BASIC SQL

FOR HEALTHCARE DATA SCIENCE

Electronic Medical Record Project included

DR. YE THI HA HTWE

BASIC SQL FOR HEALTHCARE DATA SCIENCE

FOR BUN BUN

Table of Contents

CHAPTER 1
INTRODUCTION

1 Chapter 1 - Introduction

နည်းပညာအသစ်တစ်ခုကို စတင်လေ့လာတဲ့အခါမှာ နည်းပညာကလည်းအသစ်ဖြစ်တာရယ်၊ ဘာသာစကားအခက်အခဲကြောင့်ရယ် တော်ရုံနဲ့ရှေ့မရောက်ဘဲ ရှိတတ်ပါတယ်။ ဒီစာအုပ်ကတော့ **SQL** နည်းပညာစတင်လေ့လာရာမှာ ဘာသာစကားအခက်အခဲကို တစိတ်တပိုင်းဖြေရှင်းပေးနိုင်မယ်လို့ မျှော်လင့်ပါတယ်။ နည်းပညာလေ့လာရာမှာ ပထမဆုံးဖြည့်ဆည်းထားရမှာကတော့ **English reading** ဖြစ်ပါတယ်။ မူရင်းအင်္ဂလိပ်စာဖတ်နိုင်တဲ့လူက နည်းပညာအသစ်ဒီနေ့ထွက်ရင် ဒီနေ့တန်းဖတ်လို့ရပြီး သိပ်မဖတ်နိုင်တဲ့လူက ဘာသာပြန်တာစောင့်နေရတာ၊ နောက်မှဖတ်တော့မယ်ဆိုပြီး ပစ်ထားလိုက်ရာကနေ မဖတ်ဖြစ်တော့တာမျိုးရှိတတ်ပါတယ်။

စာဖတ်တာနဲ့ ဗီဒီယိုကြည့်တာမှာ စာဖတ်တာက အချိန်ကုန်ပိုသက်သာစေပါတယ်။ ကျွန်တော်တို့အများစုက **visual learner** တွေမို့လို့ နည်းပညာလေ့လာရာမှာ ဗီဒီယိုတွေကြည့်တာမျိုးကို ပိုအားသန်လေ့ရှိပါတယ်။ ဗီဒီယိုအများစုမှာက လိုရင်းကိုတော်တော်နဲ့မရောက်ဘဲ သိုင်းဝိုင်းနေတာကိုစောင့်ရတာ၊ ဗီဒီယိုတစ်ခုလုံးမှာ အရေးကြီးဆုံးအပိုင်းက ဘယ်နားမှာရှိမလဲဆိုတာ ကြိုခန့်မှန်းလို့မရတာမျိုးတွေရှိပါတယ်။ စာအုပ်ဖတ်ရင်တော့ ကိုယ့်အတွက်သိပ်အသုံးမဝင်တဲ့အပိုင်းကို ကျော်သွားလို့ရပြီး အရေးအကြီးဆုံးတွေကို ကျေညက်အောင် ထပ်တလဲလဲဖတ်နေလို့ရပါတယ်။

ဒါကြောင့်နည်းပညာလေ့လာတဲ့အခါမှာ ဗီဒီယိုကြည့်တာအပြင် စာဖတ်တဲ့အပိုင်းကိုပါ ဦးစားပေးဖို့တိုက်တွန်းလိုပါတယ်။

1.1 Databases and Database Management Systems

Database ဆိုတာ အချက်အလက်တွေကို စနစ်တကျ သိမ်းဆည်းထားတဲ့ နေရာပါ။ **Database** အမျိုးအစားများစွာရှိတဲ့အထဲက အဓိကအမျိုးအစားနှစ်ခုက **SQL** နဲ့ **NoSQL** ဖြစ်ပါတယ်။

1.1.1 SQL

Structured Query Language ရဲ့အတိုကောက်ဖြစ်ပါတယ်။ **Database** ထဲကို အချက်အလက်တွေထည့်သွင်းပြီး ပုံစံအမျိုးမျိုးနဲ့ပြန်ကြည့်ဖို့အတွက် သုံးတဲ့နည်းပညာဖြစ်ပါတယ်။ ဥပမာ လူတွေရဲ့မွေးနေ့တွေကို မှတ်ထားတယ်ဆိုပါစို့။ **SQL** နဲ့ဆို ဒီလိုရှာလို့ရပါတယ် - "ဒီနေ့ကနေကြည့်ရင် အနီးဆုံးမွေးနေ့ရှင်ကဘယ်သူလဲ?" ဆိုရင်

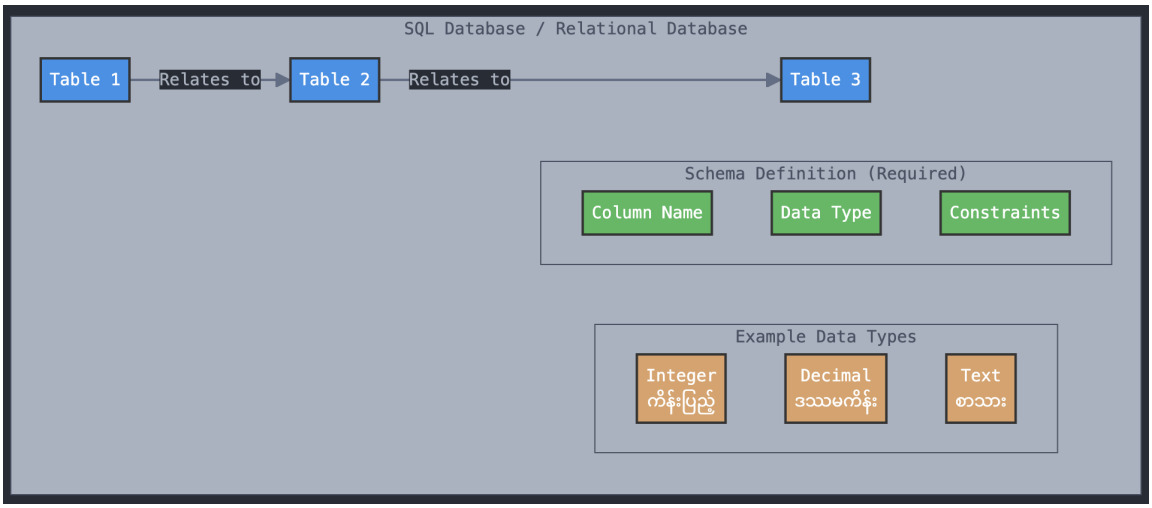
```sql
SELECT *
FROM people
WHERE birthday >= CURRENT_DATE
ORDER BY birthday ASC
LIMIT 1;
```

people

id	name	birthday
1	မောင်မောင်	1995-12-28
2	အေးအေး	1998-01-15
3	စန်းစန်း	1990-03-20
4	သန်းသန်း	1992-06-10
5	ကျော်ကျော်	1997-09-05
INT	VARCHAR(255)	DATE

ဒါဆိုရင် ဒီနေ့နောက်ပိုင်း အနီးဆုံးမွေးနေ့ရှင်ကို ရှာပေးပါလိမ့်မယ်။ တကယ်လို့ အားလုံးရဲ့ ဒီနှစ်မွေးနေ့တွေ ကျော်သွားပြီဆိုရင်တော့ နောက်နှစ်မွေးနေ့ကိုပြပါလိမ့်မယ်။

SQL ဒေတာဘေ့စ် တွေကို **relational database** တွေလို့လည်းခေါ် ကြပါတယ်။ **SQL table** တခုနဲ့တခုချိတ်ဆက်ပြီးအလုပ်လုပ်နိုင်တဲ့အတွက်ကြောင့် **relational** လို့ခေါ်ရတာဖြစ်ပါတယ်။ **SQL** ဒေတာဘေ့စ်တွေမှာ **schema** လို့ခေါ်တဲ့ ဒေတာဖွဲ့စည်းပုံ ကြိုသတ်မှတ်ထားဖို့ လိုပါတယ်။ ဥပမာ **table** တစ်ခုဆောက်မယ်ဆိုရင် ဒေတာတွေမထည့်ခင် **column** တွေက ဘာတွေပါမလဲ၊ ဒီ **column** တွေထဲမှာ ကိန်းပြည့် **(integer)** တန်ဖိုးတွေထည့်မလား၊ **(decimal)** တန်ဖိုးတွေထည့်မလား စတာတွေကို အရင်ဆုံးသတ်မှတ်ရပါတယ်။

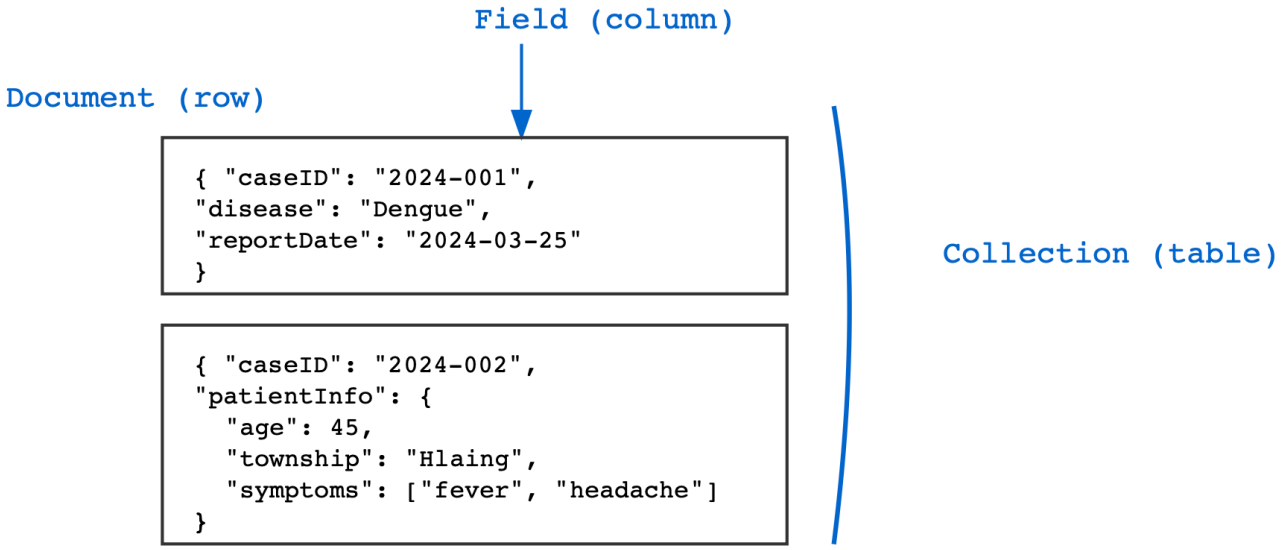


ဒါပေမယ့် ရံဖန်ရံခါ ဒေတာတွေကို ဒီလိုစနစ်တကျကြိုတင်သတ်မှတ်ပေးထားလို့မရတဲ့ အခြေအနေတွေရှိပါတယ်။ **column** တွေက တသမတ်တည်းမဟုတ်ဘဲ အမြဲပြောင်းလဲဖို့လိုတဲ့ အခြေအနေမျိုးမှာဖြစ်ဖြစ်၊ ဒေတာအများကြီးကို အစုလိုက်အပြုံလိုက်သိမ်းဆည်းပြီး ရယူသုံးစွဲဖို့လိုတဲ့ အခြေအနေတွေလည်း ရှိပါတယ်။ အဲဒီလိုအခြေအနေတွေအတွက် **NoSQL** ကို သုံးကြပါတယ်။ အထူးသဖြင့် **message** ပို့တဲ့ **application** တွေမှာ **NoSQL** ဒေတာဘေ့စ်အမျိုးအစားအသုံးပြုရလေ့ရှိပါတယ်။

1.1.2 NoSQL

NoSQL ရဲ့အရှည်က **Not only SQL** ဖြစ်ပါတယ်။ **NoSQL** ရဲ့ဖွဲ့စည်းပုံကိုက **table** နဲ့မဟုတ်ဘဲ ပုံစံအမျိုးမျိုးနဲ့သိမ်းနိုင်ပါတယ်။ **NoSQL** ဒေတာဘေ့စ်တွေကို **non-relational database** လို့လည်းခေါ်ပါတယ်။ **SQL** မှာတုန်းက **table schema** ကိုပုံသေသတ်မှတ်ပေးထားရပေမယ့် **NoSQL** မှာ အဲဒီလိုကြိုသတ်မှတ်ပေးထားဖို့မလိုပါဘူး။ **dynamic schema** လို့ခေါ်ပါတယ်။ နောက်တစ်ခုက **NoSQL** ဒေတာဘေ့စ်တွေကို **horizontal scaling** လုပ်လို့ရပါတယ်။ သဘောကတော့ ဒေတာတွေကိုကွန်ပြူတာတစ်လုံးထက်မက နေရာခွဲပြီးသိမ်းလို့ရပါတယ်။

NoSQL ဒေတာဘေ့စ်တွေထဲမှာ အသုံးအများဆုံးက **MongoDB** ဖြစ်ပါတယ်။ **MongoDB** မှာ ဒေတာတွေကို **table** နဲ့မသိမ်းဘဲ **collection** ပုံစံနဲ့သိမ်းပါတယ်။ **SQL** ရဲ့ **column** လိုမျိုးကို **NoSQL** မှာ **field** လို့ခေါ်ပါတယ်။ ပုံမှာကြည့်မယ်ဆိုရင် **NoSQL document** တခုစီမှာပါဝင်တဲ့ **field** အရေအတွက်ကော၊ အမျိုးအစားကော မတူညီတာကိုတွေ့ရမှာဖြစ်ပါတယ်။



ဒီစာအုပ်မှာ SQL အကြောင်းကိုပဲအဓိကဖော်ပြသွားမှာပါ။

1.1.3 Database Management Systems (DBMS)

နည်းပညာလေ့လာတဲ့အခါမှာ PostgreSQL နဲ့ SQLite လို စကားလုံးတွေကို ခဏခဏတွေ့နေရပြီး ရိုးရိုး SQL နဲ့ဘာကွာလဲသိချင်စိတ်မကြာခဏဖြစ်ကြမယ်ထင်ပါတယ်။ MySQL, PostgreSQL, SQLite စတာတွေက Database Management System (DBMS) တွေပါ။ DBMS ဆိုတာ database တစ်ခုခုနဲ့ အလုပ်လုပ်ဖို့သုံးတဲ့ ဆော့ဖ်ဝဲပါ။ table တွေနဲ့တည်ဆောက်ထားတဲ့ ဒေတာဘေ့စ်တွေစီမံခန့်ခွဲဖို့ကိုတော့ Relational Database Management Systems (RDBMS) တွေအသုံးပြုရပါတယ်။

RDBMS တွေမှာ SQL ကို ကိုယ့်နည်းကိုယ့်ဟန်နဲ့စီမံခန့်ခွဲတာမို့လို့ ဆော့ဖ်ဝဲတခုနဲ့တခု ရေးထုံး (syntax) အနည်းငယ်ကွာခြားပါတယ်။ ဥပမာ ဒေတာ ၁၀ ကြောင်းကို ထုတ်ပြမယ်ဆိုရင် RDBMS ငါးမျိုးမှာ ရေးထုံးနည်းနည်းစီကွဲပါတယ်။

MySQL, PostgreSQL နဲ့ SQLite မှာ

```
SELECT * FROM people LIMIT 10;
```

Microsoft SQL Server မှာ

```
SELECT TOP 10 * FROM people;
```

Oracle Database မှာ

```
SELECT * FROM people WHERE ROWNUM <= 10;
```

Microsoft SQL နဲ့ Oracle လို RDMBS ကတော့ ကုမ္ပဏီပိုင်ဖြစ်လို့ အခကြေးငွေပေးသုံးရပြီး ကျန်တာတွေကတော့ free and open source ဖြစ်လို့ အခမဲ့အသုံးပြုနိုင်ပါတယ်။

Name	Owner	Description
Microsoft SQL Server	Microsoft	Microsoft Azure နဲ့ .NET framework လိုမျိုးမှာ အသုံးများ Windows မှာအဓိကသုံး MSSQL (သို့) SQL Server လို့လည်းခေါ်
MySQL	Oracle	လူသုံးများတဲ့ open source RDBMS web development မှာအဓိကသုံး Oracle ပိုင်ဆိုပေမယ့် open source အဖြစ်ဆက်ပေးထား
Oracle Database	Oracle	Enterprise အမျိုးအစားလုပ်ငန်းကြီးတွေ အသုံးများ Oracle လို့လည်းခေါ်
PostgreSQL	Open Source	Docker နဲ့ Kubernetes လို open source နည်းပညာတွေနဲ့အတူ တွဲပြီးအသုံးများ data အမြောက်အများကိုင်တွယ်နိုင်
SQLite	Open Source	ကမ္ဘာ့အသုံးအများဆုံး RDBMS iOS နဲ့ Android OS တို့မှာအဓိကသုံး ပေါ့ပါးတာကြောင့် တကိုယ်ရည်သုံး database လေးတွေအတွက်သင့်တော်

1.2 Why do we need databases?

ဒေတာတွေကို စာရွက်ပေါ်မှာလည်း ရေးမှတ်လို့ရနေတာပဲ။ စာရွက်နဲ့မဟုတ်ရင်တောင် Excel လိုမျိုးမှာ ထည့်သိမ်းထားလို့ရနေတာပဲ၊ ဘာလို့ database တွေသုံးဖို့လိုတာလဲလို့ တော်တော်များများက မေးတတ်ကြပါတယ်။

ဥပမာ ကျန်းမာရေးဌာနတွေမှာ ကူးစက်ရောဂါတွေကို စောင့်ကြည့်လေ့လာတဲ့ မှတ်တမ်းတွေကိုကြည့်ရအောင်။ အဲ့ဒီမှတ်တမ်းတွေထဲမှာ -

- ရောဂါဖြစ်ပွားတဲ့ မြို့နယ် (township)
- ရောဂါအမည် (disease)
- လူနာအရေအတွက် (patient count)
- သေဆုံးမှုအရေအတွက် (mortality)
- ဖြစ်ပွားတဲ့ရက်စွဲ (reporting date)
- အသက်အုပ်စုအလိုက် ဖြစ်ပွားမှု (age distribution)
- လိင်အလိုက် ဖြစ်ပွားမှု (gender distribution)
- ရောဂါပိုးတွေ့ရှိမှု အခြေအနေ (microscopic examination)

စတာတွေ ပါပါတယ်။

အရင်တုန်းကဆို ဒီအချက်အလက်တွေကို စာအုပ်ကြီးတွေထဲမှာ ရေးမှတ်ထားတာ (ဒါမှမဟုတ်) Excel sheet တွေနဲ့ သိမ်းထားတာမျိုး လုပ်ကြတယ်။ ဒါပေမယ့် အချိန်ပေးရသလောက် အလုပ်မတွင်တာမျိုးအများကြီး တွေ့ရတယ်။ ဘာတွေလဲဆိုတော့ -

အချက်အလက်တွေက ထပ်နေတယ်
မြို့နယ်တစ်ခုက တင်ပြတဲ့အချက်အလက်တွေကို ခရိုင်၊ ပြည်နယ်၊ ဗဟိုဆိုပြီး နေရာအသီးသီးမှာ ထပ်ခါထပ်ခါ ရေးနေရတယ်။ အောက်ခြေကသွင်းလိုက်တဲ့စာရင်းဟာ အပေါ်ရောက်လေလေ၊ စာရင်းထပ်သွင်းရလေလေဖြစ်နေတယ်။

ပြင်ဆင်ရတာခက်တယ်
ဥပမာ - လူနာအရေအတွက် မှားနေလို့ ပြင်ရမယ်ဆိုရင် နေရာတိုင်းမှာ တစ်ခုချင်းစီ ပြင်နေရတယ်။

ရှာရတာခက်တယ်
ပြီးခဲ့တဲ့ ဖေလအတွင်း ဝမ်းကိုက်ရောဂါ ဘယ်နှစ်ကြိမ်ဖြစ်ပွားခဲ့သလဲဆိုတာ သိချင်ရင် စာအုပ်အားလုံး တစ်ရွက်ချင်း လှန်ပြီး ရှာနေရတယ်။

အချက်အလက်တွေ ပျောက်သွားနိုင်တယ်
မှတ်တမ်းစာအုပ်တွေ ပျက်စီးတာ၊ ဖိုင်တွေ ပျက်သွားတာမျိုး ဖြစ်နိုင်တယ်။

အချက်အလက်တွေကို လေ့လာဆန်းစစ်ဖို့ ခက်တယ်
ဥပမာ - ဌက်ဖျားရောဂါက မိုးရာသီမှာ ပိုဖြစ်လား၊ ဆောင်းရာသီမှာ ပိုဖြစ်လားဆိုတာ လေ့လာချင်ရင် အခြိန်အရမ်းကုန်တယ်။

ဒါကြောင့် Database တွေကို သုံးလာကြပါတယ်။ Database သုံးလိုက်ရင် အချက်အလက်တွေကို စနစ်တကျ သိမ်းလို့ရပြီး လိုချင်တဲ့ အချက်အလက်ကို ခလုတ်နှိပ်လိုက်ရုံနဲ့ ချက်ချင်းရှာလို့ရတယ်။ အရေးကြီးတဲ့ မှတ်တမ်းတွေကို လုံခြုံစိတ်ချစွာ သိမ်းထားနိုင်ပြီး မြို့နယ်၊ ခရိုင်၊ ပြည်နယ်က ဝန်ထမ်းတွေ တစ်ပြိုင်နက်တည်း သုံးလို့ရပါတယ်။

ရောဂါဖြစ်ပွားမှုပုံစံ၊ ရာသီအလိုက် ရောဂါဖြစ်ပွားမှု စတာတွေကို အလွယ်တကူ လေ့လာဆန်းစစ်နိုင်ပါတယ်။

ဒါကြောင့် ကျန်းမာရေးစောင့်ကြည့်လေ့လာမှုစနစ် (disease surveillence) အပါအဝင် သတင်းအချက်အလက်စီမံခန့်ခွဲမှုစနစ်တွေမှာ Database တွေဟာ မရှိမဖြစ် အရေးပါလာပါတယ်။ ဒီအချက်အလက်တွေကို သုံးပြီး ရောဂါတွေကို ကြိုတင်ခန့်မှန်းနိုင်သလို၊ ကာကွယ်ထိန်းချုပ်ဖို့အတွက်လည်း အထောက်အကူ ဖြစ်စေပါတယ်။

1.3 Introduction to MySQL and its role in the database world

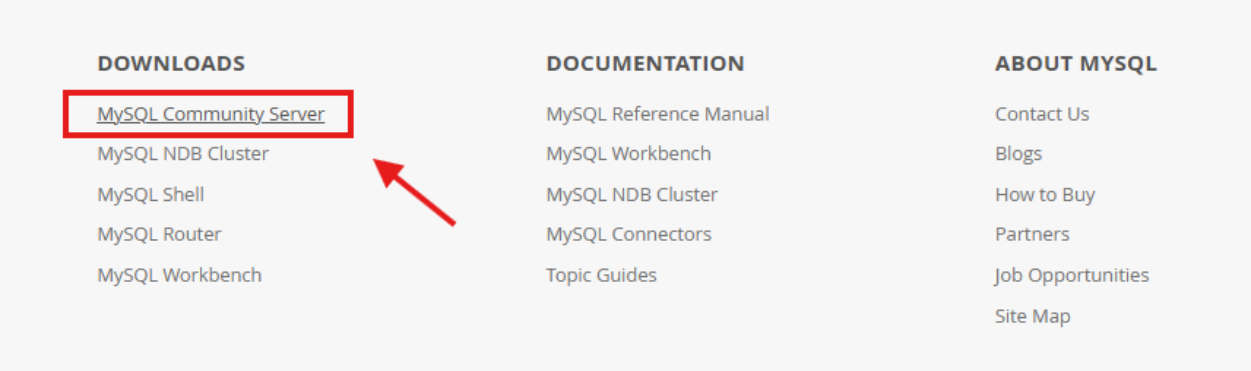
လေ့လာကာစမှာ ကျွန်တော်တို့အမှတ်မှားတတ်တာက database အတွက်အသုံးပြုနေတဲ့ software ကို database လို့ပဲထင်နေတတ်တာမျိုးပါ။ MySQL လို database management system (DBMS) တွေက database မဟုတ်ပါဘူး။ database တွေစီမံခန့်ခွဲပေးတဲ့ software ပဲဖြစ်ပါတယ်။

MySQL ဆိုတာ လွယ်လွယ်ပြောရရင် database တွေကို စီမံခန့်ခွဲဖို့အတွက် သုံးတဲ့ software တစ်ခုဖြစ်ပါတယ်။ ဥပမာ ကျန်းမာရေးဌာနရဲ့ "disease_surveillance" ဆိုတာက database ဖြစ်ပါတယ်။ အဲ့ဒီ database ထဲက အချက်အလက်တွေကို ဖန်တီးတာ၊ ပြင်ဆင်တာ၊ ရှာဖွေတာ၊ ဖျက်တာတွေ လုပ်နိုင်ဖို့ MySQL က software အနေနဲ့ ဆောင်ရွက်ပေးနေတာပါ။

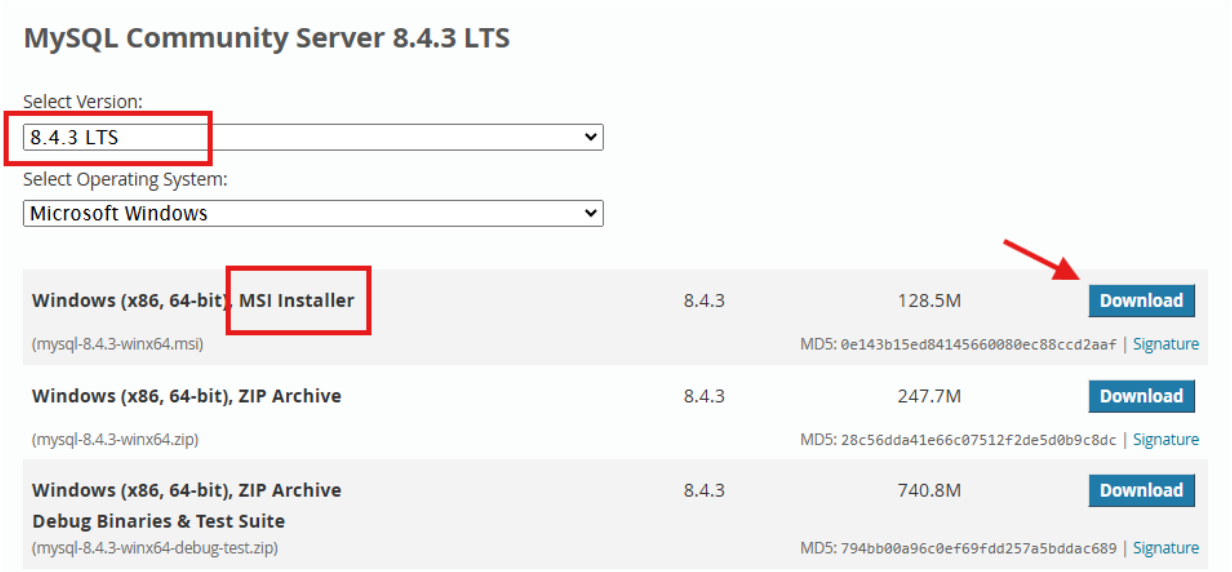
MySQL နည်းပညာကို Oracle ကဝယ်ထားလိုက်ပေမယ့် Open source ဖြစ်တဲ့အတွက် အခမဲ့သုံးလို့ရပါတယ်။ နည်းပညာချင်းဆင်တူတဲ့ PostgreSQL, Oracle, MicrosoftSQL တို့ရှိပေမယ့်လည်း ဒီစာအုပ်မှာတော့ သုံးရလွယ်ကူတဲ့ MySQL ကိုဦးစားပေးဖော်ပြသွားမှာဖြစ်ပါတယ်။

1.4 Installing and configuring MySQL

MySQL ကို Download လုပ်ရန် mysql.com website ကို သွားပါ

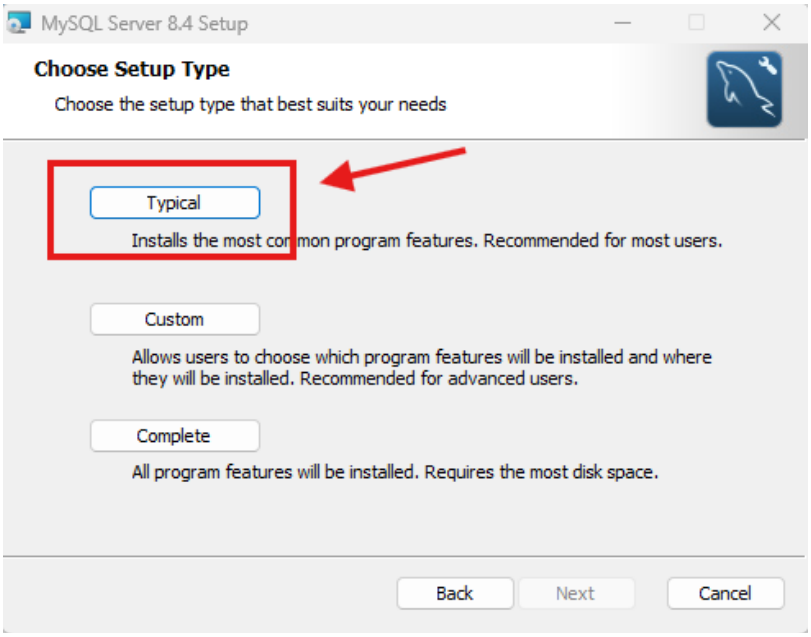


Downloads menu ကနေ "MySQL Community Server" ကို ရွေးပါ

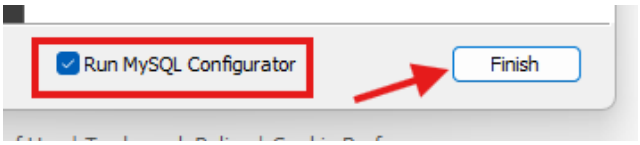


ကိုယ့်ရဲ့ computer OS ပေါ်မူတည်ပြီး (Windows/Mac/Linux) version ကို ရွေးပါ။ version မှာ LTS version ကိုရွေးပါ။ နောက်ဆုံး version တွေထက် LTS တွေက error ကင်းတဲ့အတွက်ကြောင့်ပါ။ MSI installer အမျိုးအစားကို download လုပ်ပြီး install ပါ။

