုံးပြုမယ် ဆိုပါစို့။

```
0 0 * * * psql -d lucky_store_db -c "SELECT daily_order_summary();"
```

ဒီ command ဟာ နေ့တိုင်း ည 12 နာရီမှာ **lucky_store_db** ထဲက **orders** table မှ အော်ဒါစုစုပေါင်း ကို **order_summary** ထဲကို အလိုအလျောက်ထည့်ပေးသွားမှာ ဖြစ်ပါတယ်။ ဒီနေရာမှာ ဒီလောက်ပဲ ဖော်ပြ လိုက်ပါမယ်။

2.2 Use Cases for Triggers

ဒါကတော့ triggers တွေကို ဘယ်လိုအခြေအနေတွေမှာ အသုံးပြုကြလဲ၊ ဘာတွေအတွက် အသုံးဝင်တယ် ဆိုတာကို အကျဉ်းချုံးလေး ဖော်ပြတာဖြစ်ပါတယ်။

- **Data Integrity:** Order တစ်ခုဖျက်တဲ့အခါ ဆက်စပ် log တွေကိုလည်း အလိုအလျောက် ဖျက်ပေးနိုင်ရန်။
- **Logging:** Data ပြောင်းလဲမှုတွေကို စောင့်ကြည့်ပြီး မှတ်တမ်းတင်ထားနိုင်ရန်။
- **Automatic Calculation:** စုစုပေါင်းတန်ဖိုးတွေကို အလိုအလျောက်တွက်ချက်ဖို့လိုတဲ့အခါ။

3. Working with JSON and XML

3.1 Storing and Querying JSON

အခုခေတ်မှာ သုံးတဲ့ modern database အများစုက JSON data ကို တိုက်ရိုက်သိမ်းဆည်းနိုင်သလို အဲဒီ JSON ထဲက information တွေကိုလည်း တိုက်ရိုက် query လုပ်နိုင်ပါတယ်။ ဒါကြောင့် web app တွေနဲ့ API တွေမှာ အသုံးများတဲ့ JSON format ကို database ထဲမှာ အဆင်ပြေပြေ ထိန်းသိမ်းနိုင်ပါတယ်။ ဒီနေရာမှာတော့ JSON data type အကြောင်းကို အသေးစိတ် မရှင်းပြနိုင်တော့ပါဘူး။

ဥပမာ - **customers** table မှာ JSON data type ကိုပဲ လက်ခံတဲ့ preferences column အသစ်တစ်ခု ထည့်ကြည့်ကြရအောင်ပါ။

```
ALTER TABLE customers ADD preferences JSON;

UPDATE customers
SET preferences = '{"language": "English", "theme": "Dark"}'
WHERE customer_id = 1;
```

ဒီလိုနဲ့ JSON format ကို column တစ်ခုအနေနဲ့ သိမ်းပြီး customer တစ်ဦးချင်းစီအတွက် စိတ်ကြိုက် settings တွေကို ထည့်သွင်းနိုင်ပါတယ်။ Web app တွေမှာ user preferences တွေသိမ်းတဲ့ real-world use case တစ်ခုနဲ့တောင် တူပါတယ်။

```
SELECT customer_name, preferences->'language' AS pref_language
FROM Customers
```

```
WHERE preferences→'theme' = 'Dark';
```

preferences ဆိုတဲ့ JSON column ထဲက **"language"** ဆိုတဲ့ key ရဲ့ string value ကို ထုတ်ယူတာပါ။ **"theme"** key ရဲ့ value က **'Dark'** ဖြစ်တဲ့ row တွေကိုသာ filter လုပ်တာပါ။

ဒီနေရာမှာ → ဟာ text extraction operator ပါ။ သူက JSON data ထဲက key ရဲ့ value ကို string (text) အဖြစ် ထုတ်ယူတဲ့ operator ဖြစ်ပါတယ်။

