

Begin Modern Programming
with

Python

Pyi Soe

မာတိကာ

၁	Python နှင့် ဒေတာဘေ့ခ်စ် ပရိုဂရမ်မင်း	၁
၁.၁	Database Management Systems	၁
၁.၂	PostgreSQL နှင့် SQL မိတ်ဆက်	၃
၁.၃	Table Relationships	၁၂
၁.၄	SQL ဖန်ရှင်များ	၁၇
၁.၅	Python နှင့် ဒေတာဘေ့ခ်စ် ပရိုဂရမ်မင်း	၁၈
၁.၆	Parameterized SQL	၂၂
၁.၇	fetch -ing and Using Results	၂၄
၁.၈	Python နဲ့ SQL အကြား ဒေတာတိုက်ပိ ပြောင်းလဲခြင်း	၂၆
၁.၉	Dynamic SQL	၂၉
၁.၁၀	Database Transactions	၃၁
၁.၁၁	Concurrency	၃၇
၁.၁၂	SQL Injection and Dynamic SQL	၄၀
၂	Basic Concurrency	၄၅
က	PostgreSQL ဒေတာဘေ့ခ်စ် ဆာဗာဆော့ဖ်ဝဲ ထည့်သွင်းခြင်း	၄၇

အခန်း ၁

Python နှင့် ဒေတာဘေ့စ် ပရိုဂရမ်မင်း

ဒေတာဘေ့စ်တွေဟာ ကနေ့ခေတ် အချက်အလက် အခြေပြစနစ် (Information System) အားလုံးရဲ့ အဓိကကျောရိုးလို့ ဆိုနိုင်ပါတယ်။ ဝတ်အပ်ပလီကေးရှင်း အသုံးပြုသူ ကိုယ်ရေးအချက်အလက် မှတ်တမ်း တင်ခြင်း ကိစ္စကနေ ဘဏ္ဍာရေးဆိုင်ရာ အဖွဲ့အစည်းကြီးတွေ ငွေဝင်ငွေထွက် စာရင်းအထိ အချက်အလက် မျိုးစုံ သိုလှောင်သိမ်းဆည်းရာမှာ အဓိကအကျဆုံးဟာ ဒေတာဘေ့စ်စနစ်တွေပဲ ဖြစ်ပါတယ်။ ကျန်းမာရေး၊ စီးပွားရေး၊ ပညာရေး၊ ဘဏ္ဍာရေး စတဲ့ ကဏ္ဍ အားလုံးမှာ အချက်အလက်တွေ ထိထိရောက်ရောက် သိမ်းဆည်း စီမံနိုင်ဖို့အတွက် ဒေတာဘေ့စ်တွေဟာ မရှိမဖြစ်ပါပဲ။ ဒီလို ကဏ္ဍအသီးသီးမှာ အင်မတန်မှ အရေးကြီးပါတယ်ဆိုတဲ့ အတိတ်ဖြစ်ရပ်တွေကနေ ပြန်လည်သုံးသပ် သင်ခန်းစာယူခြင်းနဲ့ ရှေ့ကိုသိမြင်ဖို့ ကြိုတင်ခန့်မှန်းတွက်ချက်နိုင်ခြင်းဟာလည်း အဲဒီလို လုပ်နိုင်ဖို့အတွက် လိုအပ်တဲ့ အချက်အလက်တွေ သိမ်းဆည်းပေးထားတဲ့ ဒေတာဘေ့စ်တွေသာ မရှိရင် မဖြစ်နိုင်ဘူး ပြောရပါမယ်။