Account ဝင်ခိုင်းရင် "No thanks, just start my download" ကိုပဲနှိပ်ပါ။

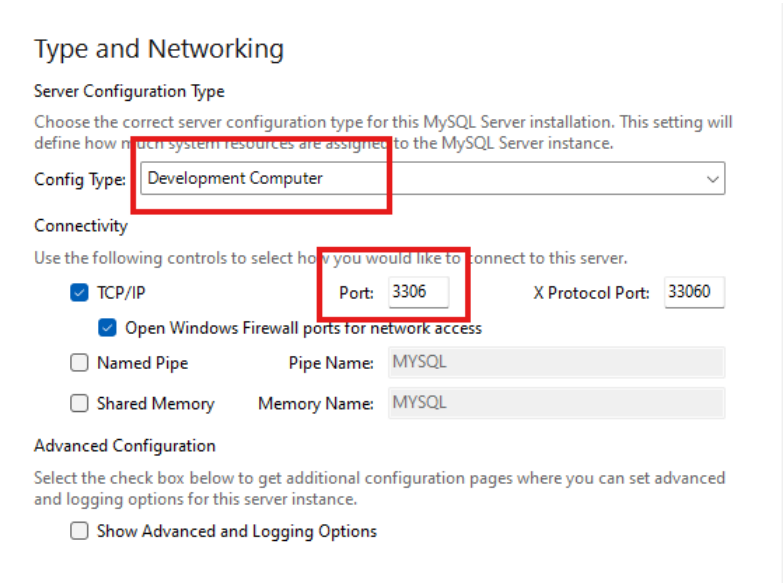


Installer ပွင့်လာရင် Next နဲ့ Accept တွေနှိပ်ပြီး Typical ကိုရွေးပါ။

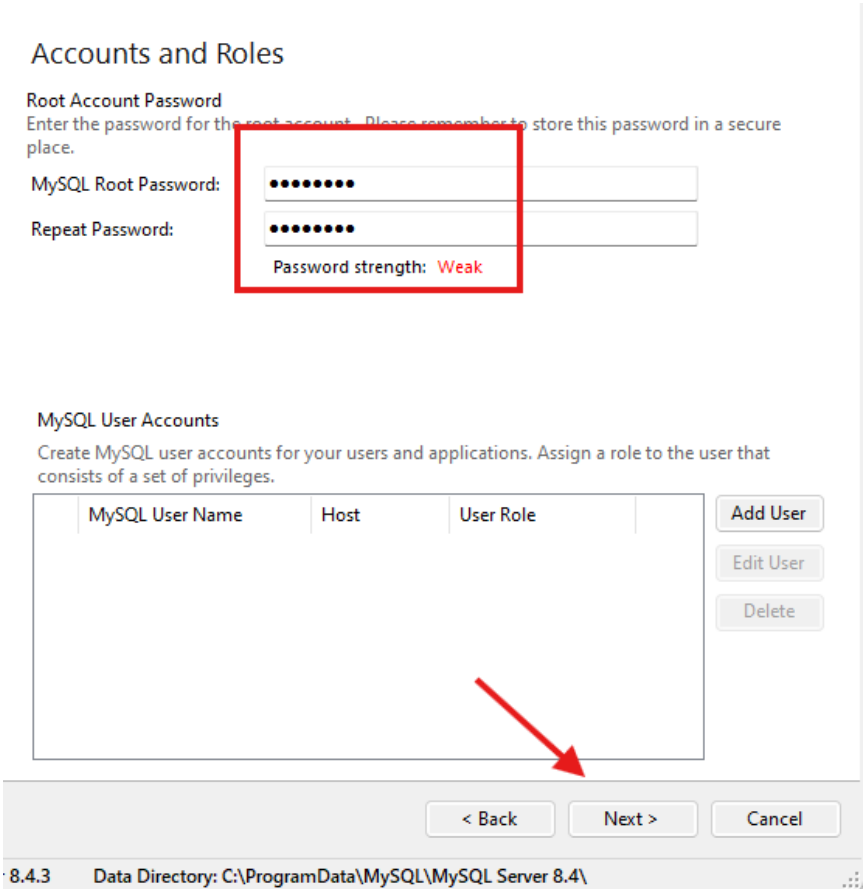


ပြီးသွားရင် "Run MySQL Configurator" ကိုအမှန်ခြစ်တဲ့အတိုင်းထားပြီး Finish နှိပ်ပါ။

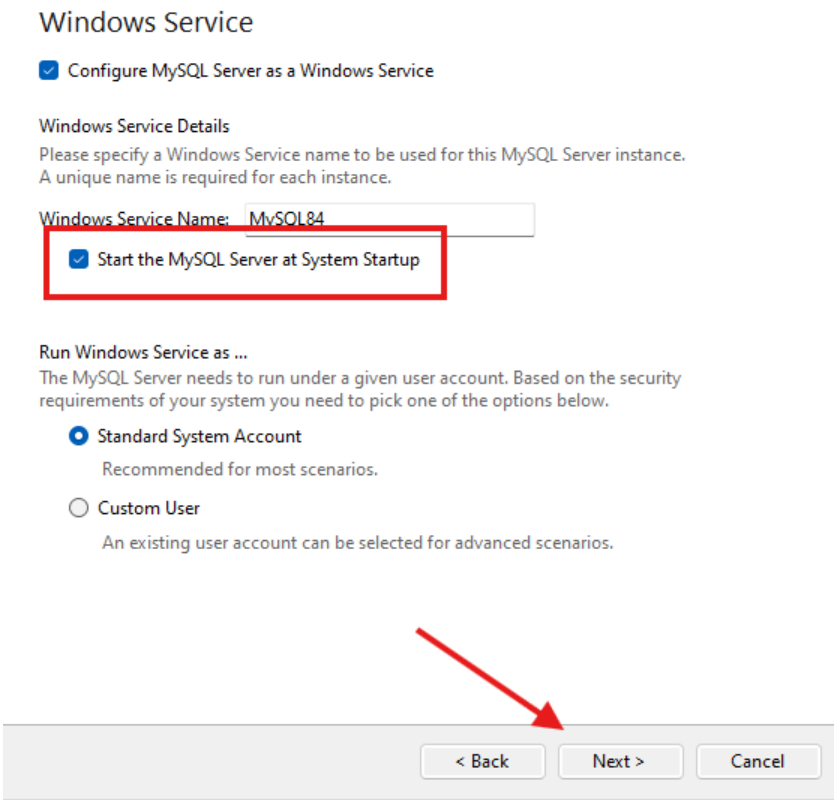
MySQL Server Configurator ပွင့်လာရင် Next နှိပ်ပါ။



Type and Networking ရောက်ရင် **config type** မှာ **"Development Computer"** ကိုပဲဆက်ရွေးပါ။ **Port** နံပါတ်ကိုမှတ်ထားပါ။ **default** က 3306 ဖြစ်ပါတယ်။

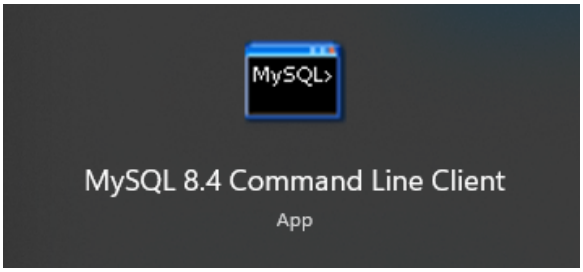


Accounts and Roles ရောက်ရင် **root password** နှစ်ကြိမ်ရိုက်ထည့်ပေးပါ။ အမြဲမှတ်မိနေမယ့် **password** ဖြစ်ရပါမယ်။ မသေချာရင်တနေရာမှာချရေးပါ။ ပြီးမှ **Next** နှိပ်ပါ။



"Start the MySQL Server at System Startup" ကိုအမှန်ခြစ်အတိုင်းထားပြီး Next နှိပ်ပါ။ ကွန်ပြူတာထဲမှာ တခြား MySQL server port တွေ run ဖို့ရှိရင်တော့ အမှန်ခြစ်ဖြုတ်ပေးရပါမယ်။ ဥပမာ web development လုပ်ဖို့ XAMPP လို development environment တွေထည့်မယ်ဆိုရင် XAMPP ရဲ့ MySQL server ကလည်း port 3306 သုံးတာမို့လို့ ဒီ MySQL Service ပွင့်နေရင် XAMPP ဘက်က MySQL service ဖွင့်မရတာမျိုးဖြစ်တတ်ပါတယ်။

Next ဆက်နှိပ်သွားရင်း Apply configuration ရောက်ရင် Execute ခလုတ်ကိုနှိပ်ပါ။ အားလုံးအမှန်ခြစ်ပြပြီးသွားရင် Next နှိပ်ပါ။ နောက်ဆုံးမှာ Finish နှိပ်ပါ။



ဒါဆိုရင် ကွန်ပြူတာမှာ MySQL ကို command line interface (CLI) နဲ့သုံးလို့ရပါပြီ။ Start ထဲမှာ mysql ရိုက်ထည့်လိုက်ရင် "MySQL command line client" တွေ့ပါမယ်။ ဖွင့်လိုက်ရင် terminal window တစ်ခုပွင့်လာပြီး password တောင်းပါလိမ့်မယ်။ install လုပ်တုန်းကပေးခဲ့တဲ့ password ထည့်ပေးလိုက်ရင် CLI ထဲဝင်လို့ရသွားမှာဖြစ်ပါတယ်။

```
show databases;
```

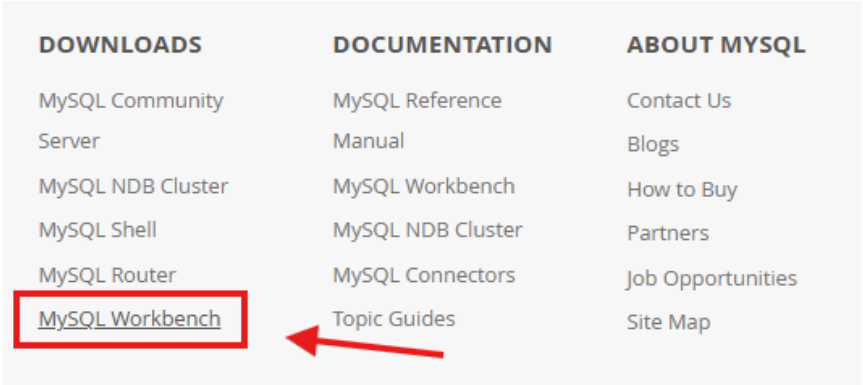
command ရိုက်ထည့်ပြီး **enter** နှိပ်လိုက်ရင် လက်ရှိ **database** နာမည်တွေမြင်ရပါမယ်။

```
exit;
```

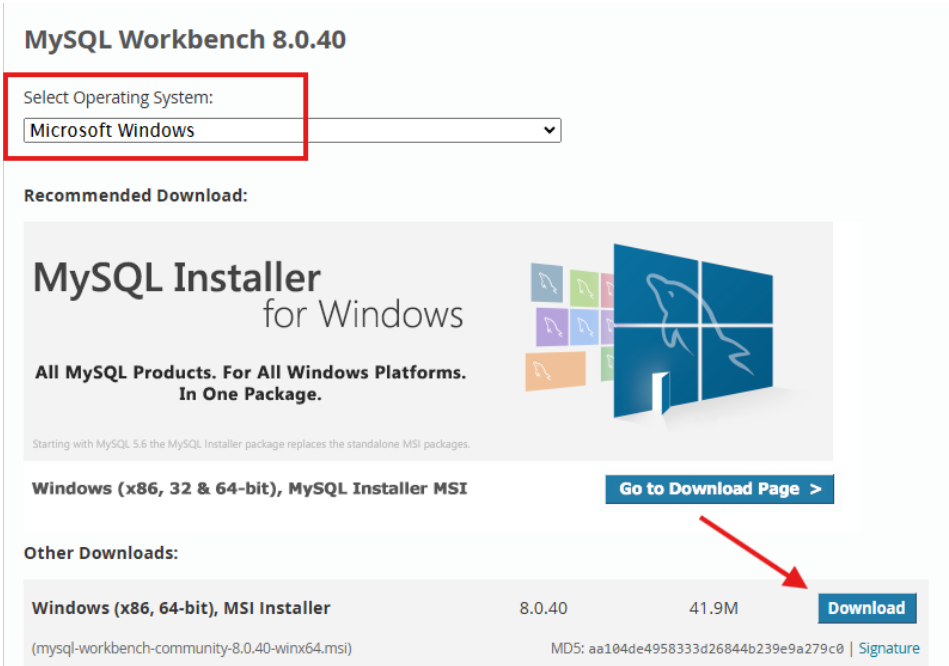
ရိုက်ထည့်၊ **enter** ခေါက်ပြီးပြန်ပိတ်ပါ။

1.5 MySQL Workbench

အခုမှစသုံးမယ့်သူတွေအတွက် **CLI** ကအနည်းငယ်ခက်ခဲနိုင်တာကြောင့် **graphical user interface (GUI)** နဲ့အသုံးပြုနိုင်တဲ့ **MySQL Workbench** ကိုထပ်ထည့်သွင်းဖို့လိုပါတယ်။

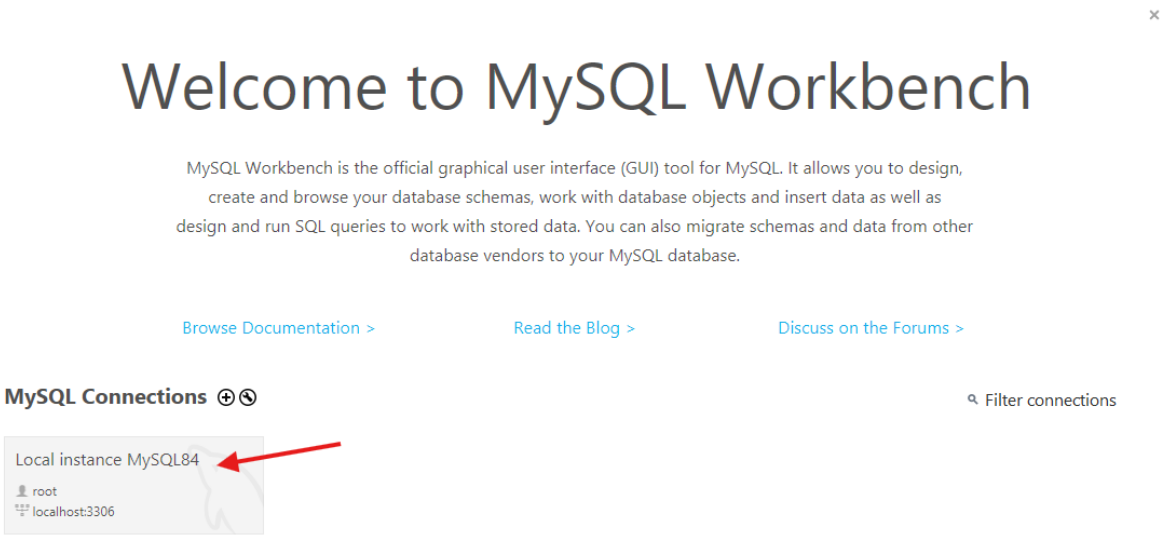


mysql.com မှာပဲ Downloads အောက်က MySQL Workbench ကိုနှိပ်ပါ။

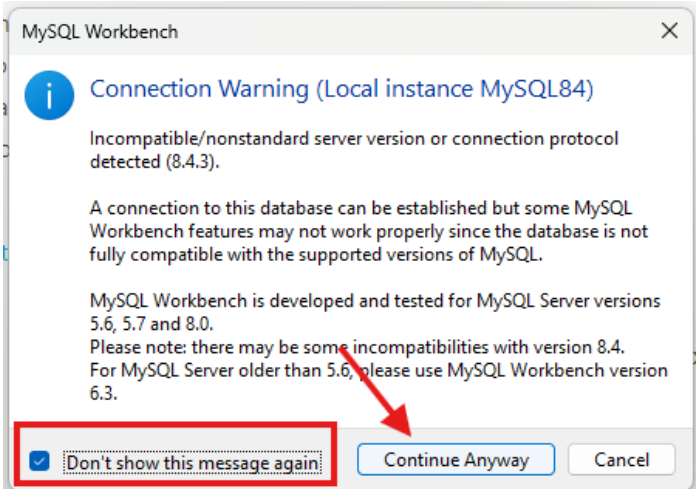


မိမိ **Operating system** ကိုရွေးပြီး **installer** ကို **Download** လုပ်ပါ။ **account** ဝင်ခိုင်းရင် **"No thanks, just start my download"** ကိုနှိပ်ပါ။ **Download** လုပ်ပြီးရင် **MySQL Workbench installer** ကိုဖွင့်ပါ။

Next နှိပ်သွားပြီး **Install** နှိပ်ပါ။ ပြီးသွားရင် **Finish** နှိပ်ပါ။ **MySQL Workbench** ပွင့်လာပါလိမ့်မယ်။



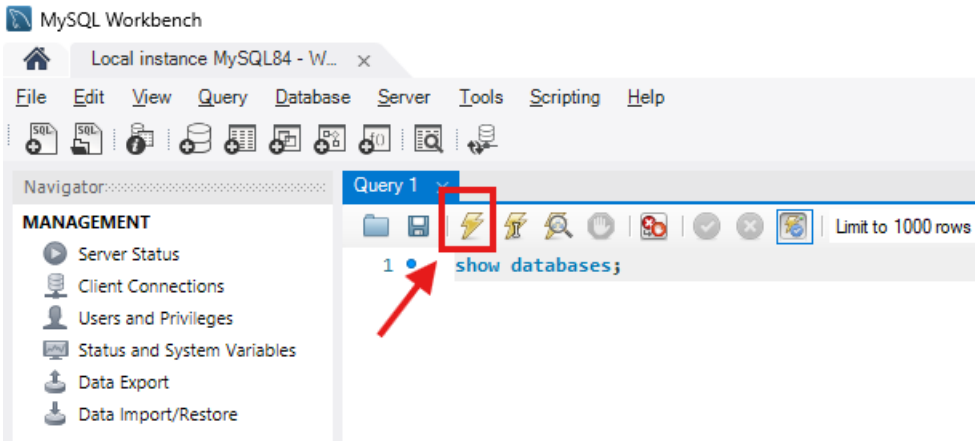
Local instance ကိုနှိပ်ပါ။ **password** ရိုက်ထည့်ပေးပါ။



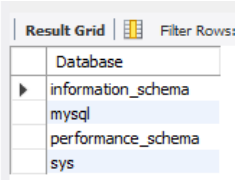
MySQL CLI က 8.4 ဖြစ်ပြီး **MySQL workbench** က 8 အထိပဲ **support** ပေးသေးတဲ့အတွက် **Warning** ပေါ်တတ်ပါတယ်။ **"Don't show this message again"** ကိုအမှန်ခြစ်ပြီး **"Continue Anyway"** ကိုနှိပ်ပါ။

MySQL Workbench ပွင့်လာပါလိမ့်မယ်။

```
show databases;
```



query ရိုက်ထည့်ပြီး **execute everything** ကိုနှိပ်ပါ။



CLI တုန်းကတွေ့ခဲ့တဲ့ရလဒ်အတိုင်းပဲ **database** တွေ တွေ့နိုင်ပါတယ်။

CHAPTER 2
BASIC MYSQL CONCEPTS

2 Chapter 2 - Basic MySQL concepts

2.1 Tables

ပထမဆုံး အနေနဲ့ **table** ဆိုတာ **Excel sheet** တစ်ခုလို့ မြင်ကြည့်ပါ။ ဒါပေမယ့် **Excel** ထက်ပိုပြီး စည်းစနစ်ကျတယ်လို့ ပြောလို့ရပါတယ်။

ဥပမာ - ရောဂါဖြစ်ပွားမှုတွေကို မှတ်တမ်းတင်တဲ့ **table** တစ်ခုရှိတယ် ဆိုကြပါစို့။ နာမည်က "**disease_cases**" ပါ။

Disease Cases Table

case_id	disease	township	date	patient_count	severity
1	ငှက်ဖျား	လှိုင်	2024-01-01	5	mild
2	ဝမ်းကိုက်	တာမွေ	2024-01-01	3	moderate
3	တီဘီ	လှိုင်	2024-01-02	2	severe

ဒီ table မှာ -

Column (ဒေါင်လိုက်တန်း) တွေက -

- **case_id** (နံပါတ်)
- **disease** (ရောဂါအမည်)
- **township** (မြို့နယ်)
- **date** (နေ့စွဲ)
- **patient_count** (လူနာအရေအတွက်)
- **severity** (ပြင်းထန်မှုအဆင့်)
  တို့ဖြစ်ပါတယ်။

Column တွေမှာသိမ်းချင်တဲ့ အချက်အလက်အမျိုးအစားအလိုက် **data type** တွေရှိပါတယ်။ ဥပမာ **case_id** က နံပါတ်နဲ့ပဲသိမ်းမယ်၊ **data** က **date format** နဲ့ပဲသိမ်းမယ်စသဖြင့်ပါ။

Row (အလျားလိုက်တန်း) တွေကတော့ အချက်အလက်တစ်ခုချင်းစီကိုကိုယ်စားပြုပါမယ်။

```
| 1 | ငှက်ဖျား | လှိုင် | 2024-01-01 | 5 | mild |
```

ဒါကို **row** လို့ခေါ်ပါတယ်။ မှတ်တမ်း **record** တစ်ခု (သို့) **entry** တစ်ခုလို့ ခေါ်ပါတယ်။ ငှက်ဖျားရောဂါတွေ့ရှိမှုနဲ့ ပတ်သက်တဲ့ အချက်အလက်အားလုံး ဒီတစ်ကြောင်းထဲမှာ ပါပါတယ်။

2.2 Constraints

column တွေမှာ **contraint** ဆိုတဲ့ကန့်သတ်ချက်တွေနဲ့ **data** ထည့်သွင်းမှုအမှားအယွင်းနည်းအောင် ထိန်းထားလို့ရပါတယ်။ ဥပမာ **case_id** က မဖြစ်မနေ ရှိရမယ်၊ ထပ်လို့မရဘူး၊ **AUTO_INCREMENT** နဲ့ အမြဲဂဏန်းတိုးသွားရမယ်လို့ သတ်မှတ်ပေးထားနိုင်ပါတယ်။

Primary Key

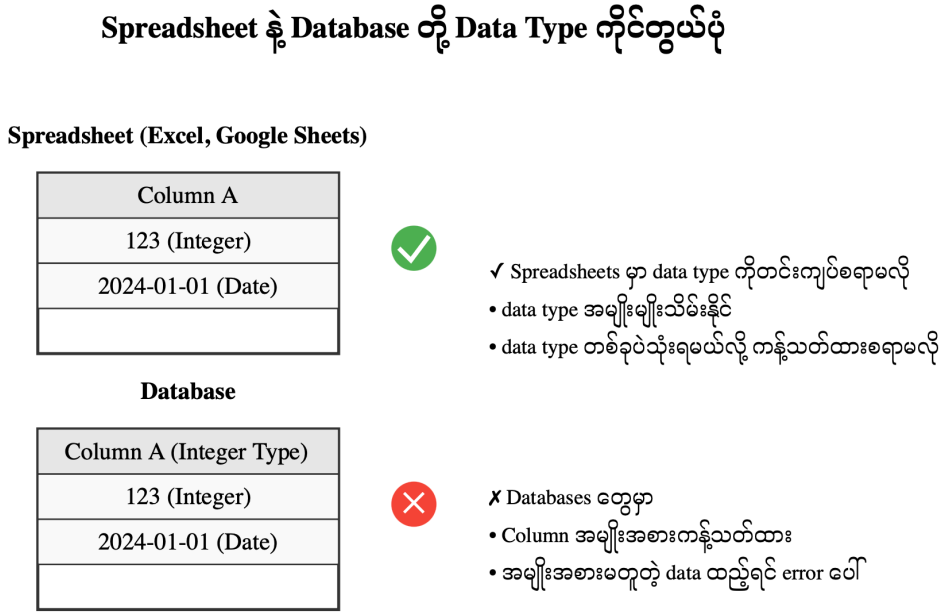
case_id လို ဂဏန်းထပ်လို့မရတဲ့ **column** ကို **Primary Key constraint** သတ်မှတ်ပေးလိုက်တဲ့အခါ **record** တစ်ကြောင်းချင်းစီကို မှတ်ပုံတင်နံပါတ်ပေးလိုက်တာနဲ့တူပါတယ်။ လူတစ်ယောက်မှာမှတ်ပုံတင်နံပါတ်တစ်ခုစီရှိသလိုပဲ **table row** တိုင်းမှာလည်း တစ်ခုနဲ့တစ်ခုမတူတဲ့ မှတ်ပုံတင်နံပါတ်ရှိမှ နောက်ပိုင်းမှာ **data** စီမံခန့်ခွဲရတာလွယ်ကူချောမွေ့စေမှာဖြစ်ပါတယ်။

Foreign Key

Foreign Key constraint က တခြား **table** တွေနဲ့ချိတ်ဆက်တဲ့အခါမှာ အသုံးပြုပါတယ်။ နောက်ပိုင်း **relationship** အပိုင်းရောက်ရင် **foreign key** အကြောင်းဆက်ပြောပါမယ်။

Data တွေကို အခုလို **table, column, row** တွေနဲ့ စနစ်တကျသိမ်းထားတဲ့အတွက် အချက်အလက်တွေရှာရတာလွယ်ကူပြီး **report** ထုတ်ရတာလည်း အဆင်ပြေချောမွေ့စေပါတယ်။ **Data analysis** လုပ်ဖို့လည်းလွယ်ကူပါတယ်။ ဒေတာတွေ ထပ်နေတာမျိုးမဖြစ်အောင် ကြိုတင်ကာကွယ်ထားလို့ရလို့ အမှားအယွင်းနည်းပါတယ်။

2.3 Data types in MySQL



Excel နဲ့ Google sheet တို့လို့ **spreadsheet software** တွေမှာ **column** တစ်ခုအောက်မှာ ကြိုက်တဲ့ **data** အမျိုးအစားလာထည့်လို့ရပါတယ်။ **column** တစ်ခုမှာ ပထမ **row** မှာထည့်ခဲ့တဲ့တန်ဖိုးက ကိန်းပြည့်ဖြစ်ပြီး နောက် **row** မှာထည့်တဲ့တန်ဖိုးက ရက်စွဲဖြစ်နေရင်လည်း ဘာ **error** မှာပြမှာမဟုတ်ပါဘူး။ **database** မှာတော့အဲဒီလိုဖြစ်လို့မရပါဘူး။ **column** တစ်ခုမှာ တူညီတဲ့ **data type** တွေပဲသိမ်းဆည်းလို့ရပါတယ်။ **Data type** တွေ မှန်မှန်ကန်ကန် သတ်မှတ်ထားခြင်းအားဖြင့် **Memory** သက်သာစေပါတယ်။ **Search**

လုပ်တဲ့အခါ မြန်စေပါတယ်။ ဒေတာတွေအလွဲလွဲအချော်ချော် ထည့်လို့မရအောင်လည်း တားဆီးပေးပါတယ်။ Data analysis လုပ်တဲ့အခါမှာလည်း လွယ်ကူစေပါတယ်။

2.3.1 Numbers

ဂဏန်းတွေ သိမ်းဖို့ပါ။ Numbers ထဲမှာ INT နဲ့ DECIMAL နှစ်မျိုးရှိပါတယ်။

INT - လူနာအရေအတွက် ၊ အသက် ၊ ကုတင်အရေအတွက် စသည် ကိန်းဂဏန်းတွေသိမ်းဖို့ပါ။ ဥပမာ - patient_count INT ဆိုပြီး သတ်မှတ်ထားရင် 1, 2, 3 စသဖြင့်ပဲ ထည့်လို့ရပါမယ်။

DECIMAL - ဆေးပမာဏ၊ အပူချိန်လိုမျိုး ဒဿမကိန်းတွေသိမ်းဖို့ပါ။ ဥပမာ - temperature DECIMAL(3,1) ဆိုရင် 100.5 လိုမျိုး ကိန်းပြည့်သုံးနေရာ၊ ဒဿမ တစ်နေရာသိမ်းလို့ရပါမယ်။

2.3.2 Strings

စာသား၊ စာကြောင်းတွေ သိမ်းဖို့ပါ။ VARCHAR, CHAR နဲ့ TEXT ဆိုပြီးရှိပါတယ်။

VARCHAR(n) - လူနာအမည်၊ လိပ်စာ၊ ရောဂါအမည် စတာတွေသိမ်းဖို့ပါ။ ဥပမာ - patient_name VARCHAR(50) ဆိုရင် စာလုံး ၅၀ ထိ သိမ်းလို့ရပါမယ်။ unicode နဲ့ဆိုရင် မြန်မာလိုရော အင်္ဂလိပ်လိုရော သိမ်းလို့ရမှာဖြစ်ပါတယ်။ VARCHAR ဆိုတာ variable character ဖြစ်ပါတယ်။ စာလုံးအရေအတွက်အမျိုးမျိုးအတွက်အသုံးအဝင်ပါတယ်။ သူနဲ့မတူတဲ့ CHAR ဆိုတာရှိပါသေးတယ်၊ CHAR(5) ပေးထားရင်ကျတော့ ဘာပဲသိမ်းသိမ်း ငါးလုံးအတိအကျပဲလက်ခံပါတယ်။ ဥပမာ ကားနံပါတ်၊ Zip code စသည် စာလုံးအရေအတွက်ပုံသေရှိမယ့် column တွေအတွက်ဖြစ်ပါတယ်။

TEXT - ရောဂါရာဇဝင်၊ ဆရာဝန်မှတ်ချက် စသည်ဖြင့် အရှည်ကြီး ရေးရမယ့်တန်ဖိုးအတွက်သုံးပါတယ်။

DATE and TIME

ရက်စွဲတွေ၊ အချိန်တွေ သိမ်းဖို့ပါ။ DATE က နေ့စွဲတွေသိမ်းဖို့ဖြစ်ပြီး DATETIME ကတော့ 2024-01-01 13:45:00 လိုမျိုး ရက်စွဲရော အချိန်ပါ သိမ်းချင်ရင်သုံးပါတယ်။ TIME က 13:45:00 လိုမျိုး အချိန်ပဲသိမ်းမယ်ဆိုရင် သုံးပါတယ်။

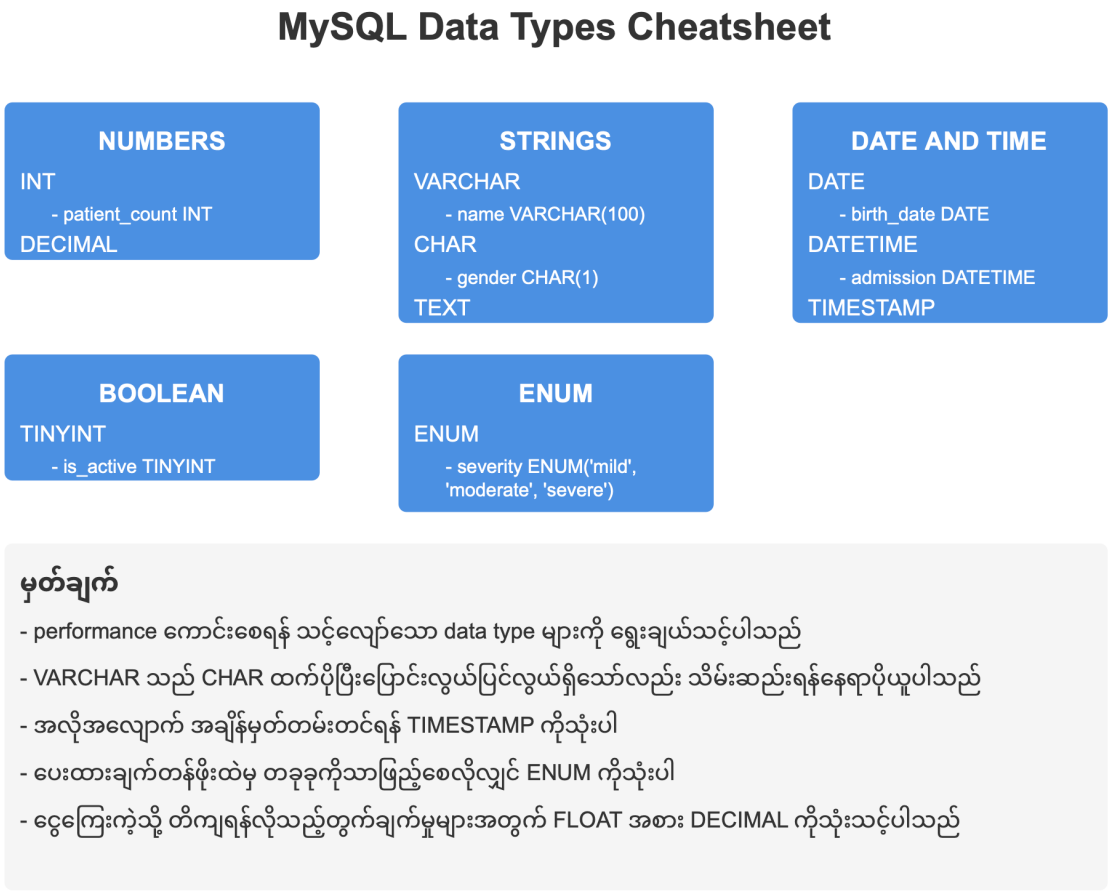
Boolean

true/ false တန်ဖိုးတွေသိမ်းဖို့ပါ။ TINYINT လို့သုံးနှုန်းပါတယ်။ ဥပမာ ဆေးဓာတ်မတည့်မှုရှိ/မရှိ၊ ရောဂါအခံရှိ/မရှိ စတာတွေကို 1 (true) နဲ့ 0 (false) တန်ဖိုးများအဖြစ် သိမ်းဆည်းပေးပါတယ်။

ENUM

ကြိုတင်သတ်မှတ်ထားတဲ့ တန်ဖိုးတွေထဲကပဲ ရွေးလို့ရပါမယ်။

```
ENUM('mild', 'moderate', 'severe') -- ရောဂါပြင်းထန်မှုအဆင့်
```



နမူနာ table တစ်ခုဖန်တီးတဲ့ command ကိုလေ့လာကြည့်ပါမယ်။

```
CREATE TABLE patients (
    id INT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100),
    age INT,
    gender ENUM('ကျား', 'မ'),
    admission_date DATE,
    temperature DECIMAL(3,1),
    severity ENUM('mild', 'moderate', 'severe'),
    doctor_notes TEXT
);
```