3.2 Parsing XML Data

နောက်ပိုင်း database systems တွေမှာ XML data type အတွက် ပိုမိုထိရောက်ပြီး အသုံးပြုရလွယ်ကူတဲ့ feature တွေ၊ function တွေ ထည့်သွင်းပေးထားတာ ဖြစ်ပါတယ်။ ဒါကြောင့် XML data ကို သိမ်းဆည်းဖို့၊ ရှာဖွေဖို့အစ ပြန်ပြင်ဆင်နိုင်ဖို့အလယ် manipulate လုပ်ဖို့အဆုံး ပိုအဆင်ပြေပါတယ်။

ဥပမာ - **customers** table မှာ XML data type ကိုပဲ လက်ခံတဲ့ column အသစ်တစ်ခု ထပ်ထည့်ကြည့်ရအောင်ပါ။

```
ALTER TABLE customers ADD COLUMN profile xml;

UPDATE customers
SET profile =
'<profile><city>Yangon</city><country>Myanmar</country></profile>'
WHERE customer_id = 4;
```

ကိုယ်တိုင်ပဲ စစ်ကြည့်ပါ။ **profile** ဆိုတဲ့ column အသစ်တိုးလာပြီး **customer_id** 4 ရှိတဲ့ row ထဲမှာ XML data တွေ ရောက်နေမှာပဲ ဖြစ်ပါတယ်။

```
SELECT customer_name,
       (xpath('//city/text()', profile))[1]::text AS city
FROM customers
WHERE (xpath('//country/text()', profile))[1]::text = 'Myanmar';
```

PostgreSQL မှာ XML data တွေကို ရယူဖို့ **xpath()** function ကို သုံးပါတယ်။ ဥပမာ - **xpath('//city/text()', profile)** ဆိုရင် XML ထဲက **city** tag ရဲ့ အတွင်းစာသားတွေကို array အနေနဲ့ ပြန်ပေးပါတယ်။ **[1]** ဆိုတာက ရလာတဲ့ array ထဲက ပထမဆုံး element ကို ရွေးယူတာ

ဖြစ်ပါတယ်။ **::text** ကတော့ XML node ကို string (text) data type အဖြစ် ပြောင်းပေးတာ ဖြစ်ပါတယ်။

WHERE condition မှာလည်း ဒီနည်းပေါ်မူတည်ပြီး **profile** XML ထဲက **country** tag ရဲ့ value ကို ရှာဖွေပြီး အဲဒီ **country** က **'Myanmar'** ဖြစ်တဲ့ row တွေကို filter လုပ်တာ ဖြစ်ပါတယ်။

ရလဒ်က အခုလိုတွေ့ရမှာပါ။

```
exercises_db=# SELECT customer_name,
exercises_db=#           (xpath('//city/text()', profile))[1]::text AS city
exercises_db=# FROM customers
exercises_db=# WHERE (xpath('//country/text()', profile))[1]::text = 'Myanmar';
 customer_name | city
-----+-----
 John Doe      | Yangon
(1 row)
```

4. Introduction to NoSQL vs. SQL

4.1 Key Differences

ဒီအပိုင်းမှာ NoSQL နဲ့ SQL ကြားမှာ မတူညီတာလေးတွေကို ပြန်မှတ်လို့ရအောင် ဖော်ပြပေးသွားပါမယ်။

SQL (Relational Databases):

- SQL database တွေကတော့ stable ဖြစ်ပြီး data တွေကို စနစ်တကျ သိမ်းဆည်းဖို့အတွက် အသုံးပြုပါတယ်။
- Data ကို table ပုံစံနဲ့ သိမ်းပါတယ်။ column တွေက field ဖြစ်ပြီး row တစ်ခုချင်းက record ဖြစ်ပါတယ်။
- Table တွေကြားမှာ primary key၊ foreign key တွေနဲ့ ချိတ်ဆက်ထားလို့ relational ဖြစ်ပါတယ်။
- SQL (Structured Query Language) ဆိုတဲ့ standard language ကို အသုံးပြုပါတယ်။
- Banking system, POS software, HR system, school database စတဲ့ data အရမ်းတိကျရမယ့် application တွေကို စနစ်တကျတည်ဆောက်ဖို့အတွက် SQL-based database systems တွေကို အထူးအသုံးပြုကြပါတယ်။
- SQL database နည်းပညာတွေမှာ MySQL, PostgreSQL, Oracle, SQL Server စတဲ့ platform တွေ ပါဝင်ပါတယ်။