ဒီအခန်းမှာ Python နဲ့ ဒေတာဘေ့စ် ချိတ်ဆက်အသုံးပြုပုံကို လေ့လာကြမှာပါ။ အခြေခံ ဒေတာဘေ့စ် concept တွေကိုတော့ ဒီစာအုပ်မှာ အကျဉ်းချုံးလောက်ပဲ ဖော်ပြပေးနိုင်မယ်။ ပရော်ဖက်ရှင်နယ် အဆင့် ဒေတာဘေ့စ် ပရိုဂရမ်မင်း အတွက်ဆိုရင် တချို့အပိုင်းတွေကို ထဲထဲဝင်ဝင် ဆက်လက် လေ့လာရပါလိမ့်မယ်။ ကိုးကားစာအုပ်တွေ နောက်ဆုံးမှာ ကြည့်နိုင်ပါတယ်။

၁.၁ Database Management Systems

‘ဒေတာဘေ့စ်’ ဆိုတာ အချက်အလက် အမြောက်အများ ရေရှည်သိမ်းဆည်းပေးတဲ့ စနစ်လို့ အကြမ်းဖျဉ်းပြောနိုင်ပါတယ်။ သာမန်အားဖြင့် ရေရှည်သိမ်းထားချင်ရင် ဖိုင်စနစ် သုံးလို့ရပေမဲ့ ဒေတာ များလာတာနဲ့အမျှ ပြန်လည်ရှာဖွေရတာ၊ ထုတ်ယူရတာ၊ အမြဲမှန်ကန်ကိုက်ညီအောင် ထိန်းသိမ်းရတာ စတဲ့ ကိစ္စတွေအတွက် အဆင်မပြေနိုင်ဘူး။ Database Management Systems (DBMS) တွေကို ဒီအခက်အခဲတွေ ဖြေရှင်းပေးဖို့ တီထွင်ခဲ့ကြတာပါ။ အချက်အလက် မှန်ကန်စိတ်ချရခြင်း၊ လုံခြုံမှုရှိခြင်း၊ အလွယ်တကူ access လုပ်နိုင်ခြင်း၊ မျှဝေသုံးစွဲနိုင်ခြင်း စတဲ့အချက်တွေကို DBMS တွေ တည်ဆောက်ရာမှာ ဦးစားပေး ထည့်သွင်း စဉ်းစားထားတယ်။

သမိုင်းအကျဉ်း

ဒေတာဘေ့စ်တွေရဲ့ မူလအစ concept ဟာ IBM ကုမ္ပဏီက IMS (Information Management System) လို့ စနစ်တွေ တည်ဆောက်ခဲ့တဲ့ ၁၉၆၀ ခုနှစ်တွေလောက်ကို ပြန်သွားနိုင်တယ်။ အဲဒီစနစ်တွေက hierarchical ဖြစ်တယ်။ ဆိုလိုတာက ဒေတာသိမ်းတဲ့ စထရက်ချာက သစ်ပင်လိုပဲ၊ အပင်ရဲ့ အမြစ်၊

အရွက်၊ အကိုင်းအခက်တွေ ဆက်စပ်နေသလိုပုံစံနဲ့ အချက်အလက်တွေကို သိမ်းတယ်။ Parent-child relationship နဲ့ သိမ်းတာလို့လည်း ဆိုနိုင်တယ်။ ၁၉၇၀ ခုနှစ်တွေမှာတော့ Edgar F. Codd က ကနေ့ခေတ် Relational Database Management System (RDBMS) ရဲ့ အခြေခံအုတ်မြစ် ဖြစ်လာတဲ့ Relational Data Model ကို စတင်မိတ်ဆက်ခဲ့တယ်။ Relational model မှာက ဒေတာသိုလှောင်သိမ်းဆည်းဖို့ table တွေကို အသုံးပြုပြီး SQL (Structured Query Language) လို့ ခေါ်တဲ့ programming language ကို ထောက်ပံ့ပေးပါတယ်။