CREATE TABLE table_name နဲ့ table နာမည်ပေးပါမယ်။

id column က INT အမျိုးအစားသိမ်းမယ်၊ ကိုယ်ပိုင်နံပါတ် PRIMARY KEY column အဖြစ်လုပ်ဆောင်မယ်၊ ဂဏန်းကိုအလိုအလျောက်အမြဲတိုးသွားမယ် (AUTO_INCREMENT) လို့ သတ်မှတ်ပေးထားပါတယ်။

name column မှာ စာသားတွေသိမ်းမှာဖြစ်ပြီး စာလုံးရေ ၁၀၀ မကျော်အောင်ကန့်သတ်ပေးထားပါတယ်

age column မှာ ကိန်းဂဏန်းတွေသိမ်းပါမယ်

gender column မှာ "ကျား" နဲ့ "မ" ဆိုတဲ့တန်ဖိုးနှစ်ခုကလွဲလို့ တခြားတန်ဖိုးတွေလာသိမ်းလို့မရပါဘူး

admission date column မှာ ရက်စွဲတန်ဖိုးသိမ်းပါမယ်

temperature column မှာ ကိန်းပြည့် ၃ နေရာ၊ ဒဿမကိန်း ၁ နေရာပါဝင်တဲ့ ကိန်းဂဏန်းတွေသိမ်းပါမယ်

severity column မှာ "**mild**", "**moderate**", "**severe**" ဆိုတဲ့ တန်ဖိုးသုံးခုအပြင် တခြားလာသိမ်းခွင့်မပြုပါဘူး။

နောက်ဆုံးက **doctor_notes** မှာတော့ များများရေးချင်ရေးလို့ရအောင် **TEXT** လုပ်ထားပေးပါတယ်။

2.4 Primary keys and foreign keys

Primary Key ဆိုတာက **table** တစ်ခုထဲမှာ **row** တစ်ခုချင်းစီကို **unique** ဖြစ်အောင် သတ်မှတ်ပေးထားတဲ့တန်ဖိုးတွေပါ။ ဥပမာ - **patients table** စာရင်းမှာဆိုရင် **Patient ID** လိုမျိုးပေါ့။ **Primary Key** က **null value** လက်မခံပါဘူး။ တစ်ခုနဲ့တစ်ခု ထပ်လို့လည်း မရပါဘူး။

null value ဆိုတာက **SQL database** ထဲမှာ တန်ဖိုးတစ်ခုခု မရှိဘူးဆိုတာကို ပြပါတယ်။ တစ်နည်းအားဖြင့် "ဗလာနတ္ထိ" ဖြစ်နေတဲ့ အခြေအနေကို ဆိုလိုပါတယ်။ ဥပမာ လူနာတွေရဲ့ ဖုန်းနံပါတ် သိမ်းတဲ့ **column** မှာ လူနာတချို့က သူတို့ရဲ့ဖုန်းနံပါတ် မပေးထားရင် **NULL** ဖြစ်နေမှာပါ။ သွေးပေါင်ချိန်တိုင်းတဲ့အခါ တိုင်းလို့မရတဲ့ လူနာဆိုရင် **NULL value** ဖြစ်နေမှာပါ။ **NULL** နဲ့ **empty string** ("") က မတူပါဘူး။ **Empty string** ဆိုတာ တန်ဖိုးတစ်ခု ရှိပါတယ်။ ဒါပေမယ့် **blank** ဖြစ်နေတာလေးတစ်ခုပါပဲ။ **NULL** ကတော့ တန်ဖိုးလုံးဝမရှိတာပါ။ တစ်နည်းအားဖြင့် "မသိရှိသေးတဲ့" သို့မဟုတ် "မရှိတဲ့" အခြေအနေပါ။

Foreign Key ကတော့ **table** နှစ်ခုကို ဆက်သွယ်ပေးတဲ့ တန်ဖိုးပါ။ **table** တစ်ခုရဲ့ **Primary Key** ကို နောက် **table** တစ်ခုမှာ **reference** လုပ်ပြီး **Foreign Key** အနေနဲ့ သုံးတာပါ။

ဥပမာပြရရင် **disease surveillance system** တစ်ခုမှာ လူတွေနဲ့ပတ်သက်တဲ့အချက်အလက်တွေကို **patients table** မှာသိမ်းပြီး ရောဂါဖြစ်ပေါ်မှုနဲ့ပတ်သက်တာတွေကို **cases table** မှာသိမ်းတယ်ဆိုပါစို့။ **patients table** ဖန်တီးတဲ့ **command** ကအောက်ပါအတိုင်းဖြစ်ပါတယ်။

Patients Table

```
CREATE TABLE patients (
    patient_id INT PRIMARY KEY,
    name VARCHAR(100),
    date_of_birth DATE,
    township VARCHAR(50)
);
```

ဒါကတော့ **patients table** ကိုနမူနာဒေတာအချို့နဲ့မြင်ရမယ့်ပုံစံပါ။

patient_id	name	date_of_birth	township
1	Kyaw Swer	1990-05-15	Hlaing
2	Aye Aye	1995-08-22	Tamwe
3	Soe Myint	1980-12-10	Dagon
4	Khine Khine Win	1998-03-30	Thingangyun

cases နာမည်နဲ့ table တစ်ခုလည်းရှိပါမယ်။

Cases Table

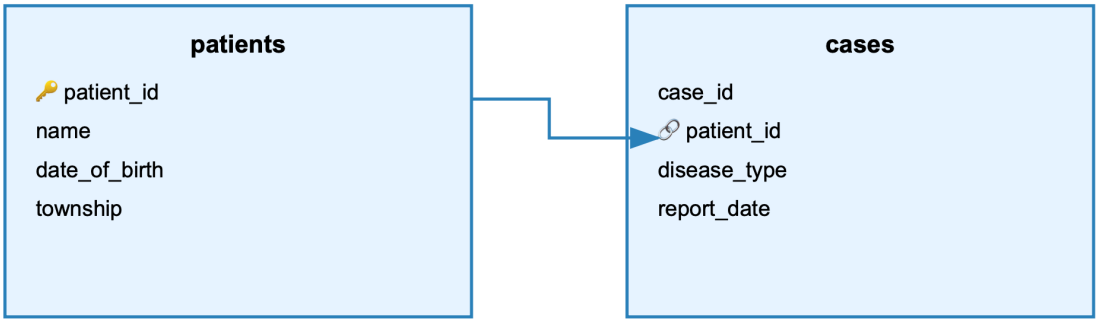
```
CREATE TABLE cases (
    case_id INT PRIMARY KEY,
    patient_id INT,
    disease_type VARCHAR(50),
    report_date DATE,
    FOREIGN KEY (patient_id) REFERENCES patients(patient_id)
);
```

ဒါကတော့ **cases table** ရဲ့နမူနာဒေတာဖြည့်ပြီးသားပုံစံပါ။

case_id	patient_id	disease_type	report_date
101	1	Common cold	2024-01-15
102	2	Diarrhoea	2024-02-01
103	1	Hypertension	2024-02-15
104	3	Diabetes	2024-03-01

* patient_id is a foreign key referencing patients(patient_id)

patients table က **patient_id primary key** နဲ့ **cases table** က **patient_id foreign key** တို့ချိတ်ဆက်ပုံကို ဒီပုံကြမ်းနဲ့မြင်ယောင်ကြည့်နိုင်ပါတယ်။



2.5 Basic SQL Queries

2.5.1 SELECT statement fundamentals

SELECT ဟာ အခြေခံအကျဆုံး SQL query ဖြစ်ပါတယ်။ SELECT ရေးထုံးက ဒီလိုပါ

```
SELECT * FROM people;
```

people table ထဲကဒေတာအားလုံးပြပါဆိုတဲ့ အဓိပ္ပါယ်ပါ။ ဒီနေရာမှာသတိထားကြည့်မယ်ဆိုရင် စာလုံးအကြီးအသေးမတူတာကိုတွေ့ရပါမယ်။ SQL query တွေဟာ case-insensitive ဖြစ်ပါတယ်။ သဘောကတော့ SELECT လို့ပဲရေးရေး၊ select ပဲရေးရေး အလုပ်လုပ်ပုံအတူတူပဲဖြစ်ပါတယ်။

query ထဲက စာလုံးအကြီးတွေကို keyword လို့ခေါ်ပါတယ်။ ဒေတာအသွင်းအထုတ် operation တွေကိုခိုင်းစေတဲ့စကားလုံးတွေကို စာလုံးအကြီး capital letter နဲ့ရေးပါတယ်။ ကျန်တာတွေအားလုံး စာလုံးအသေး small letter နဲ့ရေးပါတယ်။ table နာမည်၊ column စတာတွေကို small letter နဲ့ရေးတာပါ။ SQL query မှာ စာလုံးအကြီးနဲ့အသေး သူ့နေရာနဲ့သူရေးရမယ်လို့ အတိအကျလိုက်နာရတာမျိုးမရှိပါဘူး။ ဒါပေမယ့် ဖတ်လို့ကောင်းအောင် အကြီးနဲ့အသေးသင့်တော်သလိုရေးပေးဖို့လိုပါတယ်။

အခုလို ဒေတာတွေကို အားလုံးခေါ်တာထက် လိုတာပဲရွေးကြည့်မယ်၊ အစဉ်လိုက်စီမယ်ဆိုရင်လည်းရပါတယ်။

2.5.2 Filtering with WHERE clause

```
SELECT *
FROM people
WHERE age > 20;
```

WHERE နဲ့လိုချင်တဲ့ row တွေကိုပဲစစ်ထုတ်လို့ရပါတယ်။ ဒီမှာဆိုရင် အသက် ၂၀ အထက်ရှိတဲ့ record တွေကိုပဲ ရွေးထုတ်သွားတာပါ။

2.5.3 Sorting with ORDER BY

```
SELECT *
FROM people
```

```
WHERE age > 20
ORDER BY birthday ASC;
```

မွေးနေ့တွေကို **ascending** နဲ့ပါ အစဉ်လိုက်စီသွားတာဖြစ်ပါတယ်။ အသက် ၂၀ အထက် **record** တွေပဲရွေးထုတ်တဲ့အပြင် ရလာတဲ့ **result** ကို မွေးနေ့အလိုက်စဉ်သွားတဲ့သဘောပါ။

SQL query အလွတ်မှတ်ရန်

SQL query များအားလုံး ဤအစီအစဉ်အတိုင်းရေးရပါတယ်
အစီအစဉ်အထက်အောက်လွဲရင် အလုပ်မလုပ်ပါ

```
SELECT      -- ပြသရန် column များ
FROM        -- ဆွဲထုတ်ရန် table(s)
WHERE       -- row များကို စစ်ထုတ်ရန်
GROUP BY    -- row များကို အုပ်စုခွဲရန်
HAVING      -- အုပ်စုခွဲထားသော row များကို စစ်ထုတ်ရန်
ORDER BY    -- column များကို စီရန်
```

GROUP BY နဲ့ **HAVING** တို့ကို နောက်အခန်းတွေမှာအသေးစိတ်ဖော်ပြသွားပါမယ်။ လောလောဆယ် **SQL query** ရေးထုံးကိုအလွတ်ရအောင် မှတ်ထားဖို့လိုပါတယ်။ **SELECT → FROM → WHERE → GROUP BY → HAVING → ORDER BY** အစဉ်လိုက်အတိုင်းသွားဖို့လိုပါတယ်။ **query** ရေးတာအားလုံးမှန်ပြီး **GROUP BY** က **WHERE** ထက်အရင်ရေးမိတာမျိုးဆိုရင် အလုပ်မလုပ်ပါဘူး။

2.5.4 Basic operators (=, <, >, BETWEEN, LIKE)

Operator ဆိုတာ ဒေတာဘေ့စ်ထဲက ဒေတာတွေကို **filter** လုပ်တဲ့အခါ အသုံးပြုတဲ့ **operator** တွေပဲဖြစ်ပါတယ်။

ညီမျှခြင်း (=) operator

တန်ဖိုးနှစ်ခုတူညီမှုကိုစစ်ဆေးဖို့အတွက် အသုံးပြုပါတယ်။ ဥပမာ - လူနာတွေထဲက အသက် 25 နှစ်ရှိတဲ့သူတွေကိုပဲ ရွေးထုတ်ချင်ရင်

```
SELECT * FROM patients WHERE age = 25;
```

ကိန်းဂဏန်းတွေမဟုတ်ဘဲ **string** တန်ဖိုးတွေကိုစစ်ဆေးချင်ရင် **single quote** ' (သို့) **double quote** " နဲ့ရေးရပါတယ်။ **=** **operator** မှာ **case-sensitive** မဖြစ်ပါဘူး။ သဘောကတော့ **WHERE disease = 'Malaria'** နဲ့ **WHERE disease = 'MALARIA'** ကအလုပ်လုပ်ပုံချင်းအတူတူပါပဲ။ **=** **operator** သုံးတဲ့အခါ သတိပြုရမှာက **NULL** တန်ဖိုးစစ်ချင်ရင် **= NULL** နဲ့အလုပ်မလုပ်ပါဘူး။ **IS NULL** နဲ့စစ်မှရပါတယ်။

= operator ကို WHERE clause ထဲမှာအဓိကသုံးပေမယ့် ရှေ့ဆက်ဖော်ပြမယ့် table JOIN တွေ၊ UPDATE statement တွေမှာလည်း သုံးလို့ရပါတယ်။

သေးခြင်း/ကြီးခြင်း (<, >) operator

ဘယ်ဘက်ကတန်ဖိုးကို ညာဘက်ကတန်ဖိုးနဲ့နှိုင်းယှဉ်ပြီး အလုပ်ဆက်လုပ်ပါတယ်။ ဥပမာ WHERE temperature < 37.5 ။ ကိန်းဂဏန်း၊ ရက်စွဲတွေနှိုင်းယှဉ်တဲ့အခါ အသုံးများပါတယ်။

နှစ်ခုကြား (BETWEEN) operator

ဥပမာ - အသက် 20 နဲ့ 30 ကြား လူနာတွေကိုပဲ ရွေးထုတ်ချင်ရင်

```
SELECT * FROM patients
WHERE age BETWEEN 20 AND 30;
```

Between operator က အစတန်ဖိုးနဲ့ အဆုံးတန်ဖိုးနှစ်ခုစလုံးကို ထည့်သွင်းပြီး (inclusive) တွက်ချက်ပါတယ်။ ကိန်းဂဏန်းတွေ၊ ရက်စွဲတွေမှာ အဓိကအသုံးပြုပါတယ်။ Between operator သုံးတိုင်းမှာ အမြဲတမ်း AND ခံပေးရပါတယ်။ သတိထားရမှာတစ်ခုက တန်ဖိုးတွေကို AND နဲ့စီတဲ့အခါ ငယ်စဉ်ကြီးလိုက်စီရပါမယ်။ BETWEEN 30 AND 20 လို့ရေးရင်အလုပ်မလုပ်လို့ပါ။

စာလုံးရှာဖွေခြင်း (LIKE) operator

LIKE operator က search နဲ့ရှာသလိုပါပဲ။ စာလုံးတစ်လုံး (သို့) အများကိုရှာနိုင်ပါတယ်။ LIKE operator မှာ အဓိကသုံးတဲ့ wildcard က % ဖြစ်ပါတယ်။ ဥပမာ "တုပ်ကွေး" စာလုံးပါတဲ့ ရောဂါနာမည်တွေကိုရှာချင်ရင်

```
-- နာမည်မှာ "တုပ်ကွေး" ပါတာမှန်သမျှရှာ
SELECT disease_name FROM diseases WHERE disease_name LIKE '%တုပ်ကွေး%';

-- "တုပ်ကွေး"နဲ့စတဲ့နာမည်တွေရှာ
SELECT disease_name FROM diseases WHERE disease_name LIKE 'တုပ်ကွေး%';

-- တုပ်ကွေးနဲ့ဆုံးတဲ့နာမည်တွေရှာ
SELECT disease_name FROM diseases WHERE disease_name LIKE '%တုပ်ကွေး';
```

LIKE operator သုံးတဲ့အခါ Index စနစ်ကိုအပြည့်အဝအသုံးမချနိုင်တဲ့အတွက် data အများကြီး query လုပ်တဲ့အခါ performance ကျတတ်ပါတယ်။

CHAPTER 3
INTERMEDIATE CONCEPTS

3 Chapter 3 -Intermediate Concepts

ဒီအပိုင်းမှာ **table** တွေဘယ်လိုဖန်တီးမလဲ၊ ဘယ်လိုပြုပြင်မလဲ၊ **relationship** ဘယ်လိုချိတ်ဆက်မလဲ စတာတွေဖော်ပြထားပါတယ်။

3.1 Table Operations

3.1.1 CREATE TABLE

Table တစ်ခုကို အောက်ပါပုံစံအတိုင်း တည်ဆောက်နိုင်ပါတယ်:

```
CREATE TABLE table_name (
    column1 datatype1 [constraints],
    column2 datatype2 [constraints],
    ...
    [table_constraints]
);
```

ဥပမာအားဖြင့် ကျန်းမာရေးစောင့်ရှောက်မှုမှတ်တမ်းတွေကို သိမ်းဆည်းထားတဲ့ **electronic medical record database** တစ်ခုမှာ လူတွေရဲ့ ပုဂ္ဂိုလ်ရေးအချက်အလက်တွေသိမ်းပေးတဲ့ **patients table** ကို ပြုလုပ်တဲ့ **command** ကိုလေ့လာကြည့်နိုင်ပါတယ်။

```
CREATE TABLE patients (
    patient_id INT PRIMARY KEY AUTO_INCREMENT,
    registration_no VARCHAR(20) UNIQUE NOT NULL,
    full_name VARCHAR(100) NOT NULL,
    date_of_birth DATE NOT NULL,
    gender ENUM('M', 'F', 'Other') NOT NULL,
    address TEXT,
    phone_number VARCHAR(20),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Naming tables

CREATE TABLE command နောက်မှာ **table** နာမည်သတ်မှတ်ပေးရပါမယ်။ **table** နာမည်ပေးတဲ့အခါ ကိုယ်တစ်ယောက်တည်းနားလည်မယ့် နာမည်မျိုးမပေးသင့်ပါဘူး။ **database** ကိုသုံးမယ့်လူတိုင်း နားလည်လွယ်မည့် နာမည်မျိုးပေးသင့်ပါတယ်။ ကိုယ့်ဘာသာကိုယ်တောင် ဒီ **table** နာမည်က ဘာတွေသိမ်းတဲ့ **table** မှန်းအခုနေသိချင်သိပေမယ့် လနည်းနည်းအကြာမှာ ပြန်ကြည့်ရင်မှတ်မိချင်မှမှတ်မိတော့မှာပါ။

SQL မှာ **table** နာမည်၊ **column** နာမည်တွေပေးတဲ့အခါ **lower case snake case** သုံးပါတယ်။ ဥပမာ **lab_results** ။ **table** နာမည်တွေကို နာမ် **(nouns)** တွေသုံးပြီးတော့ပဲ ပေးပါတယ်။ ကြိယာ **(verb)**၊ နာမဝိသေသန **(Adjectives)** တွေမပေးရပါဘူး။ နာမ် **(nouns)** တွေသုံးပြီး နာမည်ပေးတဲ့အခါမှာလည်း အများကိန်း **(plural)** နဲ့ပေးရပါတယ်။ ဥပမာ **patients, prescriptions, doctors** ။ **table** နာမည်ပေးတဲ့အခါမှာ အတိုကောက်တွေရှောင်ပါ။ တတ်နိုင်သလောက် နာမည်အပြည့်အစုံပေးပါ။ **query** တွေရေးတဲ့အခါမှာ နာမည်ရှည်တဲ့ **table** တွေကို အတိုကောက်ပေးလို့ရပါတယ်။ ဥပမာ **lab_results lr** ။

Naming Columns

Column နာမည်တွေပေးတဲ့အခါမှာ နာမ် **(nouns)** နဲ့ နာမဝိသေသန **(Adjectives)** တွေပဲပေးသင့်ပါတယ်။ ကြိယာ **(verb)** မသုံးသင့်ပါဘူး။ **column** နာမည်မှာ သူ့ရဲ့သဘောသဘာဝကို ရှင်းရှင်းလင်းလင်းဖော်ပြပေးရပါမယ်။ ဥပမာ ကိုယ်ဝန်ဆောင်ဟုတ်မဟုတ်မှတ်ထားပေးတဲ့

column ကို pregnancy လို့ပေးထားတဲ့ is_pregnant လို့ပေးတာက ပိုသင့်တော်ပါတယ်။ Column နာမည်ပေးတဲ့အခါမှာ table နာမည်နဲ့ထပ်အောင်မပေးသင့်ပါဘူး။

Column data types

Column တွေအတွက် Data type ရွေးချယ်တဲ့အခါမှာ စာလုံးရေ (၂၅၅) လုံးအောက်ပဲ သိမ်းမယ့်စာသားတွေအတွက် VARCHAR သုံးပါ။ ဒီထက်ရှည်နိုင်ရင် TEXT နဲ့သိမ်းပါ။ ရက်စွဲတွေကို VARCHAR column နဲ့မသိမ်းဘဲ DATE, DATETIME, TIMESTAMP column တွေသုံးပြီးသိမ်းသင့်ပါတယ်။ ဂဏန်းတွေအတွက်ကိုတော့ INT, DECIMAL အမျိုးအစား column တွေနဲ့သိမ်းသင့်ပါတယ်။

NULL ဖြစ်လို့မရတဲ့ column တွေကို NOT NULL constraint မမေ့မလျော့ထည့်ပေးရပါမယ်။ NOT NULL constraint ထည့်ဖို့မေ့ခဲ့တဲ့ database တစ်ခုဟာ တကယ်သုံးမယ့် production environment အဆင့်ရောက်သွားပြီ၊ data တွေလည်းတကယ်ဖြည့်နေကြပြီဆိုရင် နောက်မှ NOT NULL constraint လိုက်ထည့်ဖို့မလွယ်တော့ပါဘူး။ ဘာလို့လဲဆိုတော့ တချို့ record တွေက NULL အနေနဲ့ထည့်ပြီးသားဖြစ်နေလို့ပါ။

ENUM အမျိုးအစား column သုံးမယ်ဆိုရင် ရွေးချယ်ခွင့်ပေးတဲ့ option တွေကို သေချာကြိုတင်စဉ်းစားပေးရပါမယ်။ "Male, Female, Others" လိုမျိုး option ခဏခဏအတိုးအလျော့မရှိနိုင်တာမျိုးဆို Enum ပေးရတာပြဿနာမရှိပေမယ့် ရောဂါနာမည်တွေသိမ်းဖို့ disease column မှာ Enum နဲ့ပေးရွေးလိုက်လို့အဆင်မပြေပါဘူး။ ရောဂါနာမည်စာလုံးပေါင်းပြင်တာ၊ ရောဂါနာမည်အသစ်တစ်ခုတိုးတာတွေက ခဏခဏလုပ်ရတတ်တာကြောင့် table အသစ်ဆောက်ပြီး foreign key နဲ့လာချိတ်လိုက်တာက ရေရှည်မှာသက်သာစေမှာပါတယ်။

True or false နှစ်မျိုးပဲရှိနိုင်တဲ့ column တွေ ဥပမာ - is_active ၊ has_travelled စတာတွေမှာ default တန်ဖိုးသတ်မှတ်ပေးပါ။ ဥပမာ is_active column မှာ သီးသန့်သွားပြီး false မလုပ်ပေးသ၍ true လို့ အလိုအလျောက်သိမ်းအောင်လုပ်ပေးရပါမယ်။ has_travelled column မှာလည်း default အားဖြင့် false ပေးထားလို့ရပါတယ်။ MSQL မှာတော့ true/false တန်ဖိုးတွေသိမ်းဖို့ boolean အမျိုးအစား data type မရှိပါဘူး။ backend programming language တွေမှာပဲ boolean ရှိပါတယ်။ MySQL မှာတော့ ဒီရည်ရွယ်ချက်အတွက် TINYINT အမျိုးအစားရှိပါတယ်။ TINYINT 1 ဆိုရင် true ကိုကိုယ်စားပြုပြီး 0 ဆိုရင် false ပါ။

ဥပမာ လူနာတစ်ယောက်မှာ ဆီးချိုရောဂါရှိ/မရှိ မှတ်တမ်းတင်ချင်တယ်ဆိုပါစို့။

```
CREATE TABLE patients (
    patient_id INT PRIMARY KEY AUTO_INCREMENT,
    full_name VARCHAR(100),
    has_diabetes TINYINT(1) DEFAULT 0,
    is_hypertensive TINYINT(1) DEFAULT 0
);
```

DEFAULT 0 ပေးထားတာဟာ သီးသန့် true သွားမပေးသ၍ အလိုအလျောက် false နဲ့သိမ်းပေးဖို့ဖြစ်ပါတယ်။ VARCHAR နဲ့ true ၊ false စာသားတွေသွားသိမ်းတာထက် TINYINT နဲ့ 1,0 ပဲသိမ်းတာက performance ပိုကောင်းစေပါတယ်။ MySQL workbench သုံးတဲ့အခါမှာလည်း checkbox နဲ့ပြပေးပါတယ်။

Data validation

အကြမ်းအားဖြင့် Data validation လုပ်လို့ရတာ သုံးနေရာရှိပါတယ်။ Database ၊ Backend နဲ့ Frontend ပါ။

MySQL table ထဲကို record တစ်ခုလာ save တဲ့အခါ NOT NULL column ကို ဒေတာပေးဖို့ကျန်ခဲ့ရင် error code 1048 နဲ့ပြပြီး validate လုပ်ပေးပါတယ်။

Backend programming ပိုင်းဆက်ရေးတဲ့အခါမှာတော့ MySQL error code တွေကို front end ဘက်မှာ နားလည်လွယ်တဲ့ error message တွေအဖြစ်ပြောင်းရေးပေးရသလို backend programming ပိုင်းမှာ CRUD operation တွေအတွက် data validation rules တွေသတ်မှတ်ပေးရပါတယ်။ NOT NULL column အတွက် data မပါလာရင် backend ဘက်ကနေလည်း error ပစ်လို့ရပါတယ်။

Frontend ကတော့ user နဲ့တိုက်ရိုက်ထိတွေ့ရတဲ့အပိုင်းဖြစ်ပါတယ်။ backend နဲ့ database မှာ data validation rule တွေမပါရင်တောင်မှ frontend web form တွေမှာ required field validation တွေထည့်ပေးထားလို့ရပါတယ်။

3.1.2 Security considerations

Database encryption
MySQL enterprise edition မှာ file level encryption ပါလို့ Database file တစ်ခုလုံးကို encrypt လုပ်လို့ရပါတယ်။ MySQL server ကိုဖွင့်လိုက်တာနဲ့ encryption key ထည့်ပေးမှဖွင့်လို့ရတာမို့လို့ database file တစ်ခုလုံးသူများလက်ထဲရောက်သွားရင်တောင် encryption key မသိရင်ဖွင့်လို့မရပါဘူး။

Column encryption
users table ထဲက password column လိုမျိုး sensitive ဖြစ်တဲ့ဒေတာတွေကို encrypt လုပ်ပေးဖို့လိုပါတယ်။ password column ကို plain text VARCHAR နဲ့သာသိမ်းထားရင် database admin ကလူတိုင်းရဲ့password တွေကိုမြင်နေရမှာမို့လို့ပါ။ MySQL column ကို VARBINARY(255) နဲ့ encrypt လုပ်လို့ရပါတယ်။ လက်တွေ့မှာတော့ database column ကို encrypt လုပ်တာထက် backend programming မှာပဲ encrypt လုပ်ပြီး hash လုပ်ပြီးသား password ကိုမှ database ထဲမှာ VARCHAR အနေနဲ့သိမ်းကြတာများပါတယ်။ Programming language ရဲ့ hash algorithm သုံးပြီးသိမ်းထားတဲ့ password တွေက နဂို password ကိုပြန်မသိနိုင်တဲ့ password ဖြစ်သွားပါတယ်။ Login ဝင်တဲ့အခါ user ရိုက်ထည့်လိုက်တဲ့ password ကို hash တွက်ပြီး database ထဲက hashed လုပ်ထားတဲ့ တန်ဖိုးနဲ့တူမတူတိုက်စစ်ပါတယ်။ ဒါကြောင့် password ကိုစသိမ်းသိမ်းချင်းမှာ သုံးထားတဲ့ hash algorithm နဲ့ login ဝင်တဲ့အခါမှာ user ရိုက်ထည့်လိုက်တဲ့ password ကို hash လုပ်မယ့် algorithm နဲ့ programming language အတူတူပဲဖြစ်သင့်ပါတယ်။ column level encryption ကို PHP မှာ password_hash() ၊ Python မှာ bcrypt library တို့နဲ့ပဲ ပြုလုပ်ကြပြီး လက်တွေ့မှာ MySQL column encryption ဖြစ်တဲ့ VARBINARY အမျိုးအစားကို သိပ်မသုံးဖြစ်ကြပါဘူး။

3.1.3 ALTER TABLE operations

Table တစ်ခုဖန်တီးပြီးသွားတဲ့နောက်မှာ table ကို ပြင်ဆင်လိုတာတွေရှိလာနိုင်ပါတယ်။ Column အသစ်ထပ်ထည့်ချင်တာမျိုး၊ ရှိပြီးသား column ကို ပြင်ဆင်ချင်တာမျိုး၊ column တွေဖျက်ချင်တာမျိုးရှိလာနိုင်ပါတယ်။ ဒီလိုအခြေအနေမျိုးတွေအတွက် ALTER TABLE command ကို အသုံးပြုပါတယ်။

Adding new column

```
ALTER TABLE patients
ADD COLUMN blood_type VARCHAR(5);
```

ဒါကတော့ patients table ထဲကို blood_type ဆိုတဲ့ column အသစ်တစ်ခု ထပ်ထည့်လိုက်တာပါ။ VARCHAR(5) အမျိုးအစားဖြစ်ပြီး စာလုံးအရေအတွက် ၅ လုံးအထိ သိမ်းလို့ရမှာဖြစ်ပါတယ်။ Optional field ဖြစ်တဲ့အတွက် NOT NULL constraint မပါဝင်ပါဘူး။

မူလ table ထဲမှာ record တွေရှိပြီးသားဆိုရင် ထပ်တိုးလိုက်တဲ့ column က NULL တန်ဖိုးတွေနဲ့ စတင်ပါလိမ့်မယ်။ Default value သတ်မှတ်ပေးထားချင်ရင် အောက်ပါအတိုင်း ရေးနိုင်ပါတယ်။

```
ALTER TABLE patients
ADD COLUMN has_insurance TINYINT DEFAULT 0;
```

Modifying columns
Column တစ်ခုရဲ့ data type ကို ပြင်ဆင်ချင်တယ် ဆိုပါစို့။

```
ALTER TABLE patients
MODIFY COLUMN phone_number VARCHAR(30);
```

ဒါဆိုရင် **phone_number column** က စာလုံး ၃၀ အထိ သိမ်းလို့ရအောင် ပြင်ပေးလိုက်တာ ဖြစ်ပါတယ်။

Column နာမည်ပြောင်းချင်တယ်ဆိုရင်တော့

```
ALTER TABLE patients
RENAME COLUMN phone_number TO contact_number;
```

Deleting columns

Column တစ်ခုကို ဖျက်ချင်ရင်တော့ **DROP COLUMN** သုံးပါတယ်။

```
ALTER TABLE patients
DROP COLUMN blood_type;
```

Editing constraints

Column တစ်ခုမှာ အရင်က **optional** ဖြစ်နေတာကို မဖြစ်မနေထည့်ပေးရမယ့် **NOT NULL** ပြင်ချင်တယ်ဆိုပါစို့။

```
ALTER TABLE patients
MODIFY COLUMN email VARCHAR(100) NOT NULL;
```

Primary key သတ်မှတ်ချင်ရင်

```
ALTER TABLE patients
ADD PRIMARY KEY (patient_id);
```

Foreign key ထပ်ထည့်ချင်ရင်

```
ALTER TABLE visits
ADD FOREIGN KEY (patient_id) REFERENCES patients(patient_id);
```

Table နာမည်ပြောင်းခြင်း

Table နာမည်ကိုပါ ပြောင်းချင်တယ်ဆိုရင်တော့ **RENAME** သုံးပါတယ်။