NoSQL (Non-Relational Databases):

- NoSQL database တွေဟာ flexibility ဖြစ်တာကြောင့် လက်ရှိမှာ လူသုံးများတဲ့ applications တွေအတွက် အထူးသင့်တော်ပြီး အသုံးအများဆုံး ဖြစ်လာနေပါတယ်။
- schema-less ဒါမှမဟုတ် dynamic schema ဖြစ်တဲ့အတွက် data structure ကို ကြိုတင်သတ်မှတ်ထားစရာ မလိုပါဘူး။
- JSON documents, Key-Value pair, Graphs, Column-store စတဲ့ format အမျိုးမျိုးနဲ့ သိမ်းဆည်းပါတယ်။
- NoSQL database များမှာတော့ Join မလိုအပ်ဘဲ တစ်နေရာထဲမှာ data အပြည့်အစုံကို သိမ်းဆည်းလေ့ရှိပါတယ်။
- SQL syntax ကိုမသုံးဘဲ ကိုယ်ပိုင် query method (ဥပမာ - API ဒါမှမဟုတ် custom query language) နဲ့ data ကို ရှာဖွေလုပ်ဆောင်ပါတယ်။
- social media feeds, real-time chat systems, IoT sensor data, နဲ့ online product catalogs တို့က NoSQL databases တွေမှာ အသုံးအများဆုံး use cases တွေဖြစ်ပါတယ်။
- နာမည်ကြီး NoSQL databases တွေမှာတော့ MongoDB (Document DB), Redis (Key-Value DB), Cassandra (Wide-column DB), နဲ့ Neo4j (Graph DB) တို့ဖြစ်ပါတယ်။

ဥပမာ - (NoSQL - MongoDB)

```
db.customers.insertOne({
  customer_name: "Henry",
  email: "henry@example.com",
  preferences: { language: "English", theme: "Dark" }
});
```

MongoDB မှာ **insertOne** method ကို အသုံးပြုပြီး document တစ်ခုကို **customers** ဆိုတဲ့ collection ထဲသို့ ထည့်သွင်းနိုင်ပါတယ်။ ဒီဥပမာထဲမှာ **preferences** ကတော့ nested JSON object အနေနဲ့ **"language"** နဲ့ **"theme"** ဆိုတဲ့ key-value တွေကို သိမ်းဆည်းထားတာဖြစ်ပါတယ်။

MongoDB ရဲ့ collection ဆိုတာကတော့ documents တွေကို စုစည်းထားတဲ့ နေရာတစ်ခုဖြစ်ပါတယ်။ document တစ်ခုချင်းစီဟာ structure တူဖို့ မလိုအပ်ပါဘူး။ ဥပမာ - တစ်ခုမှာ **name** တစ်ခုတည်း ရှိနိုင် သလို နောက်တစ်ခုမှာတော့ **name** နဲ့ **email** နှစ်ခုလုံး ပါဝင်နိုင်ပါတယ်။ ဒါကတော့ MongoDB လို NoSQL database တွေရဲ့ flexibility ဖြစ်မှုပါ။

ဒါနဲ့ ပြောင်းပြန်ကြည့်ရရင် traditional SQL databases မှာတော့ table တွေမှာ schema တူညီမှု လိုအပ်ပါတယ်။ ဆိုလိုတာက row တစ်ခုချင်းစီဟာ column အရေအတွက်၊ data type အမျိုးအစားတွေ တူညီဖို့ လိုပါတယ်။

ဒါကြောင့် MongoDB ကတော့ တချို့သော dynamic data structures အတွက် ပိုအဆင်ပြေပါတယ်။

အခုအချိန်မှာတော့ SQL နဲ့ NoSQL ဆိုတာ ဘာလဲ။ ဘယ်လိုကွာခြားချက်တွေရှိလဲ ဆိုတာနဲ့အတူ ဘယ်အခြေအနေမျိုးမှာ SQL ကို သုံးသင့်တယ်၊ ဘယ်လို application မျိုးတွေ တည်ဆောက်တဲ့အခါမှာ NoSQL က ပိုအဆင်ပြေတယ် စတာတွေကို အနည်းအကျဉ်း နားလည်ထားလောက်ပြီလို့ ယူဆပါတယ်။

5. Exercises

ရှေ့အခန်းတွေမှာ တည်ဆောက်ပြီးခဲ့တဲ့ **students** နဲ့ **scores** table ကို ပြန်လည်အသုံးပြုပါ။

Question 1:

Write and call a **stored procedure** that calculates the **total score** for a single student from the **scores** table.

Question 2:

Create a **trigger** that automatically calculates the **average score** whenever a new score is inserted into the **scores** table and saves it into the **score_log** table.

Question 3:

Add a **preferences** JSON column to the **students** table, set the **language** value to "English", and write a SQL query to find all students who prefer English.

Question 4:

Add a **profile** XML column to the **students** table, set the city value to "Mandalay", and write a SQL query to find all students from that city.

Question 5:

Create an event or function that calculates the **daily average score** from the **scores** table and saves the result into a **score_summary** table.

Chapter 15: Best Practices and Next Steps

နောက်ဆုံးအခန်းကို ရောက်ရှိလာတာဟာ စာဖတ်သူရဲ့ ကြိုးစားအားထုတ်မှုနဲ့ သင်ယူလိုစိတ်ကို အသိအမှတ်ပြုပါတယ်။ ဒီအထိ ရောက်လာနိုင်တာဟာ မလွယ်လှပါဘူး။ ဒါပေမယ့် SQL ကို တကယ်ထိထိရောက်ရောက် လက်တွေ့အသုံးပြုနိုင်ဖို့အတွက်တော့ စဉ်ဆက်မပြတ် လေ့လာမှုနဲ့ ဆက်လက်သင်ယူမှုတွေက အရေးအကြီးဆုံးဖြစ်ပါတယ်။ ဒီအခန်းမှာတော့ စာဖတ်သူရဲ့ SQL journey ကို ဆက်လက်တိုးတက်ဖို့အတွက် အကောင်းဆုံးလေ့လာမှု အကြံဉာဏ်တွေနဲ့ လိုက်နာလုပ်ဆောင်သင့်တဲ့ လုပ်ထုံးလုပ်နည်းတွေကို မျှဝေသွားမှာပါ။ ပေါ့ပေါ့ပါးပါးနဲ့ လေ့လာသွားကြမှာဖြစ်ပါတယ်။

1. Writing Clean and Maintainable SQL

1.1 Formatting and Commenting

သပ်ရပ်ပြီး စနစ်တကျရေးသားထားတဲ့ SQL ကုဒ်ဟာ ဖတ်ရှုရလွယ်ကူသလို ပြုပြင်ထိန်းသိမ်းရာမှာလည်း အချိန်သက်သာစေနိုင်ပါတယ်။ လေ့လာသူတွေ၊ developer တွေအတွက် လွယ်လင့်တကူ ဖတ်ရှုနိုင်အောင် formatting နှင့် commenting ကို အထူးအရေးထားသင့်ပါတယ်။

// Unclean Code

```
SELECT c.customer_name,o.product,o.amount FROM customers c JOIN orders
o ON c.customer_id=o.customer_id WHERE o.amount>10000;
```

// Clean Code

```
-- Get customer name, product, and amount for orders over 10,000
SELECT
    c.customer_name,
    o.product,
    o.amount
FROM
    customers c
JOIN
    orders o
```

```
ON c.customer_id = o.customer_id
WHERE
o.amount > 10000;
```

Best Practices

- **Formatting:** SQL ကုဒ်ကို ဖတ်ရလွယ်အောင် **SELECT, FROM, WHERE** စတဲ့ clauses တွေကို စာကြောင်းအသစ်နဲ့ ရေးပါ။
- **Commenting:** -- သို့မဟုတ် /* */ ကို အသုံးပြုပြီး SQL ကုဒ်ရဲ့ ရည်ရွယ်ချက်ကို ဖော်ပြပေးပါ။ အထူးသဖြင့် မကြာခဏပြန်ပြင်ရမယ့် queries တွေအတွက် အသုံးဝင်ပါတယ်။
- **Capitalization:** SQL keyword များ (ဥပမာ - **SELECT, FROM, WHERE**) ကို အကြီးစာလုံးဖြင့် ရေးသားပါ။ ဒါကတော့ keyword နှင့် column names တို့ကို ခွဲခြားပြီးသိအောင် ကူညီပေးပြီး ဖတ်ရလွယ်ကူစေပါတယ်။