```
RENAME TABLE patients TO hospital_patients;
```

ALTER TABLE command တွေကို အသုံးပြုတဲ့အခါ **Column** တစ်ခုကို ဖျက်လိုက်တယ်ဆိုရင် **column** ထဲက **record** အားလုံးရဲ့ဒေတာတွေပါပျက်သွားတာပါ။ ပြန်ပြင်လို့မရတော့ပါဘူး။

Data type ပြောင်းတဲ့အခါ ရှိပြီးသား **data** တွေနဲ့ ကိုက်ညီရပါမယ်။ ဥပမာ **INT** ကနေ **VARCHAR** ပြောင်းတာမျိုးက အဆင်ပြေနိုင်ပေမယ့် **VARCHAR** ကနေ **INT** ပြောင်းတာမျိုးက အဆင်မပြေနိုင်ပါဘူး။

လက်တွေ့သုံးနေပြီဖြစ်တဲ့ **Production database** တွေမှာ **ALTER TABLE** လုပ်တဲ့အခါ **table lock** လုပ်ထားတဲ့အချိန်အတွင်း တခြား **operation** တွေလုပ်လို့မရတာမို့လို့ **user** အသုံးနည်းတဲ့အချိန်မှာ လုပ်သင့်ပါတယ်။

3.1.4 DROP and TRUNCATE operations

Database ထဲမှာရှိတဲ့ **table** တွေကို လုံးဝဖျက်ချင်တဲ့အခါ **DROP** နဲ့ **TRUNCATE** ဆိုပြီး နည်းလမ်းနှစ်မျိုးရှိပါတယ်။

DROP TABLE

DROP က **table** တစ်ခုလုံးကို **structure** အပါအဝင် အကုန်လုံးဖျက်ပစ်လိုက်တာပါ။ ဒီလိုရေးပါတယ်။

```
DROP TABLE patients;
```

DROP လုပ်လိုက်ရင် table structure ရော၊ အထဲက data တွေရော၊ index တွေရော အကုန်လုံးပျက်သွားပါတယ်။ ဒါကြောင့် DROP မလုပ်ခင် သေချာစဉ်းစားဖို့လိုပါတယ်။ အမှတ်တမဲ့ DROP လုပ်မိရင် table ပြန်တည်ဆောက်တာကအစ အကုန်ပြန်လုပ်ရပါတယ်။ table တစ်ခုကို DROP လုပ်လိုက်တာနဲ့ disk space ပြန်ရပါတယ်။

DROP မလုပ်ခင် အဲဒီ table က တခြား table တွေနဲ့ foreign key နဲ့ချိတ်ထားသလားဆိုတာစစ်ဖို့လိုပါတယ်။ ဥပမာ - patients table ကို DROP လုပ်လိုက်ရင် visits table က patient_id နဲ့ချိတ်ထားတဲ့ record တွေကျန်နေမှာဖြစ်ပါတယ်။ ဒါကြောင့် patients table ကို DROP မလုပ်ခင် visits table ထဲမှာ အဲဒီ patient_id နဲ့ချိတ်ထားတဲ့ record တွေကိုပါ ဖျက်ပေးဖို့လိုပါတယ်။

TRUNCATE TABLE

TRUNCATE ကတော့ table ထဲက data တွေပဲဖျက်ပြီး table structure ကိုတော့ ဆက်ထားပေးတာပါ။ ဒီလိုရေးပါတယ်။

```
TRUNCATE TABLE patients;
```

TRUNCATE လုပ်လိုက်ရင် table ထဲက data တွေပဲပျက်သွားပြီး table structure နဲ့ column တွေကတော့ ဆက်ရှိနေပါတယ်။ ဒါကြောင့် နောက်ပိုင်း data တွေထပ်ထည့်လို့ရပါတယ်။ TRUNCATE လုပ်လိုက်တာနဲ့ auto_increment တန်ဖိုးတွေလည်း reset ပြန်ဖြစ်သွားပါတယ်။ disk space လည်းပြန်ရပါတယ်။

TRUNCATE လုပ်တဲ့အခါမှာလည်း foreign key constraint တွေကို သတိထားရပါတယ်။ ဥပမာ patients table ကို TRUNCATE လုပ်မယ်ဆိုရင် visits table ထဲမှာ အဲဒီ patient_id တွေသုံးထားတဲ့ record တွေရှိနေရင် error ပြပါလိမ့်မယ်။

DROP နဲ့ TRUNCATE နှစ်ခုလုံးက table ကိုဖျက်တာပဲဆိုပေမယ့် table structure ကိုဆက်လက်အသုံးပြုမယ်ဆိုရင် TRUNCATE သုံးသင့်ပြီး table ကိုပါ ဖျက်ပစ်ချင်တယ်ဆိုရင် DROP သုံးသင့်ပါတယ်။ TRUNCATE နဲ့ DROP နှစ်ခုလုံးက ပြန်မရနိုင်တဲ့ operation တွေဖြစ်တဲ့အတွက် မလုပ်ခင်သေချာစဉ်းစားဖို့လိုပါတယ်။

3.2 Data Manipulation

3.2.1 INSERT statements

table ထဲကို ဒေတာအသစ်ထည့်သွင်းဖို့သုံးပါတယ်။ INSERT ရေးထုံး နှစ်ခုရှိပါတယ်။ တစ်ကြောင်းချင်းသွင်းတာနဲ့ အများကြီးသွင်းတာနဲ့ပါ။ ရေးထုံးချင်းသိပ်တော့မကွာပါဘူး။ INSERT မလုပ်ခင် table တစ်ခုအရင် create လုပ်ကြည့်ပါမယ်။

```
CREATE TABLE doctors (
    doctor_id INT PRIMARY KEY AUTO_INCREMENT,
    full_name VARCHAR(100) NOT NULL,
    specialty VARCHAR(100) NOT NULL,
    license_number VARCHAR(50) UNIQUE NOT NULL,
    phone_number VARCHAR(20),
    email VARCHAR(100),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

တစ်ကြောင်းပဲ INSERT လုပ်မယ်ဆိုရင်

```
INSERT INTO doctors (full_name, specialty, license_number, phone_number, email)
VALUES
('Dr. Thiha', 'Cardiologist', 'LIC001', '09111222333', 'dr.thiha@email.com');
```

တစ်ကြောင်းထက်ပို INSERT လုပ်ရင်

```
INSERT INTO doctors (full_name, specialty, license_number, phone_number, email)
VALUES
('Dr. Aye Aye', 'Pediatrician', 'LIC002', '09222333444', 'dr.ayeaye@email.com'),
('Dr. Kyaw Min', 'General Physician', 'LIC003', '09333444555', 'dr.kyawmin@email.com'),
('Dr. Su Su', 'Dermatologist', 'LIC004', '09444555666', 'dr.susu@email.com');
```

doctor_id က **auto_increment** ဖြစ်တဲ့အတွက် ထည့်ပေးစရာမလိုပါဘူး။ **created_at column** က **default value** သတ်မှတ်ထားလို့ အလိုအလျောက်ထည့်ပေးသွားမှာဖြစ်ပါတယ်။

phone_number နဲ့ **email column** တွေက **optional** ဖြစ်လို့ မထည့်လည်းရပါတယ်။ ဒါပေမယ့် **full_name, specialty** နဲ့ **license_number** တွေကတော့ **NOT NULL constraint** သတ်မှတ်ထားလို့ မဖြစ်မနေထည့်ပေးရမှာဖြစ်ပါတယ်။

license_number မှာ **UNIQUE constraint** သတ်မှတ်ထားတဲ့အတွက် တစ်ယောက်နဲ့တစ်ယောက် လိုင်စင်နံပါတ် ထပ်လို့မရပါဘူး။ ထပ်နေရင် **error** ပြပါလိမ့်မယ်။

3.2.2 UPDATE operations

UPDATE statement က ဒေတာတွေကိုပြင်ဆင်ဖို့သုံးပါတယ်။ ရှိပြီးသားဒေတာမှာအပြောင်းအလဲဖြစ်တဲ့အခါ **UPDATE** နဲ့ပြင်ဆင်နိုင်ပါတယ်။

Before UPDATE

doctor_id	full_name	specialty	phone_number	email
1	Dr. Thiha	Cardiologist	09111222333	dr.thiha@email.com

After UPDATE

doctor_id	full_name	specialty	phone_number	email
1	Dr. Thiha	Cardiologist	09999888777	thiha@newemail.com

ပုံမှာပြထားတဲ့အတိုင်း ဖုန်းနံပါတ်နဲ့ အီးမေးလ်လိပ်စာပြင်ချင်တဲ့အခါ ဒီလိုရေးလို့ရပါတယ်။

```
UPDATE doctors
SET
    phone_number = '09999888777',
    email = 'thiha@newemail.com'
WHERE doctor_id = 1;
```

UPDATE ရဲ့နောက်မှာ **table** နာမည်ရေးရပါတယ်။ **SET** ရဲ့နောက်မှာပြောင်းချင်တဲ့ **column** နာမည်တွေထည့်ပြီး ပြင်ချင်တဲ့တန်ဖိုးကို **assign** လုပ်ရပါတယ်။ **WHERE** ကတော့ ဘယ် **record** ကိုပြင်မယ်ဆိုတာ **filter** လုပ်ပေးပါတယ်။ **WHERE** နဲ့သာမခံရင် ရှိတဲ့ **record** အားလုံးကို **update** လုပ်သွားမှာပါ။ **UPDATE** လုပ်တဲ့အခါ **WHERE clause** ကိုသတိတရ ထည့်ရေးပေးဖို့လိုပါတယ်။

WHERE နဲ့ filter လုပ်တဲ့အခါ record တစ်ခုတည်းကိုပဲထောက်လို့ရသလို record အများကြီးကိုလည်း filter ထောက်လို့ရပါတယ်။ ဥပမာ အထွေထွေရောဂါကုဆရာဝန်အားလုံးရဲ့အထူးပြုဘာသာရပ်ကို "Family Medicine" လို့ပြောင်းလို့ရပါတယ်။

```
UPDATE doctors
SET specialty = 'Family Medicine'
WHERE specialty = 'General Physician';
```

Update မလုပ်ခင် အရင်ဆုံး SELECT WHERE နဲ့အရင်ခေါ် ကြည့်တာက ပိုစိတ်ချရပါတယ်။

```
SELECT * FROM doctors WHERE specialty = 'General Physician';
```

ဒီလိုစစ်ကြည့်ပြီးမှန်တယ်ဆိုမှ update ဆက်လုပ်သင့်ပါတယ်။ မဟုတ်ရင်မဆိုင်တဲ့ဒေတာတွေ ပြင်မိတတ်လို့ပါ။
အပြင်မှာတကယ်သုံးမယ်ဆိုရင်တော့ ဒီအဆင့်က CRUD (Create, Read, Update, Delete) operation ထဲက Read အဆင့်မှာပါပြီးသားပါ။ လက်တွေ့ application တွေမှာတော့ ဒေတာကို read mode နဲ့ပြ၊ ပြင်ချင်ရင် edit ခလုတ်နှိပ်ပြီးပြင်ရတာပါ။

UPDATE လုပ်တဲ့အခါမှာလည်း INSERT တုန်းကိုလိုပဲ Constraint တွေကိုလိုက်နာပါတယ်။ NOT NULL column မှာ NULL တန်ဖိုးဖြစ်အောင် update လုပ်လို့မရပါဘူး။ UNIQUE ပေးထားတဲ့ license_number column မှာလည်း Update လုပ်ရင် ရှိပြီးသားတန်ဖိုးတစ်ခုခုနဲ့တူအောင် ပေးလို့မရပါဘူး။

3.2.3 DELETE operations

DELETE statement က table ထဲက record တွေကိုဖျက်တဲ့အခါသုံးပါတယ်။ DELETE လုပ်တဲ့အခါ သတိထားရမှာက ဖျက်ပြီးရင်ပန်ရနိုင်တဲ့နည်းမရှိပါဘူး။ ဒါကြောင့် undo ပြန်လုပ်ချင်တဲ့အခါမျိုးမှာ transaction နဲ့လုပ်တာက ပိုစိတ်ချရပါတယ်။

doctors table ထဲက Dr. Aye Aye ရဲ့ record ကိုဖျက်ချင်တယ်ဆိုရင် ဒီလိုရေးလို့ရပါတယ်။

```
DELETE FROM doctors
WHERE doctor_id = 2;
```

DELETE FROM နောက်မှာ table နာမည်သတ်မှတ်ပေးရပါတယ်။ WHERE ကတော့ table ထဲက ဘယ် record ကိုဖျက်မယ်ဆိုတာ filter လုပ်ပေးရပါတယ်။ UPDATE တုန်းကလိုပဲ WHERE နဲ့သေချာ filter မလုပ်ထားရင် table ထဲက record အားလုံးပျက်သွားမှာဖြစ်ပါတယ်။ ဒါကြောင့် DELETE လုပ်တဲ့အခါမှာလည်း WHERE clause ကိုသေချာရေးပေးဖို့လိုပါတယ်။

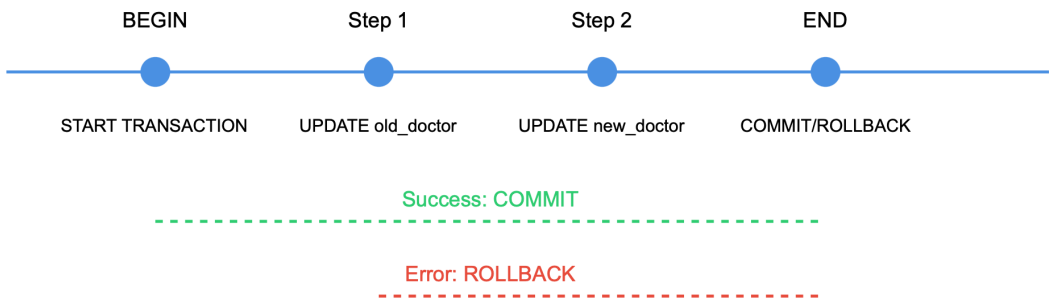
doctors table ထဲက ဆရာဝန်တစ်ယောက်ရဲ့ record ကိုဖျက်လိုက်တဲ့အခါ သူနဲ့ဆက်စပ်နေတဲ့ တခြား table တွေကိုပါ စစ်ဆေးပေးရပါမယ်။ ဥပမာ visits table မှာ foreign key အနေနဲ့ doctor_id ပါနေတဲ့အတွက် doctor_id ကိုဖျက်လိုက်ရင် visits table မှာ အဲဒီ doctor_id ပါတဲ့ record တွေကို ဘယ်လိုဆက်လုပ်မယ်ဆို စီစဉ်ပေးဖို့လိုပါတယ်။

DELETE နဲ့ဆင်တူတာက TRUNCATE ပါ။

```
TRUNCATE TABLE doctors;
```

TRUCATE က table ထဲက ဒေတာအားလုံးကိုတချက်တည်းတန်းဖျက်တာပါ။ DELETE ကတော့ filter ထောက်ပြီးဖျက်ချင်တဲ့ record ကိုရွေးဖျက်လို့ရပါတယ်။ TRUNCATE နဲ့ဖျက်ရင် auto_increment တန်ဖိုးတွေကို 0 ကနေ reset ပြန်စပေးပါတယ်။ DELETE ကတော့ လက်ရှိရောက်လက်စ auto_increment တန်ဖိုးကနေဆက်သွားပါတယ်။ TRUNCATE နဲ့ဖျက်ရင် transaction log မချန်ခဲ့တဲ့အတွက် ROLLBACK လုပ်လို့မရပါဘူး။ DELETE မှာ transaction log ချန်ခဲ့တဲ့အတွက် ROLLBACK လုပ်လို့ရပါတယ်။

3.2.4 Understanding transactions



Transactions ဆိုတာ အမျိုးအစားတူတဲ့ **database operation** တွေကို သူ့အသုတ်နဲ့သူပေါင်းလုပ်တာပါ။ ဒီအသုတ်ထဲမှာပါတဲ့ **operation** တွေအကုန်အောင်မြင်မှ **transaction** အောင်မြင်မယ်၊ တခုခု မှားရင် **transaction** တသုတ်လုံး **fail** ဖြစ်ပါတယ်။ **Transaction** တွေမှာ အတိုကောက် **ACID** ဆိုတဲ့ ဂုဏ်သတ္တိ (၄) ခုရှိပါတယ်။

Atomicity

operation အားလုံးအောင်မြင်မှ နောက်ဆုံးပိတ်အောင်မြင်မယ်၊ တစ်ခု **fail** ဖြစ်တာနဲ့ ဆက်မလုပ်တော့ပါဘူး။

Consistency

database ထဲက ဒေတာတွေ တသမတ်တည်း **valid** ဖြစ်နေရပါမယ်

Isolation

Transaction တခုနဲ့တခု အပြန်အလှန်သက်ရောက်မှုမရှိစေရပါဘူး

Durability

Transaction ပြီးသွားပြီဆိုရင်တော့ ဒေတာတွေက တည်မြဲသွားပါပြီ။

ဥပမာ လူနာတစ်ယောက်ရဲ့ **visit** မှတ်တမ်းမှာ သူ့ကိုကြည့်ခဲ့တဲ့ ဆရာဝန်ကို နောက်ဆရာဝန်တစ်ယောက်နဲ့ပြင်ကြည့်ပါမယ်။

Before Transfer

visit_id	patient_id	doctor_id	visit_date	diagnosis
1	1	1	2024-03-25 09:30	Angina
4	1	4	2024-03-26 11:30	Contact dermatitis

After Transfer (doctor_id: 4 → 2)

visit_id	patient_id	doctor_id	visit_date	diagnosis
1	1	1	2024-03-25 09:30	Angina
4	1	2	2024-03-26 11:30	Contact dermatitis

patient_id = 1 က နှစ်ခါလာပြထားပါတယ်။ ဒုတိယလာပြတဲ့အခေါက်မှာကြည်ပေးတဲ့ ဆရာဝန်က doctor_id = 4 ပါ။ doctor_id ကို 2 ပြင်ကြည့်ပါမယ်။

```sql
-- Transaction စတင်ပါမယ်
START TRANSACTION;

-- လူနာရဲ့ visit မှတ်တမ်းကို ဆရာဝန်အသစ်ဆီ ပြောင်းပါမယ်
UPDATE visits
SET doctor_id = 2  -- ဆရာဝန်အသစ် (Dr. Aye Aye)
WHERE patient_id = 1  -- လူနာ ID
AND doctor_id = 4  -- ဆရာဝန်အဟောင်း (Dr. Su Su)
AND visit_id = 4;  -- ဒီတစ်ကြိမ်လာပြတာပဲ ပြောင်းမယ်

-- ပြောင်းလဲမှုအသစ်တွေကို မှတ်တမ်းထားပါမယ်
INSERT INTO visit_transfer_logs (
    patient_id,
    old_doctor_id,
    new_doctor_id,
    visit_id,
    transfer_date,
    reason
) VALUES (
    1,  -- လူနာ ID
    4,  -- ဆရာဝန်အဟောင်း ID (Dr. Su Su)
```

```
    2,  -- ဆရာဝန်အသစ် ID (Dr. Aye Aye)
    4,  -- visit_id
    CURRENT_TIMESTAMP,
    'Transfer to pediatrician for specialized care'
);

-- အားလုံးအဆင်ပြေရင် COMMIT လုပ်ပါမယ်
COMMIT;
```

ဒီလုပ်ငန်းမှာ တနေရာမှာ **error** ဖြစ်သွားရင် ရောက်တဲ့နေရာမှာရပ်ပါတယ်။ ရှေ့ဆက်မလုပ်တော့ပါဘူး။ ဒါပေမယ့် **error** မဖြစ်ခင်အထိကတော့ **database** ထဲဝင်သွားပါပြီ။ ဒါကို **Atomicity** ဂုဏ်သတ္တိလို့ခေါ်ပါတယ်။ ပြီးသွားသလောက်ကိုပြန်ဖျက်ချင်ရင်

```
-- Error ဖြစ်တဲ့အခါ အားလုံးပြန်ဖျက်
ROLLBACK;
```

table တစ်ခုထက်မကပြင်ရတဲ့အခါမျိုးမှာ၊ **data validity** ကိုဦးစားပေးချင်တဲ့အခါမျိုးမှာ၊ ငွေရေးကြေးရေးလို အရေးကြီးတဲ့ **database operation** လုပ်တဲ့အခါ၊ **error** တခုခုဖြစ်ရင် နောက်ကြောင်းပြန်ခွင့်ရချင်တဲ့အခါမျိုးမှာ **transaction** တွေသုံးသင့်ပါတယ်။

Transaction သုံးမယ်ဆိုရင် သတိထားရမယ့်အရာလေးတွေလည်းရှိပါတယ်။ **Transaction** တစ်ခုထဲမှာ **database operation** တွေအများကြီးစုပြုံထည့်ထားမယ်ဆိုရင် အဲဒီ **operatoin** တွေအတွက် **lock** လုပ်ထားရတာမို့လို့ တခြား **database operation** တွေနှောင့်နှေးစေနိုင်ပါတယ်။ **COMMIT** လုပ်ပြီးရင် **ROLLBACK** ပြန်လုပ်လို့မရတော့ပါဘူး။ ဒါကြောင့် **COMMIT** မလုပ်ခင် **data validity** ကိုသေချာစစ်ဆေးရပါမယ်။

3.3 Joins and Relationships

Primary Key ၊ **Foreign Key** အပိုင်းမှာ **table relationship** တွေအကြောင်း အနည်းငယ်ဖော်ပြပြီးသားပါ။

ဒီပုံမှာ ညာဘက်မှာရှိတဲ့ **visits table** ထဲက **patient_id column** က **patients table** ထဲက **patient_id** ကိုရည်ညွှန်းပါတယ်။ **patient_id** က **visits table** မှာ **Foreign Key** အဖြစ်ရှိနေတာပါ။
လူနာတစ်ယောက်ဆီမှာ ဆေးခန်းလာပြတဲ့မှတ်တမ်းတစ်ခုထက်မက ရှိနိုင်ပါတယ်။ တနည်းပြောရရင် လူနာတစ်ယောက်ဟာ ဆေးခန်းကိုတစ်ခါထက်မက လာပြတတ်ပါတယ်။ ဒီလိုဆက်စပ်မှုကို **One-to-Many relationship** လို့ခေါ်ပါတယ်။ လူနာတစ်ယောက် **(One)** နဲ့ သူ့ရဲ့ဆေးခန်းလာပြတဲ့မှတ်တမ်း အများကြီး **(Many)** ရှိနိုင်တယ်ပေါ့။

အခြေခံအားဖြင့် relationship အမျိုးအစား သုံးမျိုးရှိပါတယ်

One-to-One (1:1)
လူနာတစ်ယောက်ဆီမှာ မှတ်ပုံတင်နံပါတ်တစ်ခုပဲရှိသလိုမျိုး တစ်ခုနဲ့တစ်ခု တိုက်ရိုက်ဆက်စပ်နေတာမျိုးပါ။ ဥပမာ လူနာတစ်ယောက်ဆီမှာ medical card တစ်ခုပဲရှိတယ်ဆိုပါစို့။ ဒါဆိုလူနာတိုင်းဟာ သူနဲ့သက်ဆိုင်တဲ့ medical card တစ်ခုစီရှိတယ်။ Medical card တစ်ခုဟာလည်း လူနာတစ်ယောက်ရဲ့ medical card ပဲဖြစ်တယ်။

One-to-Many (1:n)
အများဆုံးတွေ့ရတဲ့ relationship ပုံစံပါ။ ဥပမာ လူနာတစ်ယောက်ဆီမှာ ဆေးခန်းလာပြတာ၊ ဓါတ်ခွဲစစ်ဆေးတာ၊ ဆေးညွှန်းတွေတစ်ခုထက်မကရှိတတ်တာတွေပါ။ ဆရာဝန်တစ်ယောက်ဆီမှာ လူနာတစ်ယောက်ထက်မက ကြည့်ရှုရတာလည်း ဒီပုံစံပါပဲ။

Many-to-Many (m:n)
ဆရာဝန်တစ်ယောက်က လူနာအများကြီးကို ကြည့်ရှုနိုင်သလို၊ လူနာတစ်ယောက်လည်း ဆရာဝန်အများကြီးဆီက ကုသမှုခံယူနိုင်တဲ့ပုံစံမျိုးပါ။ Many-to-Many relationship တွေကို junction table လို့ခေါ်တဲ့ ကြားခံ table တစ်ခုနဲ့ချိတ်ဆက်ပေးရလေ့ရှိပါတယ်။

လက်တွေ့အသုံးချတဲ့အခါမှာ table တစ်ခုနဲ့တစ်ခုဘယ်လိုဆက်စပ်နေလဲသိထားမှ join ဆိုတဲ့ command နဲ့ အမှန်ကန်ဆုံး ဆက်သွယ်နိုင်မှာဖြစ်ပါတယ်။

3.3.1 INNER JOIN

table နှစ်ခုကနေ relationship ဆက်နွယ်နေတဲ့ အချက်အလက်တွေယူချင်တဲ့အခါ JOIN ကိုသုံးရပါတယ်။ JOIN ထဲမှာ INNER JOIN ၊ LEFT JOIN ၊ RIGHT JOIN ဆိုပြီးရှိပါတယ်။ INNER JOIN ကတော့ table နှစ်ခုလုံးမှာ တူညီတဲ့အချက်အလက်တွေကိုပဲ ယူတာပါ။ ဥပမာပြရရင် လူနာတစ်ယောက်ရဲ့အမည်နဲ့ လာပြတဲ့ရက်စွဲတွေကို တွဲကြည့်မယ်ဆိုပါစို့။

```
SELECT
    p.full_name,
    v.visit_date
FROM patients p
INNER JOIN visits v ON p.patient_id = v.patient_id;
```

Full Name	Visit Date
Aung Aung	2024-03-25
Aung Aung	2024-03-26
Ma Ma	2024-03-25
Ko Ko	2024-03-25

ဒီ query မှာ
FROM patients p က patients table ကို p ဆိုတဲ့ alias နာမည်ပေးထားပါတယ်။
INNER JOIN visits v က visits table ကိုတော့ v ဆိုတဲ့ alias နာမည်ပေးပါတယ်။
ON p.patient_id = v.patient_id က patients table ထဲက patient_id နဲ့ visits table ထဲက patient_id တူတာကို ရွေးထုတ်မယ်ဆိုတဲ့သဘောပါ။
INNER JOIN ဆိုတာ ပုံနဲ့ပြောရရင် ဝိုင်းနှစ်ခုထပ်နေတဲ့ဘုံနေရာလောက်ပဲ ယူသွားတာပါ။ လူနာနာမည်တွေပါတဲ့ ပထမဝိုင်းနဲ့ လာပြတဲ့မှတ်တမ်းတွေပါတဲ့ ဒုတိယဝိုင်းနှစ်ခုက patient_id တူညီတဲ့နေရာမှာ ထပ်နေပါတယ်။ INNER JOIN က အဲဒီထပ်နေတဲ့အပိုင်းကိုပဲယူသွားတာပါ။

ဥပမာ လူနာတွေထဲမှာ ဆေးခန်းလာမပြဖူးသေးတဲ့သူရှိနိုင်ပါတယ်။ လူနာမှတ်ပုံတင်ဌာနမှာစာရင်းသွင်းပြီးမှ ဆေးခန်းမပြဖြစ်တော့ဘဲ ပြန်သွားတာမျိုးရှိတတ်လို့ပါ။ **visits table** ထဲမှာ သူ့ရဲ့မှတ်တမ်းရှိမှာမဟုတ်ပါဘူး။ **INNER JOIN** လုပ်လိုက်တဲ့အခါ အဲဒီလူနာကို ပြမှာမဟုတ်ပါဘူး။ တနည်းအားဖြင့် ဆေးခန်းလာပြဖူးတဲ့လူနာတွေကိုပဲ ရွေးပြပေးတာပါ။
JOIN မှာ တစ်ခုထက်ပိုပြီး ချိတ်လို့ရပါတယ်။ ဥပမာ လူနာအမည်၊ ဆရာဝန်အမည်နဲ့ လာပြတဲ့နေ့စွဲတွေကို တွဲကြည့်မယ်ဆိုပါစို့။

```
SELECT
    p.full_name as patient_name,
    d.full_name as doctor_name,
    v.visit_date
FROM patients p
INNER JOIN visits v ON p.patient_id = v.patient_id
INNER JOIN doctors d ON v.doctor_id = d.doctor_id;
```

Patient Name	Doctor Name	Visit Date
Aung Aung	Dr. Thiha	2024-03-25
Aung Aung	Dr. Su Su	2024-03-26
Ma Ma	Dr. Aye Aye	2024-03-25
Ko Ko	Dr. Kyaw Min	2024-03-25

ဒီတစ်ခါ **query** မှာတော့ **table** သုံးခုကို ချိတ်ဆက်ထားပါတယ်။ **patients table** ကနေ လူနာအမည်၊ **doctors table** ကနေ ဆရာဝန်အမည်၊ **visits table** ကနေ လာပြတဲ့ရက်စွဲတွေ ယူထားပါတယ်။
ဘယ်အချိန်မှာ **LEFT JOIN** သုံးမလဲ၊ ဘယ်အချိန်မှာ **INNER JOIN** သုံးမလဲဆိုတာ မြေပြင်အခြေအနေပေါ်မူတည်ပြီး ဆုံးဖြတ်ရပါတယ်။
ဥပမာ လူနာအားလုံးကို လာပြတာပဲဖြစ်ဖြစ်၊ မလာပြတာပဲဖြစ်ဖြစ် မြင်ချင်တယ်ဆိုရင် **LEFT JOIN** သုံးပါတယ်။
လာပြဖူးတဲ့သူတွေကိုပဲကြည့်ချင်တယ်ဆိုရင် **INNER JOIN** သုံးပါတယ်။ **LEFT JOIN** သုံးမယ်ဆိုရင် **LEFT table** ပိုင် **column** တွေကို **SELECT** ထဲမှာမဖြစ်မနေထည့်ပေးရမှာဖြစ်ပြီး **RIGHT table** ပိုင် **column** တွေက **NULL** ဖြစ်နေနိုင်တာကို သတိထားရပါမယ်။

3.3.2 LEFT and RIGHT JOIN

INNER JOIN လုပ်တဲ့အခါ **table** နှစ်ခုစလုံးမှာ **relationship** ဆက်နွယ်နေတဲ့ **record** တွေပဲရပါတယ်။ **LEFT JOIN** နဲ့ **RIGHT JOIN** ကတော့ ဆန့်ကျင်ဘက် **table** မှာကိုယ်နဲ့ဆိုင်တဲ့ **record** မရှိလည်း ကိုယ့် **record** ပါပြစေချင်ရင်သုံးပါတယ်။
LEFT JOIN ဆိုတာ ဘယ်ဘက်က **table** မှာရှိတဲ့ **record** အားလုံးကိုယူပါတယ်။ ညာဘက် **table** မှာ **match** ဖြစ်တဲ့ **record** တွေပါရင်တွဲပေးပြီး၊ **match** မဖြစ်တဲ့ဟာဆိုရင်တော့ **NULL** တန်ဖိုးပေးသွားပါတယ်။
ဥပမာ လူနာလာပြတဲ့မှတ်တမ်းထုတ်ကြည့်မယ်ဆိုပါစို့။ လူနာအားလုံးကိုပြချင်တယ်။ ဒါပေမယ့် လာမပြသေးတဲ့လူနာတွေလည်း စာရင်းထဲမှာပါစေချင်တယ်ဆိုရင် **LEFT JOIN** သုံးရပါတယ်။

```
SELECT
    p.registration_no,
    p.full_name,
    v.visit_date
FROM patients p
LEFT JOIN visits v ON p.patient_id = v.patient_id
```

Registration No	Full Name	Visit Date
REG001	Aung Aung	2024-03-25
REG002	Ma Ma	2024-03-25
REG003	Ko Ko	NULL
REG004	Mya Mya	NULL

ဒီနေရာမှာ **Ko Ko** နဲ့ **Mya Mya** က ဆေးခန်းမလာပြသေးတဲ့လူတွေပါ။ **LEFT JOIN** သုံးထားတာမို့လို့ **patients table** ထဲက လူနာအားလုံးပါသွားပါတယ်။ **visit_date column** မှာ **NULL** တန်ဖိုးပြနေတာက သူတို့နဲ့သက်ဆိုင်တဲ့ **visits record** မရှိလို့ပါ။ **RIGHT JOIN** ကတော့ **LEFT JOIN** နဲ့ပြောင်းပြန်ပါပဲ။ ညာဘက် **table** က **record** အားလုံးယူပြီး ဘယ်ဘက် **table** နဲ့ တွဲပေးတာပါ။ ညာဘက် **table** မှာရှိတဲ့ **record** အားလုံးကို အဓိကထားပြီး ဘယ်ဘက်မှာ **match** မဖြစ်ရင် **NULL** တန်ဖိုးပေးသွားပါတယ်။

```
SELECT
    d.full_name AS doctor_name,
    v.visit_date,
    p.full_name AS patient_name
FROM patients p
RIGHT JOIN visits v ON p.patient_id = v.patient_id
RIGHT JOIN doctors d ON v.doctor_id = d.doctor_id;
```