1.2 Naming Conventions

- **Tables & Columns:** ဖတ်ရလွယ်ကူပြီး ရည်ရွယ်ချက်ရှင်းလင်းတဲ့ နာမည်တွေကိုသာ သုံးပါ။ ဥပမာ - **customers, order_date** တို့လိုပါ။
- **Consistency:** တသမတ်တည်းဖြစ်အောင် နာမည်ပေးဖို့ လိုပါတယ်။ ဥပမာ - အားလုံးကို English စာလုံးအသေးနဲ့ရေးတာ၊ စကားလုံးတွေကို _ နဲ့ ခြားရေးတာ (**customer_name, order_total** စသဖြင့်)။
- **Avoid:** SQL မှာ သုံးတဲ့ keywords (ဥပမာ - **TABLE, SELECT, ORDER**) တွေကို table name သို့မဟုတ် column name အဖြစ် မသုံးပါနဲ့။ syntax error မဖြစ်အောင်၊ ဖတ်သူအနေနဲ့လည်း ပိုနားလည်လွယ်အောင် ကူညီပေးပါတယ်။

2. Debugging and Troubleshooting Queries

2.1 Common Errors and Fixes

အောက်မှာတော့ SQL queries တွေရေးတဲ့အခါ မကြာခဏကြုံရတတ်တဲ့ အမှားတချို့နဲ့ အဲဒီအမှားတွေကို ပြန်ပြင်နိုင်တဲ့ နည်းလမ်းတွေကို ရှင်းပြထားပါတယ်။

Error 1: Syntax Error

```
SELECT customer_name, email, FROM customers;
```

Fix: , (ကော်မာ) တစ်ခု ပိုနေတာကြောင့် ဖယ်ရှားပေးဖို့ လိုအပ်ပါတယ်။

```
SELECT customer_name, email FROM customers;
```

Error 2: Table or Column Not Found

```
SELECT name FROM customer;
```

Fix: Table နာမည် (customer) သို့မဟုတ် column နာမည် (name) မှားရေးသားတာတွေ ဖြစ်နိုင်ပါတယ်။ ပြန်စစ်ဆေးပြီး မှန်ကန်တဲ့နာမည်တွေကို အသုံးပြုပါ။

```
SELECT customer_name FROM customers;
```

Error 3: Ambiguous Column Name

တူညီတဲ့ column name တွေရှိတဲ့ query တစ်ခုမှာ column တစ်ခုချင်းစီဟာ ဘယ် table က လာတာလဲ ဆိုတာကို ရှင်းရှင်းလင်းလင်း မသတ်မှတ်ထားဘူးဆိုရင် ရှုပ်ထွေးမှု ဖြစ်လာတတ်ပါတယ်။

```
SELECT customer_id FROM customers JOIN orders ON customer_id = customer_id;
```

Fix: ရှင်းလင်းဖို့အတွက် table alias တွေကို အသုံးပြုပြီး column တစ်ခုချင်းစီဟာ ဘယ် table က ဆိုတာ ပြန်ပြောပေးဖို့ လိုပါတယ်။

```
SELECT c.customer_id  
FROM customers c  
JOIN orders o ON c.customer_id = o.customer_id;
```

2.2 Using EXPLAIN for Insights

EXPLAIN ဆိုတာကတော့ SQL query တစ်ခု run လုပ်တဲ့အခါ database engine က query ကို ဘယ်လိုအဆင့်ဆင့် စီမံခန့်ခွဲပြီး လုပ်ဆောင်သွားမယ်ဆိုတာကို ဖော်ပြပေးတဲ့ command ပါ။ ဒါ့ကြောင့် စွမ်းဆောင်ရည်ကို ကျဆင်းစေတဲ့နေရာတွေ (performance bottlenecks) ကို ရှာဖွေဖော်ထုတ်ဖို့အတွက် အထောက်အကူပြုပါတယ်။

```
EXPLAIN
SELECT c.customer_name, o.product
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.amount > 10000;
```