Doctor Name	Visit Date	Patient Name
Dr. Thiha	2024-03-25	Aung Aung
Dr. Aye Aye	2024-03-25	Ma Ma
Dr. Kyaw Min	NULL	NULL
Dr. Su Su	NULL	NULL

ဒီ **query** မှာတော့ **doctors table** ကို **RIGHT JOIN** နဲ့ချိတ်ထားတာမို့လို့ ဆရာဝန်အားလုံးပါဝင်သွားပါတယ်။ **Dr. Kyaw Min** နဲ့ **Dr. Su Su** က လူနာမကြည့်ရသေးတဲ့ ဆရာဝန်တွေမို့လို့ **visit_date** နဲ့ **patient_name** တွေမှာ **NULL** တန်ဖိုးပြနေတာပါ။
ဒီ **query** ကိုပဲ **LEFT JOIN** နဲ့ရေးချင်ရင်လည်း **table** တွေရဲ့နေရာကို ပြောင်းရေးလို့ရပါတယ်။

```
SELECT
    d.full_name AS doctor_name,
    v.visit_date,
    p.full_name AS patient_name
FROM doctors d
LEFT JOIN visits v ON d.doctor_id = v.doctor_id
LEFT JOIN patients p ON v.patient_id = p.patient_id;
```

လက်တွေ့မှာ **LEFT JOIN** က **RIGHT JOIN** ထက်ပို အသုံးများပါတယ်။ အလုပ်လုပ်ပုံချင်းအတူတူပါပဲ။ **table** တွေနေရာ ပြောင်းပြီးရေးလို့ရတာမို့လို့ **LEFT JOIN** တစ်မျိုးတည်းသုံးပြီး **query** ကို ရှင်းရှင်းလင်းလင်းရေးတာ ပိုကောင်းပါတယ်။

CHAPTER 4
ADVANCED TOPICS AND BEST PRACTICES

4 Chapter 4 - Advanced Topics and Best Practices

4.1 Advanced Queries

4.1.1 Aggregate functions (COUNT, SUM, AVG)

SQL မှာ အသုံးများတဲ့ aggregate function တွေကတော့ COUNT, SUM, AVG တို့ဖြစ်ပါတယ်။

COUNT()
COUNT() function က record အရေအတွက်ကို ရေတွက်ပေးပါတယ်။ ဥပမာ လူနာအရေအတွက်၊ ဆေးညွှန်းအရေအတွက် စတာတွေရေတွက်တဲ့အခါ သုံးပါတယ်။

```
SELECT COUNT(*) as total_patients FROM patients;
```

ဒါက patients table ထဲက လူနာအရေအတွက်စုစုပေါင်းကို ရေတွက်တာပါ။ တစ်ခုခုစစ်ပြီးမှရေတွက်ချင်ရင်လည်း WHERE နဲ့စစ်လို့ရပါတယ်။ ဥပမာ အမျိုးသား/အမျိုးသမီးလူနာအရေအတွက်သိချင်ရင်

```
SELECT
    gender,
    COUNT(*) as patient_count
FROM patients
GROUP BY gender;
```

Gender	Patient Count
M	45
F	38

NULL တွေကိုပါ ရေတွက်ချင်ရင် COUNT(*) သုံးပြီး NULL တွေမပါစေချင်ရင် COUNT(column_name) သုံးရပါတယ်။ ဥပမာ phone_number က optional ဖြစ်လို့ NULL တန်ဖိုးတွေပါနေနိုင်ပါတယ်။ ဒီလိုအခြေအနေမှာ အောက်က query နှစ်ခုရဲ့ရလဒ်မတူနိုင်ပါဘူး။

```
SELECT COUNT(*) as all_patients,
       COUNT(phone_number) as patients_with_phone
FROM patients;
```

All Patients	Patients with Phone
83	65

SUM()
SUM() function က တန်ဖိုးတွေကို ပေါင်းပေးပါတယ်။ ဥပမာ ဆေးခန်းကြေး၊ ဓါတ်ခွဲခ စသည်တွေကို စုစုပေါင်းတွက်တဲ့အခါ SUM() ကိုသုံးပါတယ်။

```
SELECT
    MONTH(visit_date) as month,
```

```
    COUNT(*) as visit_count,
    SUM(consultation_fee) as total_fees
FROM visits
GROUP BY MONTH(visit_date)
ORDER BY month;
```

Month	Visit Count	Total Fees ($)
January	45	2,250.00
February	38	1,900.00

AVG()

AVG() function က တန်ဖိုးတွေရဲ့ပျမ်းမျှကိုတွက်ပေးပါတယ်။ ဥပမာ လူနာတွေရဲ့ ပျမ်းမျှအသက်၊ ဆရာဝန်တစ်ယောက်ကို တစ်နေ့ပျမ်းမျှလူနာဘယ်နှစ်ယောက်လာပြလဲ စတာတွေတွက်တဲ့အခါ သုံးပါတယ်။

```
SELECT
    d.full_name as doctor_name,
    COUNT(*) as total_patients,
    AVG(consultation_fee) as avg_fee
FROM visits v
JOIN doctors d ON v.doctor_id = d.doctor_id
GROUP BY d.doctor_id, d.full_name;
```

AVG() function က **NULL** တန်ဖိုးတွေကိုထည့်မတွက်ပါဘူး။ ဥပမာ **fee column** မှာ တန်ဖိုး (၅) ခုရှိပြီး နှစ်ခုက **NULL** ဆိုရင် **AVG()** က (၃) ခုကိုပဲယူပြီး သူတို့ရဲ့ပျမ်းမျှကိုတွက်ပေးပါတယ်။

Aggregate function တွေနဲ့ **GROUP BY** တွဲသုံးတဲ့အခါ သတိထားရမှာတစ်ခုက **GROUP BY** ထဲမှာမပါတဲ့ **column** ကို **SELECT** ထဲမှာ ထည့်ယူလို့မရပါဘူး။ ဥပမာပြရရင် -

```
-- ဒီလိုရေးလို့မရပါဘူး
SELECT
    specialty,
    full_name,   -- GROUP BY မှာမပါဘူး
    COUNT(*) as doctor_count
FROM doctors
GROUP BY specialty;

-- ဒီလိုပြင်ရပါမယ်
SELECT
    specialty,
    COUNT(*) as doctor_count
FROM doctors
GROUP BY specialty;
```

အခြေခံ **aggregate function** တွေအပြင် အခြား **function** တွေလည်းရှိပါသေးတယ်။

MIN() - အနိမ့်ဆုံးတန်ဖိုးကိုရှာပေးပါတယ်
MAX() - အမြင့်ဆုံးတန်ဖိုးကိုရှာပေးပါတယ်
STDDEV() - Standard Deviation တွက်ပေးပါတယ်
VARIANCE() - ကွဲလွဲချက်ပမာဏ (Variance) တွက်ပေးပါတယ်

Aggregate function တွေဟာ database ထဲက အချက်အလက်တွေကို အနှစ်ချုပ်ကြည့်တဲ့အခါ အထူးအသုံးဝင်ပါတယ်။ အထူးသဖြင့် report တွေပြုလုပ်တဲ့အခါ၊ ဂရပ်တွေဆွဲတဲ့အခါ၊ စာရင်းအင်းပိုင်းဆိုင်ရာ တွက်ချက်မှုတွေလုပ်တဲ့အခါမှာ အသုံးများပါတယ်။

4.1.2 Window function

Window function က row တစ်ခုချင်းစီအတွက် formula နဲ့ calculation လုပ်တဲ့အခါ ဆက်စပ်နေတဲ့တခြား row တွေကိုပါ calculation ထဲကိုထည့်သွင်းလို့ရအောင်လုပ်ပေးပါတယ်။ report တွေထုတ်တဲ့အခါ၊ trend analysis လုပ်တဲ့အခါ၊ ကိန်းဂဏန်းတွေကို ranking သတ်မှတ်တဲ့အခါမျိုးတွေမှာ အဓိကသုံးပါတယ်။ ဥပမာ လူနာတစ်ယောက်ချင်းစီရဲ့ ဆေးခန်းလာပြတဲ့အကြိမ်အရေအတွက်ကို တခြားလူနာတွေနဲ့နှိုင်းယှဉ်ကြည့်ချင်တယ်ဆိုရင်

```sql
SELECT
    p.full_name,
    COUNT(*) as visit_count,
    AVG(COUNT(*)) OVER () as avg_visits,
    RANK() OVER (ORDER BY COUNT(*) DESC) as visit_rank
FROM visits v
JOIN patients p ON v.patient_id = p.patient_id
GROUP BY p.patient_id, p.full_name;
```

Full Name	Visit Count	Avg Visits	Visit Rank
Aung Aung	5	3.25	1
Ma Ma	4	3.25	2
Ko Ko	2	3.25	3
Su Su	2	3.25	3

ဒီ query မှာ window function သုံးခုပါပါတယ်။

COUNT(*) က လူတစ်ယောက်ချင်းစီ ဆေးခန်းလာပြတဲ့အကြိမ်ရေကို တွက်ပေးတယ်

AVG(COUNT(*)) OVER () က လူနာအားလုံးရဲ့ ပျမ်းမျှလာပြတဲ့အကြိမ်ရေကို တွက်ပေးပါတယ်။ OVER() နောက်မှာဘာမှမရေးထားလို့ table တစ်ခုလုံးပေါ်မှာ တွက်ပေးသွားမှာပါ။

RANK() OVER (ORDER BY COUNT(*) DESC) က လူနာတစ်ယောက်ချင်းရဲ့ visit_count ကို အများဆုံးကနေ အနည်းဆုံးအထိ အဆင့်သတ်မှတ်ပေးသွားပါတယ်။

အသုံးများတဲ့ window function တွေက

ROW_NUMBER() - နံပါတ်စဉ်တပ်ပေးတယ်

RANK() - အဆင့်သတ်မှတ်ပေးတယ်

DENSE_RANK() က **RANK()** လိုပါပဲ၊ ဒါပေမယ့်အဆင့်တွေကို ကွက်လပ်မချန်ဘဲ ဆက်တိုက်ရေတွက်ပေးပါတယ်

LAG() - လက်ရှိ **row** ရဲ့အပေါ် **row** က တန်ဖိုးကိုယူပါတယ်

LEAD() - လက်ရှိ **row** ရဲ့အောက် **row** က တန်ဖိုးကိုယူပါတယ်။

4.1.3 GROUP BY and HAVING

GROUP BY ကို အုပ်စုလိုက်စုစည်းပြီး အချက်အလက်တွေကို လေ့လာဖို့သုံးပါတယ်။ **HAVING** ကတော့ **GROUP BY** နဲ့ စုစည်းထားတဲ့အုပ်စုတွေကို ထပ်ပြီးစစ်ထုတ်ဖို့သုံးပါတယ်။ **SQL query** တစ်ခုထဲမှာ **GROUP BY** နဲ့ **HAVING** တွဲသုံးတာများပါတယ်။

ဥပမာ ဆရာဝန်တိုင်းက တစ်နေ့တာအတွင်း လူနာဘယ်နှစ်ယောက်စီ ကြည့်လဲ သိချင်တယ်ဆိုပါစို့။ **GROUP BY** နဲ့ဒီလိုရေးလို့ရပါတယ်။

```sql
SELECT
    d.full_name,
    DATE(v.visit_date) AS visit_day,
    COUNT(*) AS patient_count
FROM visits v
JOIN doctors d ON v.doctor_id = d.doctor_id
GROUP BY d.full_name, DATE(v.visit_date)
ORDER BY visit_day, d.full_name;
```

Doctor Name	Visit Day	Patient Count
Dr. Aye Aye	2024-03-25	1
Dr. Kyaw Min	2024-03-25	1
Dr. Su Su	2024-03-26	1
Dr. Thiha	2024-03-25	1

query မှာ **doctors table** ထဲက ဆရာဝန်အမည်နဲ့ **visits table** ကနေ ကြည့်တဲ့လူနာအရေအတွက်ကို တွဲပြထားပါတယ်။ **COUNT(*)** နဲ့ လူနာအရေအတွက်ကိုရေတွက်တာပါ။ **GROUP BY** မှာ ဆရာဝန်အမည်နဲ့ ရက်စွဲနှစ်ခုလုံးထည့်ထားတဲ့အတွက် ဆရာဝန်တစ်ယောက်ချင်းစီ တစ်နေ့တာအတွင်း လူနာဘယ်နှစ်ယောက် ကြည့်လဲဆိုတာ တွက်ချက်ပေးသွားတာပါ။

GROUP BY နဲ့တွဲပြီး **aggregate functions** တွေသုံးတာများပါတယ်။ အသုံးများတဲ့ **aggregate functions** တွေကတော့ -

- **COUNT()** - အရေအတွက်ရေတွက်တာ
- **SUM()** - တန်ဖိုးတွေပေါင်းတာ
- **AVG()** - ပျမ်းမျှတွက်တာ

- **MIN()** - အနိမ့်ဆုံးတန်ဖိုးရှာတာ
- **MAX()** - အမြင့်ဆုံးတန်ဖိုးရှာတာ

စတာတွေပါ။

အောက်က **query** မှာ တစ်ယောက်ချင်းစီအတွက်ကိုတော့ **GROUP BY** နဲ့ရေးလို့မရနိုင်တဲ့အတွက် **HAVING** နဲ့ထပ်ပြီးစစ်ထုတ်ပြထားပါတယ်။

ဒီ **query** မှာတော့ ဆရာဝန်တစ်ယောက်ချင်းစီအလိုက် လူနာစစ်ဆေးပေးခဲ့တဲ့အကြိမ်အရေအတွက်နဲ့ စစ်ဆေးပေးခဲ့တဲ့ရက်အရေအတွက်ကိုပြထားပါတယ်။ **HAVING** နဲ့ **total_visits** အနည်းဆုံးတစ်ကြိမ်လက်ခံကြည့်ပေးဖူးတဲ့ဆရာဝန်တွေကိုပဲ စစ်ထုတ်ပြထားတာပါ။

HAVING နဲ့ **WHERE** ဘာကွာလဲဆိုရင် **WHERE** က မူလဒေတာတွေအပေါ်မှာ **filter** လုပ်တာပါ။ **HAVING** ကတော့ **GROUP BY** နဲ့ စုထားတဲ့ဒေတာတွေပေါ်မှာ **filter** လုပ်တာပါ။ **GROUP BY** မပါဘဲ **HAVING** တစ်ခုတည်းသုံးလို့မရပါဘူး။

ဥပမာပြရရင်

```
-- ရောဂါလက္ခဏာ headache ပါတာကိုပဲစစ်ထုတ်ပြီးမှ အုပ်စုလိုက်လုပ်
SELECT doctor_id, COUNT(*) as visit_count
FROM visits
WHERE chief_complaint LIKE '%headache%'
GROUP BY doctor_id;

-- အုပ်စုလိုက်လုပ်ပြီး လူနာ ၅ယောက်အထက်ကြည့်တဲ့ဆရာဝန်တွေပဲစစ်ထုတ်
SELECT doctor_id, COUNT(*) as visit_count
FROM visits
GROUP BY doctor_id
HAVING COUNT(*) > 5;
```

GROUP BY နဲ့ **HAVING** တို့ကို အဓိကအားဖြင့် **report** တွေထုတ်တဲ့အခါ အသုံးပြုကြပါတယ်။ အုပ်စုလိုက်လေ့လာလို့ရတဲ့အတွက် လူနာတွေရဲ့လာရောက်ပြသမှုပုံစံ၊ ဆရာဝန်တွေရဲ့ လူနာကြည့်ရှုမှုပုံစံတွေကို လေ့လာနိုင်ပါတယ်။

4.1.4 Subqueries

Subquery ဆိုတာ **query** တစ်ခုထဲမှာ ထပ်ပြီးရေးထည့်တဲ့ နောက်ထပ် **query** ကိုပြောတာပါ။ ရှုပ်ထွေးတဲ့ အချက်အလက်တွေရှာဖွေတဲ့အခါ **subquery** တွေ အသုံးပြုလေ့ရှိပါတယ်။ **Subquery** ကို ပုံစံသုံးမျိုးနဲ့ အသုံးပြုလို့ရပါတယ်။

WHERE clause ထဲမှာ သုံးတဲ့ subquery

ပျမ်းမျှထက်ပိုပြီး လူနာကြည့်ပေးနိုင်တဲ့ဆရာဝန်တွေရဲ့စာရင်းကို ကြည့်ပါမယ်။

```
SELECT full_name, specialty
FROM doctors
WHERE doctor_id IN (
    SELECT doctor_id
```

```
    FROM visits
    GROUP BY doctor_id
    HAVING COUNT(*) > (
        SELECT AVG(patient_count)
        FROM (
            SELECT COUNT(*) as patient_count
            FROM visits
            GROUP BY doctor_id
        ) as avg_visits
    )
);
```

ဒီ **query** မှာ **subquery** သုံးဆင့်ပါဝင်ပါတယ်။

- အတွင်းဆုံး **subquery** က ဆရာဝန်တစ်ယောက်ချင်းစီရဲ့ လူနာအရေအတွက်ကိုရှာပါတယ်
- အလယ် **subquery** က ပျမ်းမျှလူနာအရေအတွက်ကို တွက်ပါတယ်
- နောက်တဆင့် **subquery** က ပျမ်းမျှထက်များတဲ့ဆရာဝန်တွေကိုရွေးထုတ်ပါတယ်
- နောက်ဆုံး **main query** က ရွေးထုတ်ထားတဲ့ဆရာဝန်တွေရဲ့ အမည်နဲ့အထူးပြုဘာသာရပ်ကို ဆွဲထုတ်ပြပါတယ်

SELECT clause ထဲမှာ သုံးတဲ့ subquery

ဆရာဝန်တစ်ယောက်ချင်းစီရဲ့ လူနာအရေအတွက်နဲ့ ပျမ်းမျှလူနာအရေအတွက်တို့ကို နှိုင်းယှဉ်ကြည့်ပါမယ်။

```
SELECT
    d.full_name,
    COUNT(*) as total_patients,
    (SELECT AVG(patient_count)
     FROM (
         SELECT COUNT(*) as patient_count
         FROM visits
         GROUP BY doctor_id
     ) as avg_visits) as avg_patients
FROM visits v
JOIN doctors d ON v.doctor_id = d.doctor_id
GROUP BY d.doctor_id, d.full_name;
```

Doctor Name	Total Patients	Avg Patients
Dr. Thiha	5	3.5
Dr. Aye Aye	3	3.5
Dr. Kyaw Min	2	3.5

FROM clause ထဲမှာ သုံးတဲ့ subquery

လူနာတွေကို အသက်အုပ်စုအလိုက် စုစည်းပြီး လေ့လာကြည့်ပါမယ်။

```
SELECT
    age_group,
    COUNT(*) as patient_count
FROM (
    SELECT
        CASE
            WHEN TIMESTAMPDIFF(YEAR, date_of_birth, CURDATE()) < 18 THEN 'Child'
            WHEN TIMESTAMPDIFF(YEAR, date_of_birth, CURDATE()) < 60 THEN 'Adult'
            ELSE 'Elderly'
        END as age_group
    FROM patients
) as patient_ages
GROUP BY age_group;
```

Age Group	Patient Count
Child	1
Adult	2

4.1.5 Common Table Expressions (CTE) and Temporary Tables

Common Table Expression (CTE) ဆိုတာ ရှုပ်ထွေးတဲ့ **query** တွေကို ဖတ်ရလွယ်အောင် အပိုင်းလိုက်ခွဲရေးတဲ့ နည်းစနစ်ပါ။ **CTE** ကို **WITH keyword** နဲ့စရေးပြီး **query** တစ်ခုတည်းမှာပဲထည့်ရေးနိုင်ပါတယ်။ ဥပမာပြထားတာ လူနာတစ်ယောက်ချင်းစီရဲ့ နောက်ဆုံးလာပြထားတဲ့ မှတ်တမ်းတွေပဲရွေးကြည့်တဲ့ **query** ပါ။

```
WITH LastVisits AS ( -- CTE နာမည်ပေး
    SELECT
        patient_id,
        MAX(visit_date) as last_visit_date
    FROM visits
    GROUP BY patient_id
) -- visits table ကိုပဲ အရင်တဖြတ် query လုပ်ထားတယ်

SELECT
    p.full_name,
    p.registration_no,
    lv.last_visit_date,
    v.chief_complaint
FROM patients p
JOIN LastVisits lv ON p.patient_id = lv.patient_id
JOIN visits v ON lv.patient_id = v.patient_id
```

```
    AND lv.last_visit_date = v.visit_date;
-- patients table နဲ့ join ပြီး query ထပ်လုပ်တယ်
```

Last Visits

Full Name	Registration No	Last Visit Date	Chief Complaint
Aung Aung	REG001	2024-03-26	Skin rash
Ma Ma	REG002	2024-03-25	Fever and cough
Ko Ko	REG003	2024-03-25	Headache

CTE သုံးပြီး query တွေအပိုင်းလိုက်ခွဲရေးလို့ရတဲ့အတွက် ဖတ်ရလွယ်တယ်။ query တစ်ခုတည်းမှာပဲ ဆက်တိုက်ရေးရတာမို့လို့ တခြား query တွေနဲ့ ငြိတာမရှိပါဘူး။ query ပြီးတာနဲ့ အလိုအလျောက်ပျက်သွားတဲ့အတွက် database ထဲမှာနေရာမယူပါဘူး။

Temporary table တွေကကျတော့ CTE နဲ့အလုပ်လုပ်ပုံဆင်တူပေမယ့် temporary table ဖန်တီးဖို့ CREATE command နဲ့လုပ်ပေးရပြီး DROP command နဲ့ပြန်ဖျက်ပေးရပါတယ်။ CTE က query တစ်ခုတည်းမှာပဲ သုံးလို့ရပေမယ့် temporary table တွေကတော့ တခြား query တွေကနေ ခေါ်သုံးရပါတယ်။ DROP နဲ့ပြန်မဖျက်မချင်း database ထဲမှာနေရာယူပါတယ်။

4.1.6 Views

View ဆိုတာ virtual table တစ်ခုလို့ပြောလို့ရပါတယ်။ ရှုပ်ထွေးတဲ့ query တွေကို view အဖြစ်သိမ်းထားပြီး နောက်ပိုင်းလိုအပ်တဲ့အခါ ပြန်ခေါ်သုံးလို့ရပါတယ်။ View တည်ဆောက်တဲ့အခါ အဲဒီ view မှာပါဝင်တဲ့ column တွေကိုပဲ user တွေမြင်အောင် ပြလို့ရတဲ့အတွက် data security အတွက်လည်း အထောက်အကူဖြစ်စေပါတယ်။

ဥပမာ လူနာတွေရဲ့အခြေခံအချက်အလက်တွေပေါ်မှာ အခြေခံပြီး view တစ်ခုဆောက်ကြည့်ပါမယ်

```
CREATE VIEW patient_summary AS
SELECT
    p.registration_no,
    p.full_name,
    p.gender,
    TIMESTAMPDIFF(YEAR, p.date_of_birth, CURDATE()) as age,
    COUNT(v.visit_id) as visit_count,
    MAX(v.visit_date) as last_visit
FROM patients p
LEFT JOIN visits v ON p.patient_id = v.patient_id
GROUP BY p.patient_id, p.registration_no, p.full_name, p.gender, p.date_of_birth;
```

View တစ်ခုဖန်တီးပြီးသွားရင် အောက်ပါအတိုင်း သာမန် table တစ်ခုလို အသုံးပြုနိုင်ပါတယ်။

```
SELECT * FROM patient_summary;
```

View တစ်ခုကို ဖျက်ချင်ရင်တော့

```
DROP VIEW patient_summary;
```

View တွေကို အဓိကအားဖြင့် အောက်ပါအခြေအနေတွေမှာ အသုံးပြုလေ့ရှိပါတယ်။

ရှုပ်ထွေးတဲ့ query တွေကို ရိုးရှင်းအောင်လုပ်ဖို့

JOIN တွေအများကြီးပါတဲ့ query တွေကို view အနေနဲ့သိမ်းထားပြီး နောက်ပိုင်းလိုအပ်တဲ့အခါ view ကိုပဲ ပြန်ခေါ်သုံးလို့ရပါတယ်။ ဥပမာ

```
SELECT * FROM patient_summary WHERE age > 60;
```

Data security အတွက်

လုံခြုံရေးအရ user တွေကိုပြချင်တဲ့ column တွေကိုပဲ view ထဲထည့်ထားပြီး view ကိုပဲ access ပေးလို့ရပါတယ်။ ဥပမာ အထက်က patient_summary view မှာဆိုရင် patient_id တို့လို internal ID တွေကို မထည့်ထားတာကို တွေ့ရမှာဖြစ်ပါတယ်။

Report တွေထုတ်တဲ့အခါ

Report တွေထုတ်တဲ့အခါ အသုံးများတဲ့ query တွေကို view အဖြစ်သိမ်းထားပြီး လိုအပ်တဲ့အခါ အလွယ်တကူပြန်ခေါ်သုံးလို့ရပါတယ်။

View တွေနဲ့ပတ်သက်ပြီး သတိထားရမှာကတော့ View တွေအများကြီးဆောက်ထားရင် database ရဲ့ performance ကို ထိခိုက်နိုင်ပါတယ်။ အဲဒါကြောင့် တကယ်လိုအပ်တဲ့ view တွေကိုပဲ ဆောက်သင့်ပါတယ်။ View တစ်ခုကို user အများအပြားက တစ်ပြိုင်နက်တည်း access လုပ်တဲ့အခါ database server ရဲ့ load များနိုင်ပါတယ်။ ဒီလိုအခြေအနေမျိုးမှာ materialized view သုံးတာ ပိုအဆင်ပြေပါတယ်။

View နာမည်တွေပေးတဲ့အခါ ရှင်းလင်းတဲ့နာမည်တွေပေးသင့်ပါတယ်။ ဥပမာ "v1", "v2" လိုမျိုး နာမည်တွေအစား "patient_summary", "visit_history" စတဲ့ နာမည်တွေက ပိုပြီးအဓိပ္ပါယ်ရှိပါတယ်။

Database backup လုပ်တဲ့အခါ View တွေကိုလည်းregular backup လုပ်ထားသင့်ပါတယ်။ ဒါမှသာ database ပြဿနာတစ်ခုခုဖြစ်လာရင် အလွယ်တကူပြန်ရယူနိုင်မှာ ဖြစ်ပါတယ်။

Materialized view

Materialized view ဆိုတာက query ရဲ့ရလဒ်တွေကို ခဏခဏအသုံးပြုရမယ်ဆိုရင် အဲဒီရလဒ်တွေကို သိမ်းထားလို့ရတဲ့ database object တစ်ခုပါ။ ပုံမှန် view နဲ့မတူတာက ပုံမှန် view က query run တိုင်း ဒေတာအသစ်ရနိုင်ပေမယ့် materialized view ကတော့ snapshot လုပ်ထားတဲ့ရလဒ်တွေကိုပဲပြပါတယ်။ Oracle database မှာတော့ snapshot လို့ပဲသုံးပါတယ်။ ဥပမာ report တွေမှာ လအလိုက် ရောဂါဖြစ်ပွားမှုစာရင်းတွေလိုချင်တယ်၊ ဒေတာက သိန်းချီရှိတယ်ဆိုပါစို့။ ဒီလိုအခြေအနေမှာ materialized view သုံးလို့ရပါတယ်။

```
CREATE MATERIALIZED VIEW monthly_disease_summary AS
SELECT
    EXTRACT(YEAR FROM report_date) as year,
    EXTRACT(MONTH FROM report_date) as month,
    disease_name,
    COUNT(*) as case_count
FROM disease_cases
GROUP BY
    EXTRACT(YEAR FROM report_date),
    EXTRACT(MONTH FROM report_date),
    disease_name;
```

Materialized view ကို refresh လုပ်ချင်တဲ့အခါမှ update လုပ်လို့ရပါတယ်။

```
REFRESH MATERIALIZED VIEW monthly_disease_summary;
```

Materialized view သုံးခြင်းအားဖြင့် **Query run** တဲ့အချိန် သက်သာပြီး **Server load** လျော့ချနိုင်ပါတယ်။ **Report** တွေမှာ **performance** ကောင်းစေပါတယ်။ ဒါပေမယ့် သတိထားရမှာက ဒေတာတွေအမြဲပြောင်းလဲနေရင်တော့ **materialized view** က မသင့်တော်ပါဘူး။ ဒေတာတွေအသစ်တိုးလာပေမယ့် **refresh** မလုပ်ရသေးသ၍ **materialized view** က အဟောင်းတွေပဲပြနေမှာဖြစ်လို့ပါ။ ဒါကြောင့် **daily transaction** တွေလိုမျိုး အမြဲတမ်း **update** လုပ်နေရမယ့် အခြေအနေမှာ **materialized view** မသုံးသင့်ပါဘူး။

အခမဲ့ရတဲ့ **MySQL Community Edition** မှာ **materialized view feature** မပါဝင်ပါဘူး။ **PostgreSQL** တို့၊ **Oracle** တို့မှာပဲ **materialized view** သုံးလို့ရပါတယ်။ လိုင်စင်ကြေးပေးပြီး ဝယ်သုံးရတဲ့ **MySQL Enterprise Edition** မှာပဲ **materialized view** ထည့်ပေးထားပါတယ်။

4.1.7 Indexes

Database တစ်ခုမှာ **table** တွေထဲက **data** တွေကို လျင်မြန်စွာ ရှာဖွေနိုင်ဖို့အတွက် **index** တွေကို အသုံးပြုပါတယ်။ အခြေခံအားဖြင့် စာအုပ်တစ်အုပ်ရဲ့ **index** စာမျက်နှာကို နမူနာယူနိုင်ပါတယ်။ စာအုပ်တစ်အုပ်လုံးကို တစ်မျက်နှာချင်းစီလှန်မနေပဲ **index** စာမျက်နှာကနေ လိုချင်တဲ့အကြောင်းအရာတွေကို အလွယ်တကူရှာလို့ရသလိုပဲ **database** မှာလည်း **index** တွေက **data** ရှာဖွေတဲ့အခါ အထောက်အကူဖြစ်စေပါတယ်။

Index တစ်ခုဖန်တီးဖို့အတွက် **CREATE INDEX command** ကို အသုံးပြုပါတယ်။

```
CREATE INDEX idx_patient_registration
ON patients(registration_no);
```

ဒါကတော့ **patients table** ရဲ့ **registration_no column** အပေါ်မှာ **index** တစ်ခုဖန်တီးတာပါ။ **Index** နာမည်ကို "**idx_**" ဆိုတဲ့ **prefix** နဲ့ပေးလေ့ရှိပါတယ်။ **index** တစ်ခုဆိုတာကို အလွယ်တကူခွဲခြားသိနိုင်ဖို့ပါ။

Index တွေကိုအဓိကအားဖြင့် အောက်ပါအခြေအနေတွေမှာ အသုံးပြုလေ့ရှိပါတယ်။

Primary Key and Foreign Key
Primary Key တွေမှာ **MySQL** က အလိုအလျောက် **index** ဆောက်ပေးပါတယ်။ **Foreign key** တွေမှာတော့ **index** မပါလို့ အထူးသဖြင့် **JOIN operation** တွေမှာ နှေးကွေးနိုင်ပါတယ်။ ဒါကြောင့် **foreign key** တွေမှာ **index** ဆောက်ပေးသင့်ပါတယ်။

```
CREATE INDEX idx_visits_patient
ON visits(patient_id);

CREATE INDEX idx_visits_doctor
ON visits(doctor_id);
```

WHERE clause မှာ မကြာခဏအသုံးပြုတဲ့ column တွေ
ဥပမာ လူနာတွေကို မကြာခဏ အမည်နဲ့ရှာတယ်ဆိုရင် **full_name column** မှာ **index** ဆောက်ပေးသင့်ပါတယ်။

```
CREATE INDEX idx_patient_name
ON patients(full_name);
```

ORDER BY မှာ အသုံးများတဲ့ column တွေမှာ

Data တွေကို အစဉ်လိုက်စီတဲ့အခါ **index** ရှိနေရင် ပိုမြန်ပါတယ်။

```
CREATE INDEX idx_visit_date
ON visits(visit_date);
```

Index တွေနဲ့ပတ်သက်ပြီး သတိထားရမှာက **Index** တွေများလွန်းရင် **INSERT, UPDATE, DELETE operation** တွေမှာ နှေးကွေးသွားနိုင်ပါတယ်။ ဘာလို့လဲဆိုတော့ **data** တွေပြောင်းလဲတိုင်း **index** တွေကိုပါ **update** လုပ်ပေးရလို့ပါ။ **Column** တစ်ခုမှာ **unique value** တွေနည်းနေရင် **index** မဆောက်သင့်ပါဘူး။ ဥပမာ **gender** လိုမျိုး **column** မှာဆိုရင် **M, F, Other** ဆိုပြီး တန်ဖိုး ၃ခုပဲရှိပါတယ်။ ဒီလို **column** မျိုးမှာ **index** ဆောက်လို့ အကျိုးရှိမှာမဟုတ်ပါဘူး။ **Table** ထဲမှာ **row** အရေအတွက်နည်းနေရင်လည်း **index** မဆောက်သင့်ပါဘူး။ **Index** ဆောက်ရတဲ့ လုပ်အားက **table** တစ်ခုလုံးကို **scan** လုပ်တာထက် ပိုသုံးနေရတတ်ပါတယ်။ **Index** တစ်ခုဆောက်ပြီးတိုင်း **database** ရဲ့ **performance** ကောင်းသွားမှာ မဟုတ်ပါဘူး။ အရင်ဆုံး **query** အသုံးပြုပုံကို လေ့လာပြီးမှ **performance** အတွက် **index** လိုအပ်လားဆိုတာ ဆုံးဖြတ်သင့်ပါတယ်။

Index တွေကို ဖျက်ချင်ရင်တော့ **DROP INDEX command** ကို သုံးပါတယ်။

```
DROP INDEX idx_patient_registration
ON patients;
```

တစ်နည်းအားဖြင့် **index** တွေကို **database optimizer** ရဲ့ "မှတ်စုစာအုပ်" လို့သဘောထားလို့ရပါတယ်။ အရေးကြီးတဲ့ အချက်အလက်တွေကို **index** လို့ခေါ်တဲ့ မှတ်စုစာအုပ်ထဲမှာ မှတ်သားထားတဲ့အတွက် တစ်ခုခုရှာတဲ့အခါ **database table** တစ်ခုလုံးကို မလှန်ရတော့ပဲ မှတ်စုစာအုပ်ထဲကပဲ အမြန်ဆုံးရှာဖွေနိုင်တာပါ။

4.2 Database Administration Basics

4.2.1 User Management

database တွေမှာအချက်အလက်တွေထည့်သွင်းလိုက်တာနဲ့အမျှ အချက်အလက်တွေရဲ့လုံခြုံရေးကိုလည်းထည့်သွင်းစဉ်းစားရပါမယ်။ **databsae** ကိုဝင်ခွင့်ရှိသူတိုင်းကို လုပ်ပိုင်ခွင့်အလုံးစုံမပေးထားသင့်ပါဘူး။ တချို့ **user** တွေကို ကြည့်ရုံပဲပေးကြည့်ပြီး ပြင်တာ၊ ဖျက်တာ၊ အသစ်တိုးတာ လုပ်လို့မရအောင် ပိတ်ထားသင့်ပါတယ်။

MySQL မှာ **user** တွေကိုစီမံခန့်ခွဲဖို့အတွက် **CREATE USER, DROP USER,** နဲ့ **ALTER USER** ဆိုတဲ့ **command** တွေရှိပါတယ်။

```
CREATE USER 'nurse'@'localhost'
IDENTIFIED BY 'nurse123';
```

ဒီ **command** ဟာ **user** တစ်ယောက်ကို အခြေခံကျကျဖန်တီးလိုက်တာပါ။ **'nurse'@'localhost'** ဆိုတာ **nurse** ဆိုတဲ့နာမည်နဲ့ **user** တစ်ယောက်ကို **localhost** ကနေပဲ ချိတ်ဆက်ပြီးဝင်ခွင့်ရှိတယ်လို့ ဆိုလိုတာပါ။ **IDENTIFIED BY** နောက်က **nurse123** ကတော့ အဲဒီ **user** ရဲ့ **password** ဖြစ်ပါတယ်။

အကြောင်းအမျိုးမျိုးကြောင့် ရှိပြီးသား **user** တစ်ယောက်ကိုအသုံးပြုခွင့်ပိတ်ထားချင်တယ်ဆိုရင် **ACCOUNT LOCK** ကိုသုံးနိုင်ပါတယ်။ ဥပမာ အလုပ်ထွက်သွားတဲ့ဝန်ထမ်းတွေရဲ့ **account** တွေကို မဖျက်ဘဲ ပိတ်ထားချင်တဲ့အခါမျိုးမှာ အသုံးဝင်ပါတယ်။

```
ALTER USER 'nurse'@'localhost' ACCOUNT LOCK;
```

အဲဒီလိုမျိုးပဲ **lock** ပြန်ဖြည်ချင်တယ်ဆိုရင် **ACCOUNT UNLOCK** နဲ့ဖြည်ပေးလို့ရပါတယ်။

```
ALTER USER 'nurse'@'localhost' ACCOUNT UNLOCK;
```

User password ကိုပြန်ပြောင်းချင်ရင်တော့

```
ALTER USER 'nurse'@'localhost'
IDENTIFIED BY 'newpassword123';
```

User ကိုအပြီးဖျက်ချင်ရင်

```
DROP USER 'nurse'@'localhost';
```

လက်တွေ့မှာ **user** တစ်ယောက်ချင်းစီကို လုပ်ပိုင်ခွင့်အမျိုးမျိုးပေးထားရတတ်ပါတယ်။ ဥပမာ သူနာပြုတွေက လူနာမှတ်တမ်းအသစ်ထပ်ထည့်မယ်၊ ထည့်ပြီးသားတွေကို ကြည့်ခွင့်ရှိမယ်၊ ဖျက်လို့တော့မရအောင်ပိတ်ထားလို့ရပါမယ်။ ဆရာဝန်တွေကတော့ လူနာမှတ်တမ်းကို **CRUD** အားလုံးဖွင့်ပေးထားလို့ရပါတယ်။ **system admin** တွေကိုတော့ **database** တစ်ခုလုံး စီမံခွင့်ရှိပါတယ်။ ဒီလို လုပ်ပိုင်ခွင့်အမျိုးမျိုးကို **GRANT** နဲ့ **REVOKE command** တွေနဲ့ စီမံခန့်ခွဲနိုင်ပါတယ်။

4.2.2 GRANT and REVOKE

user တစ်ယောက်ကို လုပ်ပိုင်ခွင့်သတ်မှတ်ပေးတဲ့အခါ **GRANT** သုံးလို့ရပါတယ်။ **GRANT** လုပ်ပိုင်ခွင့်တွေထဲမှာ

SELECT - ကြည့်ခွင့်

INSERT - အသစ်ထည့်ခွင့်

UPDATE - ပြင်ခွင့်

DELETE - ဖျက်ခွင့်

ALL PRIVILEGES - အားလုံးခွင့်ပြု

ဥပမာ အနေနဲ့ **nurse user** ကို **data** ကြည့်၊ အသစ်ထည့်၊ ပြင်တာပေးလုပ်ပြီး ဖျက်လို့မရအောင် ဒီလိုသတ်မှတ်ပေးလို့ရပါတယ်။

```
GRANT SELECT, INSERT, UPDATE
ON emr.patients
TO 'nurse'@'localhost';

GRANT SELECT, INSERT, UPDATE
ON emr.visits
TO 'nurse'@'localhost';

GRANT SELECT, INSERT, UPDATE
ON emr.prescriptions
TO 'nurse'@'localhost';
```

ဒါဆိုရင် **nurse user** က **patients table** ထဲက လူနာမှတ်တမ်းတွေကြည့်လို့ရမယ်၊ လူနာစာရင်းအသစ်သွင်းလို့ရမယ်၊ လူနာအချက်အလက်တွေပြင်လို့ရမယ်၊ ဒါပေမယ့် လူနာမှတ်တမ်းတွေဖျက်လို့တော့မရပါဘူး။ တကယ်လို့ **nurse user** တစ်ယောက်က တခြားဌာနတစ်ခုခုကို ပြောင်းရွှေ့သွားမယ်ဆိုရင်တော့ သူ့ရဲ့လုပ်ပိုင်ခွင့်တွေကို **REVOKE command** နဲ့ပြန်ရုတ်သိမ်းလို့ရပါတယ်။

```
REVOKE UPDATE
ON emr.patients
FROM 'nurse'@'localhost';
```

ဒီလောက်က **patients table** ရဲ့ **UPDATE** လုပ်ပိုင်ခွင့်ကိုရုတ်သိမ်းလိုက်တာပါ။ ကြည့်ခွင့်၊ အသစ်ထည့်ခွင့်ကတော့ ဆက်ရှိနေဦးမှာဖြစ်ပါတယ်။
လုပ်ပိုင်ခွင့်အားလုံးကိုရုတ်သိမ်းချင်ရင်တော့

```
REVOKE ALL PRIVILEGES
ON emr.*
FROM 'nurse'@'localhost';
```

ဒါကတော့ **nurse user** တစ်ယောက်အတွက် **emr database** ထဲက **table** အားလုံးအတွက် လုပ်ပိုင်ခွင့်တွေကို ရုတ်သိမ်းလိုက်တာဖြစ်ပါတယ်။
လက်ရှိပေးအပ်ထားတဲ့ လုပ်ပိုင်ခွင့်တွေ ဘယ်လောက်ပေးထားတယ်ဆိုတာသိချင်ရင်တော့ **SHOW GRANTS command** သုံးနိုင်ပါတယ်။

```
SHOW GRANTS FOR 'nurse'@'localhost';
```

ဒါကတော့ **MySQL** အပိုင်းမှာ အခြေခံ **user** စီမံခန့်ခွဲပုံဖြစ်ပါတယ်။ လက်တွေ့မှာတော့ လုံခြုံရေးအရ လိုအပ်ချက်ပေါ်မူတည်ပြီး အဖွဲ့အစည်းအလိုက် **user management policy** ကို သတ်မှတ်ပေးရလေ့ရှိပါတယ်။ ဥပမာ **password** တွေကို နှစ်လတစ်ခါ မဖြစ်မနေပြောင်းခိုင်းတာ၊ **password** တွေကို ပြန်သုံးလို့မရအောင် တားမြစ်ထားတာ၊ **password** အနည်းဆုံး (၈) လုံးပါရမယ်၊ အကြီးအသေးရောရမယ်၊ ဂဏန်းပါရမယ်၊ **special character** ပါရမယ် စသဖြင့်သတ်မှတ်ပေးတာ၊ **user** တစ်ယောက် **login** ဝင်တာ (၅) ကြိမ်မှားသွားရင် **account lock** ချဖြစ်တာ တွေလုပ်နိုင်ပါတယ်။ **MySQL Enterprise edition** မှာဆိုရင်တော့ အဆင့်မြင့်တဲ့ **security features** တွေပါဝင်ပါတယ်။ အခမဲ့အသုံးပြုလို့ရတဲ့ **Community Edition** မှာတော့ အခုနမူနာပြထားတဲ့ **security feature** တွေပါဝင်ပါတယ်။

4.2.3 Backup and restore basics

Database backup လုပ်တယ်ဆိုတာ ဒေတာတွေကို အရန်သိမ်းဆည်းထားတာပါ။ လက်တွေ့အသုံးပြုနေတဲ့ **database** ဆိုရင် ပုံမှန် **backup** လုပ်ထားဖို့လိုပါတယ်။ **Backup** လုပ်ဖို့ဆိုရင် **logical backup** နဲ့ **physical backup** ဆိုတဲ့နည်းလမ်းနှစ်မျိုးရှိပါတယ်။ **Logical backup** က **SQL command** တွေနဲ့ **backup** လုပ်တာဖြစ်ပါတယ်။ **mysqldump command** နဲ့ **backup** လုပ်နိုင်ပါတယ်။ **Physical backup** က **database folder** တစ်ခုလုံးကို **backup** လုပ်တာပါ။ **MySQL enterprise backup** လိုမျိုး **tool** တွေသုံးနိုင်ပါတယ်။
mysqldump က **MySQL server** နဲ့အတူပါလာတဲ့ **command line tool** တစ်ခုပါ။

```
mysqldump -u root -p emr > emr_backup.sql
```

ဒါဆိုရင် **database** တစ်ခုလုံးကို **emr_backup.sql** နာမည်နဲ့ **backup** လုပ်ပေးပါလိမ့်မယ်။ **table** တစ်ခုချင်းစီရွေးပြီး **backup** လုပ်ချင်ရင်လည်း **table** နာမည်သတ်မှတ်ပေးလို့ရပါတယ်။

```
mysqldump -u root -p emr patients visits > patient_records.sql
```

Backup ဖိုင်တွေထဲမှာ **table structure** တွေရော၊ အထဲကဒေတာတွေရော ပါဝင်ပါတယ်။ **backup** လုပ်ထားတဲ့ ဖိုင်ကနေ **database** ထဲကိုပြန်ထည့်မယ်ဆိုရင် **restore command** နဲ့သွင်းလို့ရပါတယ်။

```
mysql -u root -p emr < emr_backup.sql
```

MySQL Workbench ကနေ **backup** လုပ်မယ်ဆိုရင်တော့ **Server Menu** ထဲက **Data Export** ကိုနှိပ်ပါ။ **Database** နဲ့ **table** တွေရွေးပြီး **export** လုပ်နိုင်ပါတယ်။ **Restore** လုပ်တဲ့အခါမှာလည်း **Server menu** ထဲက **Data Import** ကိုနှိပ်ပြီးသွင်းလို့ရပါတယ်။
ဒါက **manual export and import** လုပ်တာဖြစ်ပါတယ်။ လက်တွေ့သုံးတဲ့ပုံစံမှာတော့ အရေးကြီးတဲ့ **database** တွေကို **automated backup system** တွေနဲ့ ပုံမှန် **backup** ပြုလုပ်ပေးလေ့ရှိပါတယ်။ **Windows server** မှာဆိုရင် **Task Scheduler** ၊ **Linux server** မှာဆိုရင် **cron job** တွေသုံးကြပါတယ်။
Backup လုပ်ထားတဲ့ အရန်ဖိုင်တွေကို **external hard drive** တွေ၊ **cloud storage** တွေပေါ်မှာလည်းသိမ်းဆည်းထားနိုင်ပါတယ်။ ဥပမာ မီးဘေး၊ ရေဘေး စတဲ့သဘာဝဘေးအန္တရာယ်တွေ ကြုံလာရင်ဒေတာပျက်စီးဆုံးရှုံးမှုမရှိအောင် နေရာခွဲပြီးအရန်သိမ်းထားသင့်ပါတယ်။
Backup လုပ်တဲ့နေရာမှာ **full backup** နဲ့ **incremental backup** ဆိုပြီး ခွဲလုပ်လို့ရပါသေးတယ်။ **Full backup** က **database** တစ်ခုလုံးကို **backup** ယူတာပါ။ နေရာအများကြီးယူပေမယ့် **restore** ပြန်လုပ်ရတာလွယ်ပါတယ်။ **Incremental backup** မှာတော့ နောက်ဆုံး **backup** လုပ်ပြီးသလောက်ကနေ ပြောင်းလဲသွားတဲ့အပိုင်းကိုပဲ **backup** လုပ်တာပါ။ နေရာယူတာသက်သာပေမယ့် **restore** လုပ်တဲ့အခါ ပထမဆုံး **full backup** ရော၊ **incremental backup** တွေရော အကုန်ပြန် **restore** လုပ်ယူရပါတယ်။
Differential backup ဆိုတာကတော့ နောက်ဆုံး **full backup** လုပ်ပြီးချိန်ကနေ ပြောင်းလဲသွားတဲ့အပိုင်းတွေကိုပဲ **backup** လုပ်တာပါ။ **Incremental backup** ထက် နေရာနည်းနည်းပိုယူပေမယ့် **restore** လုပ်တဲ့အခါ **full backup** နဲ့ **differential backup** နှစ်ခုကိုပဲ ပြန်ယူရင်ရပါပြီ။
Backup ဘယ်လိုလုပ်မလဲစဉ်းစားကြတဲ့အခါ ဒေတာပမာဏဘယ်လောက်များမှာလဲ၊ **Disk space** ဘယ်လောက်ရှိလဲ၊ ဘယ်လောက်ကြာကြာသိမ်းထားမှာလဲ၊ **Restore** လုပ်တဲ့အခါ ဘယ်လောက်မြန်ဆန်ဖို့လိုလဲ စတာတွေထည့်သွင်းစဉ်းစားရပါမယ်။
နောက်ဆုံးအနေနဲ့ အခါအားလျော်စွာ **backup** ဖိုင်တွေကို **restore** ပြန်လုပ်ကြည့်ရပါမယ်။ ဒါမှတကယ်အရေးကြုံတဲ့အခါ **restore** လုပ်ရာမှာမမျှော်လင့်ထားတဲ့ အခက်အခဲမျိုးစုံကြုံရတာမျိုး ကာကွယ်ထားနိုင်မှာပါ။

4.2.4 Basic performance optimization tips

Database တစ်ခုမှာ အချက်အလက်တွေများလာနဲ့အမျှ **performance** နှေးကွေးလာတတ်ပါတယ်။ ဒါကြောင့် **database optimization** အပိုင်းကိုလည်း ပြုလုပ်ပေးရပါမယ်။

Indexing
နှေးကွေးနေတဲ့ **query** တွေကို **index** ထည့်လိုက်ရုံနဲ့ သိသိသာသာမြန်သွားတတ်ပါတယ်။ ဒါပေမယ့် **index** တစ်ခုထည့်တိုင်း **performance** တိုးမလာတတ်ပါဘူး။ သင့်တော်တဲ့ **column** တွေမှာပဲ **index** ထည့်ရပါမယ်။ ဥပမာ လူနာတွေကိုရှာတဲ့နေရာမှာ အများဆုံးသုံးတဲ့ **column** က နာမည်ပဲဆိုရင် **full_name column** မှာ **index** ထည့်သင့်ပါတယ်။ ခါတ်ခွဲစစ်ဆေးမှုတွေကို နေ့စွဲအလိုက်ရှာတာများရင် **date column** မှာ **index** ထည့်သင့်ပါတယ်။

Query optimization
query ရေးတဲ့နေရာမှာ **SELECT *** နဲ့မဟုတ်ဘဲ လိုတဲ့ **column** တွေပဲရွေးပြီး **select** လုပ်ပါ။ **subquery** တွေအစား **JOIN** သုံးပါ။ **LIKE** '%text%' လိုသုံးတာမျိုး တတ်နိုင်သလောက်ရှောင်ပါ။ **ORDER BY, GROUP BY** တွေမှာ **function** ခေါ်သုံးတာရှောင်ပါ။ ဥပမာ

```
SELECT * FROM patients
WHERE YEAR(date_of_birth) = 1990;

SELECT * FROM patients
WHERE date_of_birth >= '1990-01-01'
AND date_of_birth <= '1990-12-31';
```

ပထမ **query** မှာ **data_of_birth column** က **index** ထည့်ထားရင်တောင် **YEAR() function** သုံးထားတာမို့လို့ **indexing** ကိုအသုံးမချနိုင်ပါဘူး။ ဒုတိယ **query** ကတော့ **date range** နဲ့ **filter** လုပ်ထားတာမို့လို့ **indexing** ကိုအသုံးချနိုင်ပါတယ်။

Table Optimization

Table တွေကို **DELETE** လုပ်တာများရင် **disk space** ပိုယူတတ်ပါတယ်။ ဒါကြောင့် အခါအားလျော်စွာ **optimize** လုပ်ပေးသင့်ပါတယ်။

```
OPTIMIZE TABLE patients;
```

Server Configuration

MySQL server configuration ထဲမှာ အရေးပါတဲ့ **parameter** တွေဖြစ်တဲ့ **innodb_buffer_pool_size** ဟာ **InnoDB engine** အတွက် **memory** ဘယ်လောက်သုံးမလဲ၊ **max_connection** မှာ တပြိုင်နက် **connection** ဘယ်နှခုလက်ခံမလဲ၊ **query_cache_size** က **query cache** အတွက် **memory** ဘယ်လောက်သုံးမလဲ သတ်မှတ်ပေးထားနိုင်ပါတယ်။

Slow Query Log

အချိန်ကြာကြာယူရတဲ့ **query** တွေကို **log** မှတ်တမ်းနဲ့စောင့်ကြည့်ထားမယ်ဆိုရင် ဘယ် **query** ကိုသွက်လက်အောင်ပြင်ဖို့လိုမလဲ သိနိုင်ပါတယ်။

```
SET GLOBAL slow_query_log = 'ON';
SET GLOBAL long_query_time = 2;
```

ဒါဆိုရင် နှစ်စက္ကန့်ထက်ပိုအချိန်ယူတဲ့ **query** တွေကို **log file** ထဲမှာမှတ်တမ်းတင်ပေးသွားပါလိမ့်မယ်

Hardware Resources

MySQL ဆာဗာထိုင်ထားတဲ့ကွန်ပြူတာရဲ့ **RAM, CPU, Storage** စတာတွေကလည်း **performance** ကောင်းဖို့ အရေးကြီးပါတယ်။ **RAM** များများပါရင် **disk I/O** နည်းပြီး **performance** ပိုကောင်းပါတယ်။ **SSD storage** ဆိုရင် **HDD** ထက်ပိုမြန်မှာပါ။ **Multi-core CPU** နဲ့ဆိုရင် **parallel query execution** လုပ်နိုင်တာမို့လို့ ပိုမြန်ပါတယ်။

Performance optimization ဟာ တခါတည်းနဲ့ပြီးသွားတဲ့အလုပ်မဟုတ်တာကြောင့် သူ့အတွက်သီးသန့်ပုံမှန်အချိန်တစ်ခုဖယ်ထားပြီး အပတ်စဉ်၊ လစဉ်လုပ်သွားဖို့လိုပါတယ်။

CHAPTER 5
FINAL PROJECT AND BEST PRACTICES

5 Chapter 5 - Final Project and Best Practices

5.1 Creating a simple Electronic Medical Record

ဒီအပိုင်းမှာ လူနာမှတ်တမ်းစနစ် (Electronic Medical Record) လေးတစ်ခုကို SQL အသုံးပြုပြီးဘယ်လိုတည်ဆောက်မလဲဆိုတာ လေ့လာကြပါမယ်။

ဆေးရုံဆေးခန်းနဲ့ ကျန်းမာရေးစောင့်ရှောက်မှုပေးတဲ့နေရာတွေမှာ ဆေးကုသမှုမှတ်တမ်းကို စနစ်တကျသိမ်းဆည်းထားတဲ့ ဒေတာဘေ့စ်ကို Electronic Medical Record - EMR လို့ခေါ်ပါတယ်။

အောက်ပါ table တွေတည်ဆောက်ကြည့်ပါမယ်

patients - ကျန်းမာရေးစောင့်ရှောက်မှုခံယူသူတွေရဲ့ကိုယ်ရေးအချက်အလက်တွေသိမ်းမယ်

visits - ကျန်းမာရေးစောင့်ရှောက်မှုခံယူတဲ့အခါတိုင်း မှတ်တမ်းတင်မယ်

prescriptions - ဆေးညွှန်းတွေကိုမှတ်တမ်းတင်မယ်

lab_results - ဓါတ်ခွဲစစ်ဆေးရလဒ်တွေကို သိမ်းဆည်းမယ်

5.1.1 Patients table

Reg No	Full Name	Birth Date	Gender	Address	Phone
REG001	Aung Aung	1990-05-15	M	No.123, Baho Road, Yangon	09123456789
REG002	Ma Ma	1985-08-22	F	No.45, Anawrahta Road, Mandalay	09234567890
REG003	Ko Ko	1995-03-10	M	No.67, Strand Road, Yangon	09345678901

ဒီလို table တစ်ခုရလာအောင် အောက်ပါ command နဲ့ဖန်တီးပါမယ်။

```
CREATE TABLE patients (
    patient_id INT PRIMARY KEY AUTO_INCREMENT,
    registration_no VARCHAR(20) UNIQUE NOT NULL,
    full_name VARCHAR(100) NOT NULL,
    date_of_birth DATE NOT NULL,
    gender ENUM('M', 'F', 'Other') NOT NULL,
    address TEXT,
    phone_number VARCHAR(20),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

1. patient_id
   လူတိုင်းအတွက် unique ဖြစ်တဲ့ နံပါတ်ဖြစ်ပါတယ်။ INT အမျိုးအစားမို့လို့ ကိန်းပြည့်တွေပဲသိမ်းပါမယ်။ patient_id က database operation တွေအတွက်ဖြစ်ပြီး reporting မှာမမြင်ရပါဘူး။ reporting မှာအဓိကသုံးတဲ့ ကိုယ်ပိုင်နံပါတ်က registration_no ဖြစ်ပါတယ်။

PRIMARY KEY constraint သတ်မှတ်ပေးထားပါတယ်။

AUTO_INCREMENT ပေးထားလို့ **patient** တစ်ယောက်တိုးတိုင်း +1 အလိုအလျောက်တိုးပေးသွားမှာပါ။

2. **registration_no**
   ဆေးရုံဆေးခန်းသုံးမှတ်ပုံတင်နံပါတ်ဖြစ်ပါတယ်။ **database operation** တွေအတွက်သုံးမှာမဟုတ်ပါဘူး။ စာလုံးရေ (၂၀) လုံးအထိ သိမ်းလို့ရတဲ့ **VARCHAR(20)** အမျိုးအစားသတ်မှတ်ပေးထားပါတယ်။ လူတစ်ယောက်နဲ့တစ်ယောက် နံပါတ်မထပ်အောင် **UNIQUE()** လုပ်ထားပါတယ်။ နံပါတ်မဖြည့်ဘဲ ဒေတာလာသိမ်းလို့မရအောင် **NOT NULL** ပေးထားပါတယ်။

3. **full_name**
   နာမည်အပြည့်အစုံထည့်သိမ်းပါမယ်။ စာလုံးရေ (၁၀၀) ပါတဲ့ **string** တန်ဖိုးတွေသိမ်းပါမယ်။ လူနာမည်မပါဘဲ သိမ်းလို့မရအောင် **NOT NULL** ပေးထားပါတယ်။

4. **date_of_birth**
   DATE မို့လို့ နေ့စွဲ **format** နဲ့သိမ်းပါတယ်။ **NOT NULL** ပေးထားပါတယ်။

5. **gender**
   ENUM() ထဲမှာ **M, F, Other** ပေးထားတဲ့အတိုင်းဖြည့်လို့ရပါမယ်။ **NOT NULL** ပေးထားပါတယ်။

6. **address**
   စာသား (၂၅၅) လုံးထက်ပိုသိမ်းလို့ရအောင် **TEXT** ပေးထားပါတယ်။

7. **phone_number**
   ဂဏန်း ၂၀ လုံးအထိပါတဲ့ **string** တန်ဖိုးသိမ်းပါမယ်

8. **created_at**
   TIMESTAMP မို့လို့ အချိန်နဲ့နေ့စွဲတန်ဖိုးတွေသိမ်းပါမယ်။ **DEFAULT CURRENT_TIMESTAMP** ဆိုတာက ဒေတာလာသိမ်းတဲ့အချိန်ကို အလိုအလျောက်သိမ်းဆည်းပေးပါတယ်။

5.1.2 Doctors table

ID	Full Name	Specialty	License No	Phone Number	Email
1	Dr. Thiha	Cardiologist	LIC001	09111222333	dr.thiha@email.com
2	Dr. Aye Aye	Pediatrician	LIC002	09222333444	dr.ayeaye@email.com
3	Dr. Kyaw Min	General Physician	LIC003	09333444555	dr.kyawmin@email.com
4	Dr. Su Su	Dermatologist	LIC004	09444555666	dr.susu@email.com

doctors table မှာ ကျန်းမာရေးစောင့်ရှောက်မှုပေးသူတွေရဲ့အချက်အလက်တွေသိမ်းဆည်းထားပါမယ်။ **doctors table** ကို ဒီ **command** နဲ့ဖန်တီးပါမယ်။

```
CREATE TABLE doctors (
    doctor_id INT PRIMARY KEY AUTO_INCREMENT,
    full_name VARCHAR(100) NOT NULL,
    specialty VARCHAR(100) NOT NULL,
    license_number VARCHAR(50) UNIQUE NOT NULL,
```

```
    phone_number VARCHAR(20),
    email VARCHAR(100),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

1. **doctor_id**
   INT အမျိုးအစားဖြစ်ပြီး **PRIMARY KEY** တာဝန်ယူပေးပါတယ်။ **AUTO_INCREMENT** လုပ်ထားတဲ့အတွက် အလိုအလျောက် တိုးသွားမှာဖြစ်ပါတယ်

2. **full_name**
   နာမည်အပြည့်အစုံသိမ်းလို့ရအောင် **VARCHAR(100)** ပေးထားပါတယ်။ **NOT NULL constraint** ထည့်ထားတဲ့အတွက် မဖြစ်မနေ ထည့်သွင်းပေးရမှာ ဖြစ်ပါတယ်

3. **specialty**
   အထူးပြုဘာသာရပ်အတွက် **VARCHAR(100)** ပေးထားပါတယ်။ **NOT NULL constraint** ပါဝင်ပါတယ်။

4. **license_number**
   ကျန်းမာရေးစောင့်ရှောက်မှုလိုင်စင်နံပါတ်အတွက် **VARCHAR(50)** ပေးထားပါတယ်။ **NOT NULL** နဲ့ **UNIQUE constraint** နှစ်ခုလုံးပါဝင်တဲ့အတွက် မဖြစ်မနေထည့်ရမယ်၊ ထပ်လည်းမထပ်ရပါဘူး

5. **phone_number**
   ဖုန်းနံပါတ်အတွက် **VARCHAR(20)** ပေးထားပြီး မထည့်ရင်လည်းချန်ခဲ့လို့ရတဲ့ **Optional field** ဖြစ်ပါတယ်

6. **email**
   VARCHAR(100) ဖြစ်ပြီး **Optional field** ဖြစ်ပါတယ်

7. **created_at**
   Record ထည့်သွင်းလိုက်တဲ့ အချိန်ကို **DEFAULT CURRENT_TIMESTAMP** သတ်မှတ်ထားတဲ့အတွက် အလိုအလျောက်ထည့်ပေးသွားမှာ ဖြစ်ပါတယ်

5.1.3 Visits table

Visits

visit_id	patient_id	doctor_id	visit_date	chief_complaint	diagnosis	notes
1	101	201	2024-03-15 09:30	Fever and cough	URTI	Prescribed antibiotics
2	102	202	2024-03-15 10:15	Headache	Migraine	Follow up in 2 weeks
3	103	203	2024-03-15 11:00	Annual checkup	Healthy	Normal vitals
4	104	201	2024-03-15 13:45	Joint pain	Arthritis	Referred to specialist

လူနာတွေရဲ့ဆေးခန်းလာပြတဲ့ မှတ်တမ်းတွေကို **visits table** မှာ သိမ်းဆည်းသွားမှာ ဖြစ်ပါတယ်။ **visits table** ကို အောက်က **command** နဲ့ ဖန်တီးပါမယ်။

```sql
CREATE TABLE visits (
    visit_id INT PRIMARY KEY AUTO_INCREMENT,
    patient_id INT NOT NULL,
    doctor_id INT NOT NULL,
    visit_date DATETIME NOT NULL,
    chief_complaint TEXT NOT NULL,
    diagnosis TEXT,
    notes TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (patient_id) REFERENCES patients(patient_id),
    FOREIGN KEY (doctor_id) REFERENCES doctors(doctor_id)
);
```

1. **visit_id**
   INT အမျိုးအစားဖြစ်ပြီး **PRIMARY KEY** တာဝန်ယူပေးပါတယ်။ **AUTO_INCREMENT** လုပ်ထားတဲ့အတွက် အလိုအလျောက် တိုးသွားမှာဖြစ်ပါတယ်

2. **patient_id**
   INT အမျိုးအစားဖြစ်ပြီး **NOT NULL constraint** ပါဝင်ပါတယ်။ **patients table** နဲ့ချိတ်ဆက်ထားတဲ့ **FOREIGN KEY** လည်းဖြစ်ပါတယ်

3. **doctor_id**
   INT အမျိုးအစားဖြစ်ပြီး **NOT NULL constraint** ပါဝင်ပါတယ်။ **doctors table** နဲ့ချိတ်ဆက်ထားတဲ့ **FOREIGN KEY** လည်းဖြစ်ပါတယ်

4. **visit_date**
   လူနာလာရောက်ပြသတဲ့ နေ့ရက်အချိန်အတွက် **DATETIME** အမျိုးအစားသုံးထားပြီး **NOT NULL constraint** ပါဝင်ပါတယ်