Query performance ကို ကောင်းချင်ရင် **EXPLAIN** ကို အသုံးပြုပြီး full table scan ဖြစ်နေသလား စစ်ဆေးနိုင်ပါတယ်။

3. Resources for Continued Learning

3.1 Books, Blogs, and Tutorials

ဒါကတော့ တခြား Programmer Youtubers တွေ SQL နဲ့ ပတ်သက်ပြီး လမ်းညွှန်ပေးထားတဲ့ resources တွေ ဖြစ်ပါတယ်။

Books:

- **"SQL in 10 Minutes, Sams Teach Yourself"** - SQL ကို အခြေခံကနေ လေ့လာချင်သူတွေအတွက် ရိုးရှင်းပြီး လေ့လာရလွယ်တဲ့ လမ်းညွှန်စာအုပ်တစ်အုပ် ဖြစ်ပါတယ်။
- **"SQL Performance Explained"** - SQL query တွေကို ပိုမိုထိရောက်အောင် ရေးသားနိုင်ဖို့အတွက် performance မြှင့်တင်တဲ့နည်းတွေကို ရှင်းလင်းဖော်ပြပေးထားပါတယ်။
- **"Practical SQL: A Beginner's Guide to Storytelling with Data"** - Data ကို ထိထိရောက်ရောက် စီမံခန့်ခွဲနိုင်ဖို့ SQL ကို လက်တွေ့အသုံးပြုတတ်အောင် သင်ကြားပေးတဲ့စာအုပ် ဖြစ်ပါတယ်။
- **"Learning SQL" (by O'Reilly)** - SQL ကို အစအဆုံး အခြေခံမှစပြီး အဆင့်မြင့်အထိ ရှင်းလင်းကောင်းမွန်တဲ့ နည်းလမ်းတွေနဲ့ ဖော်ပြထားတဲ့ စာအုပ်တစ်အုပ် ဖြစ်ပါတယ်။

Useful Blogs and Websites:

- **W3Schools** - www.w3schools.com/sql: SQL အတွက် အခြေခံသင်ခန်းစာတွေကို အခမဲ့ တင်ပြထားပြီး စတင်လေ့လာသူများအတွက် အထူးသင့်တော်ပါတယ်။
- **SQL Shack** - www.sqlshack.com: လက်တွေ့အသုံးပြုနိုင်မယ့် SQL ဆောင်းပါးတွေ၊ အတွေ့အကြုံတွေကို မျှဝေပေးတဲ့နေရာပါ။ SQL tutorials, diagrams, syntax guides နဲ့ hands-on examples တွေ ပြည့်စုံစုံပါတဲ့ သင်ယူဖို့အကောင်းဆုံး platform တစ်ခု ဖြစ်ပါတယ်။

Tutorials:

- **Khan Academy - SQL Course**: လက်တွေ့ပြန်လည်သုံးသပ်နိုင်တဲ့ interactive lessons တွေ ပါဝင်ပါတယ်။ SQL ကို လွယ်ကူစွာ သင်ယူနိုင်ဖို့ ကိုယ်တိုင်ရေးသားပြီး စမ်းသပ်နိုင်တဲ့ coding environment ကိုပါ support ပေးထားပါတယ်။
- **Mode Analytics SQL Tutorial**: Data analysis ကို SQL နဲ့ လေ့လာချင်သူတွေအတွက် အထူးအသုံးဝင်တဲ့ resource တစ်ခု ဖြစ်ပါတယ်။