5. **chief_complaint**
   လာပြရတဲ့အဓိကအကြောင်းရင်းသိမ်းဖို့ **TEXT** အမျိုးအစားသုံးထားပြီး **NOT NULL constraint** ပါဝင်ပါတယ်

6. **diagnosis**
   ရောဂါရှာဖွေတွေ့ရှိချက်တွေအတွက် **TEXT** အမျိုးအစားသုံးထားပြီး **Optional field** ဖြစ်ပါတယ်

7. **notes**
   နောက်ထပ်မှတ်ချက်တွေအတွက် **TEXT** အမျိုးအစားသုံးထားပြီး **Optional field** ဖြစ်ပါတယ်

8. **created_at**
   Record ထည့်သွင်းလိုက်တဲ့ အချိန်ကို **DEFAULT CURRENT_TIMESTAMP** သတ်မှတ်ထားတဲ့အတွက် အလိုအလျောက်ထည့်ပေးသွားမှာ ဖြစ်ပါတယ်

5.1.4 Prescriptions table

prescription_id	visit_id	medication_name	dosage	frequency	duration	notes
1	101	Amoxicillin	500mg	TDS	7 days	Take after meals
2	102	Paracetamol	650mg	QID	5 days	If fever persists
3	103	Omeprazole	20mg	OD	14 days	Before breakfast
4	104	Metformin	850mg	BD	30 days	With meals

ဒါကတော့ **prescriptions table** ရဲ့နမူနာပုံစံဖြစ်ပါတယ်။ လူနာတွေအတွက်ဆေးညွှန်းအချက်အလက်တွေကို သိမ်းဆည်းထားပါမယ်။ အောက်ပါ **command** နဲ့ပြုလုပ်ပါမယ်။

```
CREATE TABLE prescriptions (
    prescription_id INT PRIMARY KEY AUTO_INCREMENT,
    visit_id INT NOT NULL,
    medication_name VARCHAR(100) NOT NULL,
    dosage VARCHAR(50) NOT NULL,
    frequency VARCHAR(50) NOT NULL,
    duration VARCHAR(50) NOT NULL,
    notes TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (visit_id) REFERENCES visits(visit_id)
);
```

1. **prescription_id**
   INT အမျိုးအစားဖြစ်ပြီး **PRIMARY KEY** တာဝန်ယူပေးပါတယ်။ **AUTO_INCREMENT** လုပ်ထားတဲ့အတွက် အလိုအလျောက်တိုးသွားမှာဖြစ်ပါတယ်။

2. **visit_id**
   လူနာလာပြတဲ့မှတ်တမ်းနဲ့ချိတ်ဆက်ဖို့အတွက် **INT** အမျိုးအစားပေးထားပါတယ်။ **NOT NULL constraint** ထည့်ထားတဲ့အတွက် မဖြစ်မနေ ထည့်သွင်းပေးရမှာ ဖြစ်ပါတယ်။ **visits table** နဲ့ချိတ်ဆက်ဖို့အတွက် **FOREIGN KEY** ဖြစ်ပါတယ်။

3. **medication_name**
   ဆေးနာမည်အတွက် **VARCHAR(100)** ပေးထားပါတယ်။ **NOT NULL constraint** ပါဝင်ပါတယ်။ လက်တွေ့သုံးနေတဲ့ဒေတာဘေ့စ်တွေမှာတော့ ဆေးနာမည်တွေအတွက် သီးသန့် **table** တစ်ခုထပ်ခွဲလေ့ရှိပါတယ်။ အခုကသင်ခန်းစာဖြစ်တဲ့အတွက် ဆေးနာမည်ကိုပဲ တိုက်ရိုက်လာသိမ်းလိုက်ပါမယ်။

4. **dosage**
   ဆေးပမာဏအတွက် **VARCHAR(50)** ပေးထားပါတယ်။ **NOT NULL constraint** ပါဝင်ပါတယ်။

5. **frequency**
   တစ်နေ့ကို ဘယ်နှစ်ကြိမ်သောက်ရမလဲဆိုတာအတွက် **VARCHAR(50)** ပေးထားပါတယ်။ **NOT NULL constraint** ပါဝင်ပါတယ်။

6. **duration**
   ဘယ်လောက်ကြာကြာသောက်ရမလဲဆိုတာအတွက် **VARCHAR(50)** ပေးထားပါတယ်။ **NOT NULL constraint** ပါဝင်ပါတယ်။

7. **notes**
   မှတ်ချက်တွေအတွက် **TEXT** အမျိုးအစားပေးထားပြီး **Optional field** ဖြစ်ပါတယ်။

8. **created_at**
   Record ထည့်သွင်းလိုက်တဲ့ အချိန်ကို **DEFAULT CURRENT_TIMESTAMP** သတ်မှတ်ထားတဲ့အတွက် အလိုအလျောက်ထည့်ပေးသွားမှာ ဖြစ်ပါတယ်။

5.1.5 Lab Results table

Lab Results

lab_id	visit_id	test_name	test_result	reference_range	performed_at
1	101	Blood Glucose	95 mg/dL	70-100 mg/dL	2024-01-15 09:30:00
2	102	Cholesterol	180 mg/dL	<200 mg/dL	2024-01-15 10:15:00
3	103	Hemoglobin	14.2 g/dL	13.5-17.5 g/dL	2024-01-15 11:00:00
4	104	WBC Count	7.5 K/uL	4.5-11.0 K/uL	2024-01-15 11:45:00

```
CREATE TABLE lab_results (
    lab_id INT PRIMARY KEY AUTO_INCREMENT,
    visit_id INT NOT NULL,
          test_name VARCHAR(100) NOT NULL,
    test_result TEXT NOT NULL,
    reference_range VARCHAR(100),
    performed_at DATETIME NOT NULL,
    notes TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (visit_id) REFERENCES visits(visit_id)
);
```

lab_results table မှာ ဓာတ်ခွဲခန်းစမ်းသပ်မှုရလဒ်တွေကို သိမ်းဆည်းထားမှာဖြစ်ပါတယ်။ **lab_results table** ကို အပေါ်က **command** နဲ့ ဖန်တီးပါတယ်။

1. **lab_id**
   INT အမျိုးအစားဖြစ်ပြီး **PRIMARY KEY** တာဝန်ယူပေးပါတယ်။ **AUTO_INCREMENT** လုပ်ထားတဲ့အတွက် အလိုအလျောက် တိုးသွားမှာဖြစ်ပါတယ်

2. **visit_id**
   INT အမျိုးအစားဖြစ်ပြီး **NOT NULL constraint** ထည့်ထားတဲ့အတွက် မဖြစ်မနေ ထည့်သွင်းပေးရမှာ ဖြစ်ပါတယ်။ **visits table** နဲ့ချိတ်ဆက်ထားတဲ့ **FOREIGN KEY** လည်းဖြစ်ပါတယ်

3. **test_name**
   စမ်းသပ်စစ်ဆေးမှုအမည်အတွက် **VARCHAR(100)** ပေးထားပါတယ်။ **NOT NULL constraint** ထည့်ထားတဲ့အတွက် မဖြစ်မနေ ထည့်သွင်းပေးရမှာ ဖြစ်ပါတယ်

4. **test_result**
   စမ်းသပ်စစ်ဆေးမှုရလဒ်အတွက် **TEXT** ပေးထားပါတယ်။ **NOT NULL constraint** ထည့်ထားတဲ့အတွက် မဖြစ်မနေ ထည့်သွင်းပေးရမှာ ဖြစ်ပါတယ်

5. **reference_range**
   ရလဒ်များနှိုင်းယှဉ်ကြည့်ရှုနိုင်ရန်အတွက် သတ်မှတ်ထားသော **range** ဖြစ်ပြီး **VARCHAR(100)** ပေးထားပါတယ်။ **Optional field** ဖြစ်ပါတယ်

6. **performed_at**
   စမ်းသပ်စစ်ဆေးမှုပြုလုပ်တဲ့အချိန် သိမ်းဆည်းဖို့ **DATETIME** အမျိုးအစားပေးထားပါတယ်။ **NOT NULL constraint** ထည့်ထားတဲ့အတွက် မဖြစ်မနေ ထည့်သွင်းပေးရမှာ ဖြစ်ပါတယ်

7. **notes**
   မှတ်ချက်များအတွက် **TEXT** အမျိုးအစားပေးထားပြီး **Optional field** ဖြစ်ပါတယ်

8. **created_at**
   Record ထည့်သွင်းလိုက်တဲ့ အချိန်ကို **DEFAULT CURRENT_TIMESTAMP** သတ်မှတ်ထားတဲ့အတွက် အလိုအလျောက်ထည့်ပေးသွားမှာ ဖြစ်ပါတယ်

5.1.6 Creating tables in MySQL workbench

table တွေအကြောင်းသိသွားပြီဆိုရင် **MySQL Workbench** မှာ **table** တွေဖန်တီးကြပါမယ်။

MySQL Workbench ကိုဖွင့်ပြီး **MySQL Server** ကို **connect** လုပ်ပေးပါ။

Toolbar ထဲက **Create New Schema** ခလုတ်ကို နှိပ်ပါ။ ဒါမှမဟုတ် **Navigator panel** ထဲက **schema tab** မှာ **right-click** နှိပ်ပြီး **Create Schema** ကိုရွေးပါ။ **Schema name** ကို **emr** လို့ပေးပြီး **Apply** နှိပ်ပေးပါ။ **Navigation panel schema tab** ထဲမှာ **emr schema** ပေါ်လာပါလိမ့်မယ်။ **right click** နှိပ်ပြီး **Set as default schema** ကိုရွေးပါ။ ရှေ့လျှောက် **run** မယ့် **sql command** တွေဟာ ဒီ **schema** မှာပဲဖြစ်မယ့်လို့သတ်မှတ်ပေးလိုက်တာဖြစ်ပါတယ်။

ပြီးရင်ဘေးက **Query window** ထဲမှာ **table** ဖန်တီးတဲ့ **command** တွေထည့်သွင်းပြီး **run** ပေးပါ။ **table** တစ်ခုချင်းစီ **create** လုပ်လို့ရသလို အားလုံးတပြိုင်နက်ထည့် **run** လို့လည်းရပါတယ်။

Table တွေအားလုံးကို ဖန်တီးပြီးသွားရင် **Navigation panel** ထဲက **emr** အောက်က **Tables** မှာ **right click > refresh All** ရွေးပါ။ **Tables** ခေါင်းစဉ်အောက်မှာ **table** ၅ ခုစလုံးကို မြင်ရမှာဖြစ်ပါတယ်။ **Table** တစ်ခုချင်းစီဘေးက **info icon** နှိပ်ပြီး **structure** တွေပြန်စစ်ဆေးလို့ရပါတယ်။

ဒါဆိုရင် **Electronic Medical Record** စနစ်အတွက် အခြေခံကျတဲ့ **database structure** တစ်ခုရပြီဖြစ်ပါတယ်။ ဆက်လက်ပြီး **data** တွေထည့်သွင်းကြည့်ပါမယ်။

5.1.7 Inserting data in MySLQ workbench

Patients table

patients table ထဲ အောက်ပါ command နဲ့ ဒေတာတွေထည့်သွင်းနိုင်ပါတယ်။

```
INSERT INTO patients (registration_no, full_name, date_of_birth, gender, address, phone_n
umber)
VALUES
('REG001', 'Aung Aung', '1990-05-15', 'M', 'No.123, Baho Road, Yangon', '09123456789'),
('REG002', 'Ma Ma', '1985-08-22', 'F', 'No.45, Anawrahta Road, Mandalay', '09234567890'),
('REG003', 'Ko Ko', '1995-03-10', 'M', 'No.67, Strand Road, Yangon', '09345678901');
```

ဒီ command ဟာပုံပါအတိုင်း ဒေတာတွေဝင်သွားမှာဖြစ်ပါတယ်။

Reg No	Full Name	Birth Date	Gender	Address	Phone
REG001	Aung Aung	1990-05-15	M	No.123, Baho Road, Yangon	09123456789
REG002	Ma Ma	1985-08-22	F	No.45, Anawrahta Road, Mandalay	09234567890
REG003	Ko Ko	1995-03-10	M	No.67, Strand Road, Yangon	09345678901

registration_no က REG နဲ့စတဲ့ unique နံပါတ်ဖြစ်ပြီး လူနာတိုင်းမှာ တစ်ယောက်နဲ့တစ်ယောက် မတူနိုင်ပါဘူး။

အသက်အရွယ်တွက်ချက်နိုင်အောင် date_of_birth ကို YYYY-MM-DD format နဲ့သိမ်းထားပါတယ်။

gender က M (ကျား)၊ F (မ)၊ Other (အခြား) ဆိုပြီး ၃မျိုးထဲက တစ်မျိုးမျိုးဖြစ်ရပါမယ်။

address နဲ့ phone_number တွေကတော့ optional fields တွေဖြစ်တဲ့အတွက် မဖြစ်မနေထည့်စရာမလိုပါဘူး။

created_at က အချက်အလက်ထည့်သွင်းလိုက်တဲ့အချိန်ကို system က အလိုအလျောက်ထည့်ပေးသွားမှာဖြစ်ပါတယ်။

Doctors table

doctors table ထဲကို ဒီ command နဲ့ထည့်သွင်းနိုင်ပါတယ်။

```
INSERT INTO doctors (full_name, specialty, license_number, phone_number, email)
VALUES
('Dr. Thiha', 'Cardiologist', 'LIC001', '09111222333', 'dr.thiha@email.com'),
('Dr. Aye Aye', 'Pediatrician', 'LIC002', '09222333444', 'dr.ayeaye@email.com'),
('Dr. Kyaw Min', 'General Physician', 'LIC003', '09333444555', 'dr.kyawmin@email.com'),
('Dr. Su Su', 'Dermatologist', 'LIC004', '09444555666', 'dr.susu@email.com');
```

Full Name	Specialty	License No	Phone Number	Email
Dr. Thiha	Cardiologist	LIC001	09111222333	dr.thiha@email.com
Dr. Aye Aye	Pediatrician	LIC002	09222333444	dr.ayeaye@email.com
Dr. Kyaw Min	General Physician	LIC003	09333444555	dr.kyawmin@email.com
Dr. Su Su	Dermatologist	LIC004	09444555666	dr.susu@email.com

doctor_id ကို **auto_increment** လုပ်ထားတဲ့အတွက် ထည့်သွင်းစရာမလိုပါဘူး။ **system** က အလိုအလျောက် တိုးပေးသွားမှာ ဖြစ်ပါတယ်။

full_name နဲ့ **specialty** တို့က **NOT NULL** ဖြစ်တဲ့အတွက် မဖြစ်မနေထည့်ပေးရမှာ ဖြစ်ပါတယ်။

license_number က **unique** ဖြစ်တဲ့အတွက် ဆရာဝန်တစ်ယောက်နဲ့တစ်ယောက် ထပ်လို့မရပါဘူး။

phone_number နဲ့ **email** တွေကတော့ **optional fields** တွေဖြစ်တဲ့အတွက် မဖြစ်မနေထည့်စရာမလိုပါဘူး။

created_at က အချက်အလက်ထည့်သွင်းလိုက်တဲ့အချိန်ကို **system** က အလိုအလျောက်ထည့်ပေးသွားမှာဖြစ်ပါတယ်။

Visits table

visits table ထဲ ဒီ **command** နဲ့ထည့်သွင်းနိုင်ပါတယ်။

```
INSERT INTO visits (patient_id, doctor_id, visit_date, chief_complaint, diagnosis, notes)
VALUES
(1, 1, '2024-03-25 09:30:00', 'Chest pain and shortness of breath', 'Angina', 'Patient ad
vised to take rest and prescribed medication'),
(2, 2, '2024-03-25 10:15:00', 'Fever and cough', 'Upper respiratory tract infection', 'Fo
llow up after 3 days if symptoms persist'),
(3, 3, '2024-03-25 14:00:00', 'Headache and dizziness', 'Migraine', 'Patient advised to a
void triggers and take prescribed medication'),
(1, 4, '2024-03-26 11:30:00', 'Skin rash', 'Contact dermatitis', 'Prescribed antihistamin
e and topical cream');
```

Visit ID	Patient ID	Doctor ID	Visit Date	Chief Complaint	Diagnosis	Notes
1	1	1	2024-03-25 09:30:00	Chest pain and shortness of breath	Angina	Patient advised to take rest
2	2	2	2024-03-25 10:15:00	Fever and cough	Upper respiratory tract infection	Follow up after 3 days if symptoms persist
3	3	3	2024-03-25 14:00:00	Headache and dizziness	Migraine	Patient advised to avoid triggers
4	1	4	2024-03-26 11:30:00	Skin rash	Contact dermatitis	Prescribed antihistamine and topical crean

visit_id ကို **auto_increment** လုပ်ထားတဲ့အတွက် ထည့်သွင်းစရာမလိုပါဘူး။ **system** က အလိုအလျောက် တိုးပေးသွားမှာ ဖြစ်ပါတယ်။

patient_id နဲ့ **doctor_id** တွေက **foreign key** တွေဖြစ်တဲ့အတွက် သက်ဆိုင်ရာ **table** တွေမှာ ရှိပြီးသား **ID** တွေကိုပဲ အသုံးပြုလို့ရပါတယ်။

visit_date က **DATETIME format** နဲ့ သိမ်းထားတဲ့အတွက် နေ့စွဲနဲ့ အချိန်ပါ ထည့်သွင်းပေးရပါတယ်။

chief_complaint က **NOT NULL** ဖြစ်တဲ့အတွက် မဖြစ်မနေထည့်ပေးရမှာ ဖြစ်ပါတယ်။

diagnosis နဲ့ **notes** တွေကတော့ **optional fields** တွေဖြစ်တဲ့အတွက် မဖြစ်မနေထည့်စရာမလိုပါဘူး။

created_at က အချက်အလက်ထည့်သွင်းလိုက်တဲ့အချိန်ကို **system** က အလိုအလျောက်ထည့်ပေးသွားမှာဖြစ်ပါတယ်။

Prescriptions table

```
INSERT INTO prescriptions (visit_id, medication_name, dosage, frequency, duration, notes)
VALUES
      (1, 'Nitroglycerin', '0.4mg', 'As needed', '30 days', 'Take when chest pain occurs'
),
(2, 'Amoxicillin', '500mg', 'Three times daily', '5 days', 'Take after meals'),
(3, 'Sumatriptan', '50mg', 'As needed', '30 days', 'Take at onset of migraine'),
(4, 'Cetirizine', '10mg', 'Once daily', '7 days', 'Take in the morning');
```

Prescription ID	Visit ID	Medication Name	Dosage	Frequency	Duration	Notes
1	1	Nitroglycerin	0.4mg	PRN	30 days	Take when chest pain occurs
2	2	Amoxicillin	500mg	TDS	5 days	Take after meals
3	3	Sumatriptan	50mg	PRN	30 days	Take at onset of migraine
4	4	Cetirizine	10mg	OD	7 days	Take in the morning

visit_id က **foreign key** ဖြစ်ပြီး **visits table** နဲ့ချိတ်ဆက်ထားတဲ့အတွက် **visits table** ထဲမှာရှိပြီးသား **ID** ဖြစ်ဖို့လိုပါတယ်။

Lab Results table

လူနာတွေရဲ့ဓာတ်ခွဲခန်းစစ်ဆေးမှု မှတ်တမ်းတွေကို **lab_results table** ထဲ ထည့်သွင်းကြည့်ရအောင်။

```
INSERT INTO lab_results (visit_id, test_name, test_result, reference_range, performed_at,
 notes)
VALUES
(1, 'ECG', 'Normal sinus rhythm', '60-100 bpm', '2024-03-25 10:00:00', 'No significant ST
-T wave changes'),
(1, 'Troponin-I', '0.02 ng/mL', '< 0.04 ng/mL', '2024-03-25 10:30:00', 'Within normal lim
its'),
(2, 'Complete Blood Count', 'WBC: 12,000/µL', '4,500-11,000/µL', '2024-03-25 11:00:00', '
Slightly elevated WBC count'),
(3, 'Blood Pressure', '130/85 mmHg', '120/80 mmHg', '2024-03-25 14:30:00', 'Slightly elev
ated');
```

Lab ID	Visit ID	Test Name	Test Result	Reference Range	Performed At	Notes
1	1	ECG	Normal sinus rhythm	60-100 bpm	2024-03-25 10:00:00	No significant ST-T wave changes
2	1	Troponin-I	0.02 ng/mL	0.04 ng/mL	2024-03-25 10:30:00	Within normal limits
3	2	Complete Blood Count	WBC: 12,000/µL	4,500-11,000/µL	2024-03-25 11:00:00	Slightly elevated WBC count
4	3	Blood Pressure	130/85 mm	120/80 mmHg	2024-03-25 14:30:00	Slightly elevated

5.1.8 Testing basic queries

table (၅) ခုဖန်တီးပြီး ဒေတာတွေထည့်သွင်းပြီးတဲ့နောက်မှာ ထည့်သွင်းထားတဲ့ဒေတာတွေကို လိုသလိုပြန်လည်ထုတ်ကြည့်နိုင်တဲ့ **SQL query** တွေအကြောင်းဆက်ဖော်ပြပါမယ်။

SELECT

ပထမဆုံး **patients table** ထဲက လူနာအားလုံးရဲ့စာရင်းကို **SQL query** နဲ့ကြည့်ကြည့်ရအောင်။

```
SELECT * FROM patients;
```

patient_id	registration_no	full_name	date_of_birth	gender	address	phone_number
1	REG001	Aung Aung	1990-05-15	M	No.123, Baho Road, Yangon	09123456789
2	REG002	Ma Ma	1985-08-22	F	No.45, Anawrahta Road, Mandalay	09234567890
3	REG003	Ko Ko	1995-03-10	M	No.67, Strand Road, Yangon	09345678901

ဒါကတော့ **patients table** ထဲက အချက်အလက်အားလုံးကို ပြတဲ့ **query** ရဲ့ရလဒ်ပဲဖြစ်ပါတယ်။ အခုလို **column** တွေအားလုံးမဟုတ်ဘဲ တချို့ကိုပဲရွေးထုတ်ချင်ရင်လည်း **SELECT** နောက်မှာ **column** နာမည်တွေ ရေးပေးလို့ရပါတယ်။

```
SELECT registration_no, full_name, gender FROM patients;
```

registration_no	full_name	gender
REG001	Aung Aung	M
REG002	Ma Ma	F
REG003	Ko Ko	M

ဒါဆိုရင် ကိုယ်လိုချင်တဲ့ **column** တွေကိုပဲ ရွေးထုတ်ပြီးကြည့်လို့ရပါပြီ။ အလားတူပဲ ဆရာဝန်တွေရဲ့စာရင်းကိုလည်း **doctors table** ကနေကြည့်ကြည့်ရအောင်။

```
SELECT full_name, specialty, phone_number FROM doctors;
```

full_name	specialty	phone_number
Dr. Thiha	Cardiologist	09111222333
Dr. Aye Aye	Pediatrician	09222333444
Dr. Kyaw Min	General Physician	09333444555
Dr. Su Su	Dermatologist	09444555666

WHERE

ဒီတစ်ခါတော့ ကိုယ်လိုချင်တဲ့ဒေတာကိုပဲရွေးထုတ်ဖို့ **WHERE** နဲ့စစ်ထုတ်ကြည့်ရအောင်။

```
SELECT full_name, visit_date, chief_complaint
FROM visits v
JOIN patients p ON v.patient_id = p.patient_id
WHERE p.full_name = 'Aung Aung';
```

full_name	visit_date	chief_complaint
Aung Aung	2024-03-25 09:30:00	Chest pain and shortness of breath
Aung Aung	2024-03-26 11:30:00	Skin rash

ဒီ **query** မှာ **visits table** နဲ့ **patients table** ကို **JOIN** လုပ်ထားပါတယ်။ ဘာလို့လဲဆိုတော့ **visits table** မှာ လူနာအမည်မပါလို့ပါ။ **visits table** မှာ **patient_id** ပဲပါတဲ့အတွက် **patients table** နဲ့ချိတ်ပြီး လူနာရဲ့အမည် ယူထားတာပဲဖြစ်ပါတယ်။ **WHERE clause** နဲ့ **Aung Aung** ဆိုတဲ့အမည်ရှိတဲ့ လူနာရဲ့ဆေးခန်းလာပြတဲ့ မှတ်တမ်းတွေကိုပဲ ရှာဖွေထားပါတယ်။ ဒါဆိုရင် **Aung Aung** မှာ ဆေးခန်းလာပြတာ နှစ်ကြိမ်ရှိတာကို တွေ့ရပါတယ်။

နောက်ထပ် **WHERE clause** နဲ့ပတ်သက်တဲ့ **query** တစ်ခုထပ်ရေးကြည့်ရအောင်။

```
SELECT
    p.full_name,
    v.visit_date,
    pr.medication_name,
    pr.dosage,
    pr.frequency
FROM visits v
JOIN patients p ON v.patient_id = p.patient_id
JOIN prescriptions pr ON v.visit_id = pr.visit_id
WHERE DATE(v.visit_date) = '2024-03-25';
```

full_name	visit_date	medication_name	dosage	frequency
Aung Aung	2024-03-25 09:30:00	Nitroglycerin	0.4mg	As needed
Ma Ma	2024-03-25 10:15:00	Amoxicillin	500mg	Three times daily
Ko Ko	2024-03-25 14:00:00	Sumatriptan	50mg	As needed

ဒီ **query** မှာတော့ လူနာတစ်ယောက်ချင်းစီရဲ့ဆေးညွှန်းတွေကိုကြည့်ဖို့ **patients, visits** နဲ့ **prescriptions table** သုံးခုကို **JOIN** လုပ်ထားပါတယ်။ **WHERE clause** နဲ့ မတ်လ ၂၅ ရက်နေ့က ညွှန်းတဲ့ဆေးတွေကိုပဲ ရွေးထုတ်ကြည့်ထားပါတယ်။

DATE() function ကိုသုံးရတာက **visit_date column** က **datetime** အမျိုးအစားဖြစ်နေလို့ပါ။ အချိန်ပါတဲ့ရက်စွဲတွေကို နှိုင်းယှဉ်တဲ့အခါ **DATE() function** နဲ့ ရက်စွဲကိုပဲဆွဲထုတ်ပြီး နှိုင်းယှဉ်လေ့ရှိပါတယ်။

ORDER BY

ဒါကတော့ ဆရာဝန်တွေရဲ့စာရင်းကို အထူးပြုဘာသာရပ်အလိုက် အစဉ်လိုက်စီပြီး ကြည့်တာပါ။

```
SELECT full_name, specialty, phone_number
FROM doctors
ORDER BY specialty ASC;
```

full_name	specialty	phone_number
Dr. Thiha	Cardiologist	09111222333
Dr. Su Su	Dermatologist	09444555666
Dr. Kyaw Min	General Physician	09333444555
Dr. Aye Aye	Pediatrician	09222333444

ORDER BY specialty ASC ဆိုတာ **specialty column** ကို **alphabetical order** အတိုင်း **ascending (A** ကနေ **Z)** အစဉ်လိုက်စီပါဆိုတဲ့အဓိပ္ပါယ်ပါ။ **ASC** အစား **DESC** လို့ရေးရင် အစဉ်ပြောင်းပြန် **(Z** ကနေ **A)** စီသွားပါလိမ့်မယ်။ **ORDER BY** နဲ့ **column** တစ်ခုထက်ပို စီချင်ရင်လည်းရပါတယ်။ ဥပမာ **ORDER BY specialty ASC, full_name DESC** လို့ရေးရင် **specialty** နဲ့အရင်စီပြီး **specialty** အတူတူတွေကို **full_name** ပြောင်းပြန် **Z** ကနေ **A** စီပေးသွားပါလိမ့်မယ်။

GROUP BY နဲ့ **column** တစ်ခုချင်းစီအလိုက် အုပ်စုလိုက်စုစည်းပြီး **aggregate functions** တွေနဲ့ စိတ်ဝင်စားဖွယ်ရာ အချက်အလက်တွေကို ဘယ်လိုဆွဲထုတ်ကြည့်လို့ရလဲဆိုတာ နောက်အပိုင်းမှာ ဆက်လက်လေ့လာကြပါမယ်။

GROUP BY and HAVING

ဥပမာ ဆရာဝန်တိုင်းက တစ်နေ့တာအတွင်း လူနာဘယ်နှစ်ယောက်စီ ကြည့်လဲ သိချင်တယ်ဆိုပါစို့။ **GROUP BY** နဲ့ဒီလိုရေးလို့ရပါတယ်။

```
SELECT
    d.full_name,
    DATE(v.visit_date) AS visit_day,
```

```
    COUNT(*) AS patient_count
FROM visits v
JOIN doctors d ON v.doctor_id = d.doctor_id
GROUP BY d.full_name, DATE(v.visit_date)
ORDER BY visit_day, d.full_name;
```

Doctor Name	Visit Day	Patient Count
Dr. Aye Aye	2024-03-25	1
Dr. Kyaw Min	2024-03-25	1
Dr. Su Su	2024-03-26	1
Dr. Thiha	2024-03-25	1

Subqueries

WHERE clause ထဲမှာ သုံးတဲ့ subquery

ပျမ်းမျှထက်ပိုပြီး လူနာကြည့်ပေးနိုင်တဲ့ဆရာဝန်တွေရဲ့စာရင်းကို ကြည့်ပါမယ်။

```
SELECT full_name, specialty
FROM doctors
WHERE doctor_id IN (
    SELECT doctor_id
    FROM visits
    GROUP BY doctor_id
    HAVING COUNT(*) > (
        SELECT AVG(patient_count)
        FROM (
            SELECT COUNT(*) as patient_count
            FROM visits
            GROUP BY doctor_id
        ) as avg_visits
    )
);
```

SELECT clause ထဲမှာ သုံးတဲ့ subquery

ဆရာဝန်တစ်ယောက်ချင်းစီရဲ့ လူနာအရေအတွက်နဲ့ ပျမ်းမျှလူနာအရေအတွက်တို့ကို နှိုင်းယှဉ်ကြည့်ပါမယ်။

```
SELECT
    d.full_name,
    COUNT(*) as total_patients,
    (SELECT AVG(patient_count)
     FROM (
         SELECT COUNT(*) as patient_count
         FROM visits
         GROUP BY doctor_id
```

```
    ) as avg_visits) as avg_patients
FROM visits v
JOIN doctors d ON v.doctor_id = d.doctor_id
GROUP BY d.doctor_id, d.full_name;
```

Doctor Name	Total Patients	Avg Patients
Dr. Thiha	5	3.5
Dr. Aye Aye	3	3.5
Dr. Kyaw Min	2	3.5

ဒါကနမူနာမြင်ရတဲ့ **table view** ပါ။ လက်ရှိထည့်ထားလက်စ ဒေတာအတိုင်းဆိုရင် **total_patient 1** နဲ့ **avarage patient 1.0** ပဲဖြစ်နေဦးမှာပါ။

FROM clause ထဲမှာ သုံးတဲ့ subquery

လူနာတွေကို အသက်အုပ်စုအလိုက် စုစည်းပြီး လေ့လာကြည့်ပါမယ်။

```
SELECT
    age_group,
    COUNT(*) as patient_count
FROM (
    SELECT
        CASE
            WHEN TIMESTAMPDIFF(YEAR, date_of_birth, CURDATE()) < 18 THEN 'Child'
            WHEN TIMESTAMPDIFF(YEAR, date_of_birth, CURDATE()) < 60 THEN 'Adult'
            ELSE 'Elderly'
        END as age_group
    FROM patients
) as patient_ages
GROUP BY age_group;
```

Age Group	Patient Count
Child	1
Adult	2

ဒါက နမူနာပြထားတဲ့ **table view** ပါ။ လက်ရှိထည့်လက်စ ဒေတာအတိုင်းဆိုရင် **Adult 3** ပဲပြနေပါဦးမယ်။

5.1.9 Views

လူနာတွေရဲ့အခြေခံအချက်အလက်တွေပေါ်မှာ အခြေခံပြီး **view** တစ်ခုဆောက်ကြည့်ပါမယ်

```
CREATE VIEW patient_summary AS
SELECT
```

```
    p.registration_no,
    p.full_name,
    p.gender,
    TIMESTAMPDIFF(YEAR, p.date_of_birth, CURDATE()) as age,
    COUNT(v.visit_id) as visit_count,
    MAX(v.visit_date) as last_visit
FROM patients p
LEFT JOIN visits v ON p.patient_id = v.patient_id
GROUP BY p.patient_id, p.registration_no, p.full_name, p.gender, p.date_of_birth;
```

ပြီးရင် **view** ကိုပြန်ခေါ် ကြည့်ပါ

```
SELECT * FROM patient_summary;
```

5.1.10 Using MySQL built-in functions

MySQL မှာ အသင့်သုံး **function** တွေအများကြီးပါဝင်ပါတယ်။ အသုံးများတဲ့ **function** တွေကို အမျိုးအစားလိုက်ခွဲပြီး လေ့လာကြည့်ရအောင်။

String functions

String function တွေက စာသားတွေကိုကိုင်တွယ်တဲ့နေရာမှာ တော်တော်အလေးအသုံးဝင်ပါတယ်။

CONCAT() - စာသားတွေကို တွဲစပ်ပေးပါတယ်

SUBSTRING() - စာသားထဲက လိုတဲ့အပိုင်းကို ဖြတ်ထုတ်ပါတယ်

UPPER() - စာလုံးအသေးကနေ အကြီးပြောင်းပါတယ်

LOWER() - စာလုံးအကြီးကနေ အသေးပြောင်းပါတယ်

LENGTH() - စာသားအရှည်ကို ရေတွက်ပါတယ်

TRIM() - စာသားရဲ့ ရှေ့နဲ့နောက်က **space** တွေကို ဖြတ်ထုတ်ပါတယ်

ဥပမာ

```
SELECT
    CONCAT(full_name, ' (', specialty, ')') as doctor_info,
    UPPER(specialty) as specialty_caps,
    LENGTH(full_name) as name_length
FROM doctors;
```

Doctor Info	Specialty Caps	Length
Dr. Thiha (Cardiologist)	CARDIOLOGIST	9
Dr. Aye Aye (Pediatrician)	PEDIATRICIAN	10

Numeric functions

ကိန်းဂဏန်းတန်ဖိုးတွေကို ကိုင်တွယ်တဲ့အခါ အသုံးဝင်တဲ့ function တွေပါ။

ROUND() - ဒဿမနေရာ သတ်မှတ်ပေးထားသလောက်ပဲထားပြီး အနီးဆုံးဒဿမကိန်းယူပါတယ်

CEILING() - အနီးဆုံးကိန်းပြည့်အထိ ပေါင်းပါတယ်

FLOOR() - အနီးဆုံးကိန်းပြည့်အထိ နှုတ်ပါတယ်

ABS() - တန်ဖိုးရဲ့အပေါင်းကိန်းကိုယူပါတယ်

MOD() - စားလို့ကျန်တဲ့ တန်ဖိုးကိုယူပါတယ်

Numeric function စမ်းဖို့အတွက် temporary table တစ်ခုပြုလုပ်ပါမယ်

```
CREATE TEMPORARY TABLE lab_results_temp (
    test_id INT,
    test_name VARCHAR(50),
    test_result DECIMAL(10,4)
);
```

temporary table ထဲကို data ထည့်ပါမယ်

```
INSERT INTO lab_results_temp (test_id, test_name, test_result) VALUES
    (1, 'Glucose', 5.6783),
    (2, 'Cholesterol', 4.2987),
    (3, 'Hemoglobin', 14.7823),
    (4, 'Vitamin D', 29.9999),
    (5, 'Creatinine', 0.8456);
```

ပြီးရင်တော့ numeric function တွေသုံးပြီးစမ်းကြည့်ပါမယ်။

```
SELECT
    test_name,
    ROUND(test_result, 2) as rounded_result,
    CEILING(test_result) as ceiling_result,
    FLOOR(test_result) as floor_result
FROM lab_results_temp;
```

test_name	rounded_result	ceiling_result	floor_result
Glucose	5.68	6	5
Cholesterol	4.30	5	4
Hemoglobin	14.78	15	14
Vitamin D	30.00	30	29
Creatinine	0.85	1	0

Date and time functions

နေ့စွဲတွေ၊ အချိန်တွေကိုင်တွယ်တဲ့အခါမှာ အသုံးဝင်ပါတယ်။

NOW() - လက်ရှိ နေ့စွဲနဲ့အချိန်ကိုယူပါတယ်

CURDATE() - လက်ရှိ နေ့စွဲကိုယူပါတယ်

CURTIME() - လက်ရှိ အချိန်ကိုယူပါတယ်

DATE() - datetime ထဲက နေ့စွဲကိုပဲ ဆွဲထုတ်ပါတယ်

YEAR(), MONTH(), DAY() - နေ့စွဲထဲက နှစ်၊လ၊ရက် ကို ဆွဲထုတ်ပါတယ်

DATEDIFF() - နေ့စွဲနှစ်ခုကြား ကွာခြားချက်ကို တွက်ပါတယ်

DATE_ADD(), DATE_SUB() - နေ့စွဲတစ်ခုကို ပေါင်း၊နှုတ် လုပ်ပါတယ်

```sql
SELECT
    full_name,
    date_of_birth,
    TIMESTAMPDIFF(YEAR, date_of_birth, CURDATE()) as age,
    DATE_ADD(date_of_birth, INTERVAL 1 YEAR) as next_birthday
FROM patients;
```

Full Name	Date of Birth	Age	Next Birthday
Aung Aung	1990-05-15	34	1991-05-15
Ma Ma	1985-08-22	39	1986-08-22
Ko Ko	1995-03-10	29	1996-03-10

Control flow functions

Query ထဲမှာ condition တွေစစ်ပြီး condition အလိုက်ဘာသာပြန်ပြစေချင်တဲ့အခါ အသုံးဝင်ပါတယ်။

IF() - condition တစ်ခုပေါ်မူတည်ပြီး တန်ဖိုးတစ်ခုပြောင်းယူပါတယ်

CASE - condition အများအပြားစစ်ပြီး တန်ဖိုးတစ်ခုရွေးယူပါတယ်

IFNULL() - NULL တန်ဖိုးဖြစ်နေရင် အခြားတန်ဖိုးတစ်ခုနဲ့အစားထိုးပါတယ်

COALESCE() - NULL မဟုတ်တဲ့ ပထမဆုံးတန်ဖိုးကို ယူပါတယ်

```sql
SELECT
    full_name,
    CASE
        WHEN TIMESTAMPDIFF(YEAR, date_of_birth, CURDATE()) < 18 THEN 'Minor'
        WHEN TIMESTAMPDIFF(YEAR, date_of_birth, CURDATE()) < 60 THEN 'Adult'
        ELSE 'Senior'
    END as age_group,
    IFNULL(phone_number, 'No Phone') as contact
FROM patients;
```

full_name	age_group	contact
Aung Aung	Adult	09123456789
Ma Ma	Adult	09234567890
Ko Ko	Adult	09345678901

လက်ရှိ data ကို query ခေါ် ကြည့်ရင်ဒီလိုမြင်ရပါမယ်။ အသက် (၁၈) နှစ်အောက်၊ အသက် (၆၀) အထက်၊ ဖုန်းနံပါတ်မရှိတဲ့ record တွေထပ်ထည့်ကြည့်ပါ။

Function တွေကို အသုံးပြုတဲ့အခါ သတိထားရမယ့်အချက်တွေကတော့ Function တွေများလွန်းရင် query ရဲ့ performance ကို ထိခိုက်နိုင်ပါတယ်။ လိုအပ်တာကိုပဲ သုံးသင့်ပါတယ်။ NULL တန်ဖိုးတွေနဲ့ function တွေ အလုပ်လုပ်တဲ့အခါ သတိထားရပါမယ်။ ဥပမာ CONCAT() function မှာ parameter တစ်ခုက NULL ဖြစ်နေရင် result ကလည်း NULL ပဲဖြစ်သွားပါမယ်။ Date/Time တန်ဖိုးတွေနဲ့အလုပ်လုပ်တဲ့အခါ time zone setting ကို သတိထားရပါမယ်။ ကိုယ့်ကွန်ပြူတာထဲမှာ MySQL workbench က ကိုယ့် timezone နဲ့ပြနေတာဆိုပေမယ့် အွန်လိုင်ဆာဗာပေါ် deploy လုပ်လိုက်ပြီဆိုရင်တော့ အဲဒီဆာဗာရဲ့ timezone အတိုင်းတွက်ပေးမှာပါ။ ဒါကြောင့် US မှာထိုင်တဲ့ဆာဗာပေါ်တင်လိုက်ပြီဆိုရင် အရှေ့အာရှ timezone ထက်တစ်ရက်နောက်ကျတဲ့ရက်တွေ ပြပေးပါလိမ့်မယ်။

String function တွေမှာ character set နဲ့ collation တွေကို သတိထားရပါမယ်။ မြန်မာစာလိုမျိုး unicode စာသားတွေနဲ့ အလုပ်လုပ်တဲ့အခါ အထူးသတိထားရပါမယ်။

Function တွေကို သင့်တော်သလို အသုံးပြုခြင်းအားဖြင့် database query တွေကို ပိုမိုထိရောက်အောင် ရေးနိုင်ပါတယ်။ သို့သော် function တွေကို မလိုအပ်ဘဲ အလွန်အကျွံသုံးစွဲခြင်းက database performance ကို ထိခိုက်စေနိုင်တာကို သတိချပ်သင့်ပါတယ်။

5.2 Writing efficient queries

ဒီအပိုင်းမှာ လက်တွေ့အသုံးများတဲ့ query တွေကို ရေးကြည့်ပါမယ်။ အရင်သင်ခန်းတွေမှာပါတဲ့ optimize ဘယ်လိုလုပ်ရမလဲဆိုတဲ့ ဗဟုသုတတွေကိုပါ ထည့်သွင်းအသုံးချကြည့်ပါမယ်။ လက်တွေ့မှာ server side programming language တွေဖြစ်တဲ့ Python, Java, Node.js, PHP, Ruby စတဲ့ code တွေမှာ SQL script တွေထည့်သွင်းရေးရပါတယ်။ Object-relational Mapping (ORM) နည်းပညာပါတဲ့ framework တွေ ဥပမာ Laravel (PHP), Django (Python), Prisma library (Javascript) တွေသုံးမယ်ဆိုရင်တော့ SQL script တွေတိုက်ရိုက်ရေးစရာမလိုပါဘူး။ သက်ဆိုင်ရာ language ရေးထုံးတွေအတိုင်း ရေးရုံနဲ့ RAW SQL script နဲ့ query လုပ်ပေးပါတယ်။ framework တွေ၊ ORM library တွေသုံးပေမယ့်လည်း ရံဖန်ရံခါ project လိုအပ်ချက်အရ RAW SQL query တွေကြားညှပ်ရေးရတတ်တာမို့လို့ ထိရောက်တဲ့ SQL query တွေရေးတတ်ဖို့ဆိုတာ ချန်ထားလို့မရတဲ့ skill တစ်ခုဖြစ်နေပါသေးတယ်။

Searching Patients

ဒီဥပမာမှာ လူနာနာမည်ကို ရှာဖွေတဲ့ **query** ကိုရေးကြည့်ပါမယ်။ ဆေးရုံဆေးခန်းမှာ ရက်ချိန်းလာပြန်ပြတဲ့အခါ နာမည်မေးမယ်၊ နာမည်ရိုက်ပြီးရှာပေးရပါမယ်။ လူတိုင်းကကိုယ့် **registration number** ကိုမမှတ်မိသလို ရက်ချိန်းလာပြတဲ့အခါမှာလည်း ဆေးမှတ်တမ်းစာအုပ် မပါလာတတ်ကြတာမို့လို့ နာမည်မေးပြီးပြန်ရှာရတာများပါတယ်။ ဒီလိုပြန်ရှာတဲ့အခါ **SELECT *** နဲ့ **query** လုပ်တာထက် အောက်ပါ **query** နဲ့ရှာကြည့်ပါမယ်။

```
SELECT
            p.patient_id,
            p.registration_no,
            p.full_name,
            p.date_of_birth,
            p.phone_number,
            DATE_FORMAT(p.date_of_birth, '%Y-%m-%d') as dob,
            TIMESTAMPDIFF(YEAR, p.date_of_birth, CURDATE()) as age
FROM patients p
WHERE p.full_name LIKE 'Aung%'
ORDER BY p.date_of_birth DESC;
```

patient_id	registration_no	full_name	date_of_birth	phone_number	dob	age
1	REG001	Aung Aung	1990-05-15	09123456789	1990-05-15	34

ဒါဆိုရင် နာမည် **Aung** နဲ့စတဲ့လူအားလုံးကိုရှာပေးပါမယ်။ အသက်အငယ်ဆုံးကနေ အစဉ်လိုက်ပြပေးသွားပါမယ်။ **Registration number** ၊ နာမည်၊ အသက်နဲ့ ဖုန်းနံပါတ်တွေ ပါပါမယ်။ **patient_id** က **CRUD operation** တွေအတွက် အရေးကြီးလို့ ထည့်ထားတာပါ။ **patient record** တစ်ခုကို **CRUD** ဆက်လုပ်မယ်ဆိုရင် သက်ဆိုင်ရာ **programming language** ကနေ **patient_id** ကို **page** တခုနဲ့တခုကြား **parameter** အဖြစ်သယ်သွားပြီးလိုအပ်တဲ့ **function** တွေ ဆက်ရေးပေးရမှာဖြစ်ပါတယ်။

အဲဒီမှာရက်ချိန်းလာပြန်ပြတဲ့လူရဲ့အချက်အလက်တွေနဲ့ကိုက်ညီတဲ့ **record** တစ်ခုရှာတွေ့ရင် ဒီလုပ်ငန်းပြီးပါပြီ။ ကျန်တဲ့ **CRUD** အပိုင်းတွေက သက်ဆိုင်ရာ **programming language** မှာ ဆက်ရေးသွားပါလိမ့်မယ်။

Viewing a patient's visit history

ရက်ချိန်းလာပြတဲ့လူတစ်ယောက်ကို စာရင်းထဲမှာပြန်ရှာတွေ့ပြီဆိုရင် သူ့ရဲ့ကျန်းမာရေးစောင့်ရှောက်မှုမှတ်တမ်းကို ပြန်ကြည့်ရပါမယ်။ ပုဂ္ဂိုလ်ရေးအချက်အလက်တွေက **patients table** မှာသိမ်းပြီး ကျန်းမာရေးမှတ်တမ်းက **visits table** မှာသိမ်းတယ်။ ကြည့်ပေးတဲ့ဆရာဝန်ရဲ့အချက်အလက်က **doctors table** မှာဖြစ်ပြီး ဆေးညွှန်းတွေက **prescriptions table** ၊ ဓါတ်ခွဲစမ်းသပ်အဖြေတွေက **lab_results** မှာသိမ်းပါတယ်။ ဒါကြောင့် လူတစ်ယောက်ချင်းစီကိုအသေးစိတ်ကြည့်မယ့် စာမျက်နှာမှာ အဲဒီ **table** အားလုံးက ချိတ်ဆက်နေတဲ့ **record** တွေကိုခေါ်ပြရပါမယ်။

ဒီနေရာမှာအချုပ်က **patient_id** ဖြစ်ပါတယ်။ **patient_id** တန်ဖိုးရရုံနဲ့ **table** ပေါင်းစုံက **relationship** ဖြစ်နေတဲ့ **record** တွေကိုဆွဲထုတ်လို့ရသွားတာမို့လို့ပါ။

```
SELECT
      p.registration_no,
```

```
    p.full_name,
    v.visit_date,
    d.full_name as doctor_name,
    v.diagnosis,
    pr.medication_name,
    pr.dosage,
    pr.frequency,
    l.test_name,
    l.test_result
FROM patients p
JOIN visits v ON p.patient_id = v.patient_id -- visits table နဲ့ join
JOIN doctors d ON v.doctor_id = d.doctor_id -- doctors table နဲ့ join
LEFT JOIN prescriptions pr ON v.visit_id = pr.visit_id -- ဆေးညွှန်းကအမြဲမရှိနိုင်လို့ left join
LEFT JOIN lab_results l ON v.visit_id = l.visit_id -- ဓါတ်ခွဲစစ်ဆေးအမြဲမရှိနိုင်လို့ left join
WHERE p.patient_id = 1 -- patient_id နံပါတ်က search ရှာတွေ့တဲ့ record ကနေ ဒီကိုလှမ်းပို့မယ်
ORDER BY v.visit_date DESC; -- နောက်ဆုံးလာတဲ့ရက်ကိုထိပ်ဆုံးစီမယ်
```

Registration	Full Name	Visit Date	Doctor Name	Diagnosis	Medication	Dosage	Frequency	Test Name	Test Result
REG001	Aung Aung	2024-03-26 11:30:00	Dr. Su Su	Contact dermatitis	Cetirizine	10mg	Once daily	NULL	NULL
REG001	Aung Aung	2024-03-25 09:30:00	Dr. Thiha	Angina	Nitroglycerin	0.4mg	As needed	ECG	Normal sinus rhythm
REG001	Aung Aung	2024-03-25 09:30:00	Dr. Thiha	Angina	Nitroglycerin	0.4mg	As needed	Troponin-I	0.02 ng/mL

date တွေကို ascending စီရင် အစောဆုံးရက်ကို ထိပ်ဆုံးပြပါတယ်။ descending ဆိုရင် နောက်ဆုံးရက်ကို ထိပ်ဆုံးပြပါတယ်။

patient_id တစ်ခုမှာ visits နဲ့ doctors record တွေအမြဲရှိနေမှာမို့လို့ ရိုးရိုး JOIN နဲ့ချိတ်ပြီး ဆေးညွှန်းမပါတဲ့လူ၊ ဓါတ်ခွဲစစ်ဆေးမှုမလုပ်တဲ့လူတွေလည်း ရှိနိုင်တာမို့လို့ prescriptions နဲ့ lab_results တို့ကို LEFT JOIN နဲ့ချိတ်ပါတယ်။ နမူနာအရ patient_id ကို 1 နဲ့ filter လုပ်ထားတာဆိုပေမယ့် ဒီတန်ဖိုးက dynamic ဖြစ်ပါမယ်။ သက်ဆိုင်ရာ programming language ကနေ search လုပ်တဲ့စာမျက်နှာမှာ click နှိပ်လိုက်တဲ့ record ကနေ patient_id တန်ဖိုးဆွဲယူပြီး patient visit history စာမျက်နှာဆီကို ပို့ပေးပါလိမ့်မယ်။

Getting monthly patient count per diagnosis

report လိုအပ်ချက်အရ လအလိုက် ရောဂါအမျိုးအစားတစ်ခုချင်းစီ လူနာအရေအတွက်သိချင်တယ်ဆိုပါစို့။ subquery တွေမသုံးဘဲ Common table expression (CTE) သုံးပြီး ယာယီတခါသုံး table ပြုလုပ်ပါမယ်။

```
WITH MonthlyStats AS ( -- တခါသုံး table နာမည်
    SELECT
        DATE_FORMAT(v.visit_date, '%Y-%m') as month, -- date ထဲက နှစ်နဲ့လပဲဆွဲထုတ်
        v.diagnosis,
        COUNT(*) as case_count -- record count ရေတွက်
```

```
    FROM visits v
    WHERE v.visit_date >= DATE_SUB(CURDATE(), INTERVAL 1 MONTH) -- ဒီနေ့ကနေပြီးခဲ့တဲ့တစ်လအတွင်း
    GROUP BY
        DATE_FORMAT(v.visit_date, '%Y-%m'),
        v.diagnosis
) -- တခါသုံး table ဒီမှာပြီး။ အောက်မှာသူ့ကို query ဆက်လုပ်မယ်
SELECT
    month,
    diagnosis,
    case_count,
    ROUND(case_count * 100.0 / SUM(case_count) OVER (PARTITION BY month), 2) as percentag
e
    -- အဲဒီလမှာ ဒီရောဂါနဲ့လာပြတဲ့ ရာခိုင်နှုန်း
FROM MonthlyStats
ORDER BY month DESC, case_count DESC;
```

month	diagnosis	case_count	percentage
2024-03	Angina	1	25.00
2024-03	Upper respiratory tract infection	1	25.00
2024-03	Migraine	1	25.00
2024-03	Contact dermatitis	1	25.00

ဒီ **query** မှာတော့ **subquery** တွေမသုံးဘဲ **common table expression (CTE)** သုံးပါတယ်။ **CTE** ကိုအလွယ်မှတ်မယ်ဆိုရင်တော့ ယာယီတခါသုံး **table** ပါ။ ဒီ **query run** နေတဲ့အချိန်ပဲရှိပါတယ်။ **query** ပြီးရင်မရှိတော့ပါဘူး။

DATE_SUB(CURDATE(), INTERVAL 1 MONTH) function က ဒီနေ့ကနေ လွန်ခဲ့တဲ့ တစ်လအတွင်း **record** တွေကိုရှာပေးပါတယ်။ လောလောဆယ်လွန်ခဲ့တဲ့တစ်လအတွင်း **record** တွေမရှိသေးရင် **INTERVAL MONTH** ကိုတိုးပြီး **query** လုပ်ကြည့်ပါ။

ဒုတိယပိုင်း **query** မှာ **CTE** က **alias column** ခေါင်းစဉ်လေးတွေပဲဆက်ယူသုံးလို့ရသွားပါပြီ။ **case_count** ဆိုတဲ့ **alias** ခေါင်းစဉ်ကိုလည်း **calculation formula** တွေ တန်းသုံးလို့ရသွားပါပြီ။ ရာခိုင်နှုန်းတွက်တဲ့နေရာမှာနှစ်ပိုင်းပါပါတယ်။ **case_count * 100.0 / SUM(case_count) OVER (PARTITION BY month)** က **window function** သုံးပြီး လက်ရှိ **month** ရဲ့တစ်လအတွင်းစုစုပေါင်း **case_count** နဲ့ စား၊ ၁၀၀ နဲ့မြှောက်ပါတယ်။ ရလာတဲ့တန်ဖိုးကို **ROUND(x, 2) function** နဲ့အုပ်ပေးလိုက်ပါတယ်။ ရာခိုင်နှုန်းတန်ဖိုးကို ဒဿမနှစ်နေရာအထိ ယူမယ်ဆိုတဲ့သဘောပါ။

ဒီနေရာမှာ **common table expression** မသုံးဘဲ တန်းပြီး **case count percentage** ကိုတွက်လိုက်လို့ရပါတယ်။ ဒါပေမယ့် **case count percentage** တွက်တာကရှည်တဲ့အတွက် နောက်ပိုင်းပြန်ဖတ်ရလွယ်အောင် ဒီလို ခွဲရေးလိုက်တာပို့ရှင်းပါတယ်။

5.3 Common pitfalls to avoid

database တွေနဲ့အလုပ်လုပ်တဲ့အခါ မကြာခဏကြုံရတတ်တဲ့ အမှားလေးတွေလည်းရှိပါတယ်။

Wrong data type

❌ data type အမှား

```
CREATE TABLE lab_results (
    test_result VARCHAR(50),    -- ဂဏန်းတွေသိမ်းမယ့် column ကို VARCHAR သုံးထားတယ်
    performed_at VARCHAR(20)    -- အချိန်သိမ်းမယ့် column ကို VARCHAR သုံးထားတယ်
);
```

✅ data type အမှန်

```
CREATE TABLE lab_results (
    test_result DECIMAL(10,2),   -- ဂဏန်းတွေသိမ်းဖို့ DECIMAL သုံးတယ်
    performed_at DATETIME        -- အချိန်သိမ်းဖို့ DATETIME သုံးတယ်
);
```

Data type မှားသုံးမိရင် ဂဏန်းတွေနှိုင်းယှဉ်တဲ့အခါ မှားနိုင်ပါတယ်။ အချိန်ကို **VARCHAR** နဲ့သိမ်းမိရင် အချိန်ပေါ်မူတည်ပြီးတွက်ချက်လို့မရတော့ပါဘူး။ **Data type** မှားရင် **disk space** လည်းပိုကုန်တတ်ပါတယ်။ **Indexing** လုပ်လည်းအသုံးမဝင်တော့ပါဘူး။

Missing foreign key constraint

Foreign key မထည့်ဘဲ **table** တစ်ခုနဲ့တစ်ခု ချိတ်လို့ရနေသေးတာမို့လို့ **foreign key** မထည့်ခဲ့ရင် နောက်ပိုင်းမှာ **data integrity** ပျက်စီးနိုင်ပါတယ်။

❌ ပုံစံအမှား

```
CREATE TABLE prescriptions (
    visit_id INT    -- Foreign key constraint မပါဘူး
);
```

✅ ပုံစံအမှန်

```
CREATE TABLE prescriptions (
    visit_id INT,
    FOREIGN KEY (visit_id) REFERENCES visits(visit_id)
);
```

column တစ်ခုကို **Foreign key constraint** မထည့်ထားရင် တကယ်မရှိတဲ့ **visit_id** တွေနဲ့ ဆေးညွှန်းတွေဝင်လာပါလိမ့်မယ်။ ဆေးခန်းလာပြတဲ့ **visits record** တွေဖျက်တဲ့အခါမှာလည်း အဲဒီ **visit** နဲ့သက်ဆိုင်တဲ့ ဆေးညွှန်း **prescriptions record** တွေက ဒီတိုင်းကြီးကျန်ခဲ့မှာဖြစ်ပါတယ်။ သူနဲ့ချိတ်မယ့် **visit** မရှိတော့ပါဘူး။ **Table join** တွေသုံးထားမယ်ဆိုရင် မှား **join** မိတာ၊ **join** အလုပ်မလုပ်တာမျိုးဖြစ်တတ်ပါတယ်။

Improper handling of NULL values

❌ ပုံစံအမှား

```
SELECT * FROM patients
WHERE phone_number = NULL;   -- ဒီလိုရေးလို့မရပါဘူး
```

✅ ပုံစံအမှန်

```
SELECT * FROM patients
WHERE phone_number IS NULL;   -- IS NULL သုံးရပါမယ်
```

Null စစ်မယ်ဆိုရင် = NULL နဲ့မရပါဘူး။ IS NULL နဲ့မှစစ်ပေးပါတယ်။

Null မဟုတ်တာကိုစစ်မယ်ဆိုရင် != NULL နဲ့မဟုတ်ဘဲ IS NOT NULL သုံးရပါမယ်။

Null ဖြစ်နေရင် user ဘက်ခြမ်းမှာ NULL လို့ပေါ်နေတာ ရုပ်ဆိုးတဲ့အတွက် COALESCE() ၊ IFNULL() function တွေသုံးပြီး Null ဖြစ်နေရင်တော့ 'N/A' လိုပြပေးပါ၊ 0 ပြပေးပါ စသဖြင့် null safe default value ပြပေးသင့်ပါတယ်။

SQL injection risk

Web form တွေမှာ user input လက်ခံတဲ့ form field တွေရှိပါတယ်။ အဲဒီ form field မှာ ဥပမာအားဖြင့် SQL ရဲ့ comment syntax ဖြစ်တဲ့ double dash -- ထည့်လိုက်မယ်ဆိုရင် အဲဒီ form field column ရဲ့နောက်က query တွေအလုပ်မလုပ်တော့ပါဘူး။ username နောက်မှာ -- ထည့်ပြီး password မှာ ထည့်ချင်တာထည့်လိုက်ရင် password မစစ်တော့ဘဲ username မှန်တာနဲ့ login ဝင်သွားတာမျိုးဖြစ်တတ်ပါတယ်။ ဒါကြောင့် web programming ဆက်လုပ်တဲ့အခါမျိုးမှာ WHERE နောက်မှာ user input property နဲ့ တိုက်ရိုက်မတွဲပေးရပါဘူး။ PHP language နဲ့ဥပမာပြပါမယ်။

❌ ပုံစံအမှား

```
-- User input ကို တိုက်ရိုက်သုံးထားတယ်
$query = "SELECT * FROM patients WHERE id = " . $_GET['id'];
```

✅ ပုံစံအမှန်

```
-- Prepared statement သုံးထားတယ်
$stmt = $pdo->prepare("SELECT * FROM patients WHERE id = ?");
$stmt->execute([$_GET['id']]);
```

ဒီစာအုပ်မှာတော့ programming syntax တွေ၊ Cybersecurity နဲ့ပတ်သက်တာတွေ အသေးစိတ်မရေးတော့ပါဘူး။ web programming အပိုင်းရောက်ရင် WHERE နောက်မှာ user input property တိုက်ရိုက်တွဲမရေးရဘူးလို့ပဲ မှတ်ထားပေးပါ။

Over-indexing

❌ ပုံစံအမှား

```
-- Column တိုင်းကို index ထည့်ထားတယ်
CREATE INDEX idx_1 ON patients(full_name);
CREATE INDEX idx_2 ON patients(date_of_birth);
CREATE INDEX idx_3 ON patients(gender);
```

```
CREATE INDEX idx_4 ON patients(address);
CREATE INDEX idx_5 ON patients(phone);
```

✅ ပုံစံအမှန်

```
-- အသုံးများတဲ့ column တွေကိုပဲ index ထည့်ထားတယ်
CREATE INDEX idx_patient_name ON patients(full_name);
CREATE INDEX idx_patient_dob ON patients(date_of_birth);
```

လိုတာထက်ပိုပြီး **index** လုပ်ထားရင် **INSERT/UPDATE** တွေနှေးစေပါတယ်။ **Disk space** နေရာပိုယူပြီး **query optimizer** ကိုလည်းရှုပ်ထွေးစေပါတယ်။

5.4 Resources for further learning

ဒီလောက်ဆို **Basic SQL** အပိုင်းမှာ တော်တော်လေး **cover** ဖြစ်သွားပါပြီ။ ဒါပေမယ့် လက်တွေ့မှာ **SQL** ချည်းသက်သက်သိပ်မသုံးကြတော့ဘဲ **web application backend** တွေနဲ့ချိတ်ဆက်ပြီး **user interface** တွေတည်ဆောက်ပေးနိုင်ဖို့လိုပါတယ်။ **Basic** အပိုင်းသိပြီဆိုရင် **Server side programming language** တွေဖြစ်တဲ့ **Java** ၊ **Python** ၊ **PHP** ၊ **Node.js** ၊ **Ruby** စတာတွေ ဆက်လေ့လာပြီး **backend code** တွေနဲ့ချိတ်ဆက်ရေးသားကြည့်ဖို့တိုက်တွန်းပါတယ်။

SQL ကိုပဲပိုစိတ်ဝင်စားပြီး အသေးစိတ်ဆက်လေ့လာလိုသူများကတော့

freeCodeCamp (freecodecamp.org)

Basic ကနေ advanced သင်ခန်းစာတွေရှိပါတယ်။ သူ့ **website** မှာလည်းလေ့လာလို့ရသလို **YouTube video** တွေလည်းရှိပါတယ်။

Coursera (coursera.org)

Database Management Essentials course တွေ၊ Healthcare Data Analytics သီးသန့် **course** တွေတက်လို့ရပါတယ်။

W3Schools (w3schools.com)

အခြေခံဗဟုသုတတွေ တော်တော်လေးစုံစုံလင်လင်လေ့လာနိုင်ပြီး **online editor** လည်းပါလို့ **website** ပေါ်မှာတင် **SQL script** တွေစမ်းသပ်ရေးသားနိုင်ပါတယ်။

Healthcare Data Science နဲ့သက်ဆိုင်တာတွေက

HealthIT.gov

Healthcare related database တွေ၊ စမ်းသပ်လေ့ကျင့်လို့ရမယ့် **dataset** တွေရနိုင်ပါတယ်။

HIMSS (himss.org)

Healthcare Information and Management Systems Society ရဲ့ **website** ဖြစ်ပါတယ်။ ကျန်းမာရေးနဲ့သက်ဆိုင်တဲ့ **database management** ပိုင်းဆိုင်ရာ ဆောင်းပါးတွေ၊ **webinar** တွေရှိပါတယ်။

Conclusion

SQL ဟာ များများလုပ်လေ ကျွမ်းကျင်လေဖြစ်တဲ့ **skill** တစ်ခုပါ။ စာတွေ့လေ့လာရုံနဲ့မပြီးဘဲ လက်တွေ့ရေးကြည့်၊ တစ်ခုခုအမှားတွေ့ရင် စိတ်ညစ်မသွားဘဲ **error message** ကိုသေချာဖတ်ကြည့်ပြီး ပြင်တဲ့အကျင့်မွေးမြူဖို့လိုပါတယ်။ လက်တွေ့ရေးရင်း တဖြည်းဖြည်းပိုတိုးတက်လာတတ်တဲ့ **skill** ဖြစ်ပါတယ်။ **SQL** နဲ့တင်ရပ်တန့်မသွားဘဲ **R** (သို့) **Python programming language** နဲ့တွဲလေ့လာပြီး **healthcare data science** နယ်ပယ်မှာကျွမ်းကျင်သူများဖြစ်လာစေဖို့ မျှော်လင့်ပါတယ်။