3.2 Online SQL Communities

- **Stack Overflow** - stackoverflow.com: SQL နဲ့ ပတ်သက်တဲ့ မေးခွန်းတွေအပြင် နည်းပညာနဲ့ ဆိုင်တဲ့ မေးခွန်းတွေကိုပါ ဖြေဆိုပေးတဲ့ အကြီးမားဆုံး developer forum ဖြစ်ပြီး မသိသေးတဲ့ အကြောင်းအရာတွေရှိရင် မိမိအနေနဲ့ မေးခွန်းတင်ပြီး အခြား developer တွေဆီကနေ အကူအညီ ရယူနိုင်ပါတယ်။
- **Reddit r/SQL**: SQL စိတ်ဝင်စားသူတွေအတွက် အကြံပြုချက်တွေ၊ resource တွေ၊ အတွေ့အကြုံ မျှဝေရာနေရာတစ်ခု ဖြစ်ပါတယ်။
- **DBA Stack Exchange** - dba.stackexchange.com: Database management နဲ့ performance ပိုင်းမှာ အထူးပြုထားတဲ့ forum တစ်ခုဖြစ်ပါတယ်။
- **X (Twitter)**: #SQL သို့မဟုတ် #Database လို့ hashtag တွေကို အသုံးပြုပြီး နောက်ဆုံး SQL ပတ်သက်တဲ့ သတင်းတွေ၊ ဆောင်းပါးတွေနဲ့ တင်ပြချက်တွေကို လေ့လာနိုင်ပါတယ်။

4. Exploring SQL Certifications

4.1 Popular Certifications

ပညာရပ်တစ်ခုမှာ certificate တစ်ခု ရရှိထားတာဟာလည်း အရမ်းအကျိုးရှိပါတယ်။ SQL အသိအမှတ်ပြု လက်မှတ်တွေကို ရရှိထားရင် မိမိရဲ့ နည်းပညာနဲ့ ကျွမ်းကျင်မှုကို သက်သေပြနိုင်တဲ့အပြင် အလုပ်အကိုင်ရရှိ နိုင်ဖို့အတွက်ပါ ပိုအထောက်အကူ ဖြစ်ပါတယ်။ Popular ဖြစ်တဲ့ certificates တွေရှိပါတယ်။

- Oracle Certified Professional, MySQL Database Administrator
- Microsoft Certified: Azure Data Fundamentals
- IBM Certified Database Associate - DB2 Fundamentals

4.2 Preparing for Exams

ဒါကတော့ ကျောင်းသားတွေအတွက် ရည်ရွယ်တာပါ။

- Udemy၊ Pluralsight စတဲ့ online platforms တွေမှာ လေ့ကျင့်ခန်းတွေ ပြုလုပ်နိုင်ပါတယ်။
- MySQL, PostgreSQL, SQL Server တွေမှာ လက်တွေ့ စမ်းသပ်နိုင်ပါတယ်။
- Oracle: <https://education.oracle.com>
- Microsoft: <https://learn.microsoft.com/en-us/certifications/>

5. Exercises

Question 1:

Rewrite the following query to be clean and add comments:

```
SELECT student_name,score FROM students s JOIN scores sc ON  
s.student_id=sc.student_id WHERE score>80;
```

Question 2:

Find and fix the errors in the following query:

```
SELECT name, subject FROM student JOIN scores ON student_id = id WHERE  
score > 85;
```

Question 3:

Create an **index** on the **score** column in the **scores** table, then test the following query using **EXPLAIN**:

```
SELECT student_id, subject, score FROM scores WHERE score > 85;
```

Conclusion

ဒီစာအုပ်ကို အစအဆုံး စိတ်အားထက်သန်စွာ ဖတ်ရှုပေးတဲ့အတွက် စာဖတ်သူအသီးသီးအား ကျေးဇူးအထူး တင်ပါတယ်။

SQL ကို အခြေခံကနေစပြီး သေသေချာချာလေ့လာဖို့ စာဖတ်သူ မြုပ်နှံလိုက်တဲ့ အချိန်နဲ့အားထုတ်မှုတွေဟာ နောင်လာမယ့် နည်းပညာလမ်းခရီးအတွက် တန်ဖိုးရှိတဲ့ dedication တစ်ခု ဖြစ်လာမှာ သေချာပါတယ်။

ဒီစာအုပ်မှာ SQL ရဲ့ အခြေခံ syntax တွေနဲ့ advanced queries တွေကို ရှိရင်းပြီး နားလည်လွယ်အောင် ဖော်ပြခဲ့ပြီးပါပြီ။ ဒါပေမယ့် အခြေခံအပိုင်းတွေကို ဦးစားပေးပြီး ပိုင်ပိုင်နိုင်နိုင်ဖြစ်အောင် လက်တွေ့လေ့ လာဖို့က ပိုအရေးကြီးပါတယ်။ ပြီးမှ တဆင့်ချင်းစီ ဆက်လေ့လာသွားဖို့ ဖြစ်ပါတယ်။

ဒီစာအုပ်ဟာ beginner level ထက်ပိုပြီး practical knowledge အခြေပြုတဲ့ စာအုပ်တစ်အုပ် ဖြစ်ပါတယ်။ ကိုယ်ပိုင် project တွေမှာ အသုံးပြုလို့ရမယ့် လက်တွေ့အသုံးဝင်တဲ့ concept တွေကိုလည်း ထည့်သွင်း ဖော်ပြပေးခဲ့ပါတယ်။ သင်ယူခဲ့တဲ့အရာတွေကို လက်တွေ့လုပ်ဆောင်ကြည့်သလို ကိုယ်ပိုင် idea တွေနဲ့ပါ ပေါင်းစပ်ပြီး အသစ်အသစ်တွေ ဖန်တီးကြည့်ပါ။

အောင်မြင်မှုဆိုတာ နေ့ချင်းညချင်းဝေဆာလာတာမျိုး မဟုတ်တဲ့အတွက် နေ့စဉ်လေ့လာရင်း ကြိုးစားရင်းနဲ့ တဖြည်းဖြည်း တိုးတက်လာမှာပဲ ဖြစ်ပါတယ်။ ကျွန်တော်ကိုယ်တိုင်လည်း ဆက်လက်လေ့လာစရာရှိတာ တွေကို လေ့လာနေဆဲ ဖြစ်ပါတယ်။

နောက်ဆုံးအနေနဲ့ စာဖတ်သူရဲ့ နည်းပညာလမ်းခရီးမှာ ဆက်လက်အောင်မြင်မှုတွေ ရရှိခံစားနိုင်ပါစေလို့ ကျွန်တော်စာရေးသူ ဆုမွန်ကောင်းတောင်းပေးလိုက်ပါတယ်။

— Ruben Htun

၂၀၂၅ ခုနှစ် ဇူလိုင်လ ၁၅ ရက်

နံနက် ၁ နာရီ ၂၇ မိနစ် တွင် ရေးသားပြီးစီးပါသည်။

References

DeBarros, A. (2020). *Practical SQL: A beginner's guide to storytelling with data* (2nd ed.).

Saturngod. (2020, October 27). *Database basics with MySQL*. Retrieved from <https://dbbasic.saturngod.net/book/print.html>

Official Documentation

PostgreSQL Global Development Group. (n.d.). *PostgreSQL: Installation Tutorial*. Retrieved from <https://www.postgresql.org/docs/current/tutorial-install.html>

W3Schools Guide

W3Schools. (n.d.). *PostgreSQL Installation*. Retrieved from https://www.w3schools.com/postgresql/postgresql_install.php

macOS Installation Video Tutorial

Coding With Tashi. (2021, October 25). *How to Install PostgreSQL on Mac OS*. [YouTube Video]. <https://www.youtube.com/watch?v=wCMXbM5J0X8&t=5s>

Windows PSQL & pgAdmin 4 Setup Tutorial

TubeMint. (2021, June 13). *How to Install PostgreSQL and pgAdmin 4 on Windows 10*. [YouTube Video]. <https://www.youtube.com/watch?v=XxyxkmH87nU>

macOS PSQL Setup Tutorial

Prisma. (2022, July 4). *How to Use PostgreSQL PSQL CLI on Mac*. [YouTube Video]. <https://www.youtube.com/watch?v=feuRPRpy0VU>

A Journey Through Data

As each page turns, new ideas unfold,
Where queries move and insights are told.

With every line, new patterns appear,
In data's depth, the truth grows clear.

Dive deep, explore, let questions flow,
In SQL's stream, your skills will grow.

May this book guide your eager mind,
To treasures that you seek to find.

Happy reading, and onward sail,
Through data's vast and wondrous trail.

-Ruben-