SQL Language

SQL ဟာ table ဒေတာ အမြောက်အများကနေ မိမိစူးစမ်းလိုတဲ့ အချက်အလက်ကို အလွယ်တကူ ထုတ်ယူ (သို့) မေးမြန်းလို့ရအောင် ကူညီထောက်ပံ့ပေးဖို့ အဓိကရည်ရွယ်တယ်။ “ဒီနှစ် ဇွန်လစာမေးပွဲ မှာ စန္ဒာ ဘာသာရပ်အသီးသီး ရမှတ်ဘယ်လောက်လဲ” လို့ ခပ်ရိုးရိုး မေးခွန်းကနေ “ဘယ်ကျောင်းသူ ကျောင်းသား တွေ စာမေးပွဲအားလုံးမှာ သုံးနှစ်ဆက်တိုက် သင်္ချာရမှတ် ၉၅ မှတ်အထက် ရကြလဲ” ဆို တဲ့ အတော်လေး ရှုပ်ရှုပ်ထွေးထွေး မေးခွန်းမျိုးတွေထိ SQL နဲ့ အလွယ်တကူ မြန်မြန်ဆန်ဆန် ရှာဖွေ ဖော်ထုတ်နိုင်ပါတယ်။

အချက်အလက် ထုတ်ယူတာအပြင် ဒေတာဘေ့စ် အသစ်ဆောက်တာ၊ table ဆောက်တာ၊ ပြန်ဖျက် တာ စတဲ့ကိစ္စတွေကိုလည်း SQL နဲ့ပဲ လုပ်ရပါတယ်။ Table မှာ record အသစ်ထည့်တာ၊ ရှိပြီးသား record ကို update လုပ်တာ၊ ဖျက်ပစ်တာ စတာတွေအတွက်လည်း SQL ကိုပဲ သုံးရတာပါ။

SQL ဟာ RDBMS အားလုံးမှာ အသုံးပြုနိုင်တဲ့ standard language တစ်ခုလည်းဖြစ်တယ်။ ဆိုလိုတာက ဘယ် RDBMS ကိုပဲ သုံးသုံး၊ SQL တစ်မျိုးတည်းကိုပဲ သုံးရမှာပါ။ RDBMS တစ်ခု မှာ သူ့ကိုယ်ပိုင် ချဲ့ထွင်ထားတဲ့ အပိုင်းတွေ အနည်းအကျဉ်း ရှိကြပေမဲ့ SQL standard ကို ရာနှုန်းပြည့် မဟုတ်တောင် အဲ့လောက်နီးနီး လိုက်နာထားကြတဲ့အတွက် ပြောပလောက်အောင် မကွာကြဘူး။ SQL သာနားလည်ပါစေ၊ ဘယ်ဒေတာဘေ့စ်နဲ့မဆို အလုပ်ဖြစ်တယ်လို့ ပြောရင်လည်း မမှားဘူး။

Client-Server Applications

တစ်ချိန်မှာ တစ်ယောက်ပဲ သုံးလို့ရတဲ့ ဆော့ဖ်ဝဲတွေနဲ့ မတူတာက DBMS တွေဟာ တစ်ယောက်မက တပြိုင်နက် သုံးလို့ရတဲ့ ဆာဗာ (server) ဆော့ဖ်ဝဲတွေ ဖြစ်ပါတယ်။ တပြိုင်နက် ဝင်ရောက်လာတဲ့ အသုံးပြု သူတွေရဲ့ တောင်းဆိုချက်တွေကို ဖြည့်ဆည်းဖို့အတွက် ကိုင်တွယ်ဆောင်ရွက် ပေးနိုင်စွမ်း ရှိတယ်။

တစ်ခုထက်ပိုတဲ့ client တွေက ဆာဗာတစ်ခုနဲ့ ချိတ်ဆက်အသုံးပြုတဲ့ ဆော့ဖ်ဝဲစနစ်မျိုးကို Client-Server Application လို့ ခေါ်တယ်။ Client ဆိုတာ အသုံးပြုသူ user (သို့) ၎င်းအသုံးပြုတဲ့ ပရိုဂရမ် ကို ဆိုလိုတာ။ ဒေတာဘေ့စ် application အများစုဟာ Client-Server Application တွေပါ။ Web Application တွေဟာ Client-Server Application ဖြစ်ပြီး ဒေတာသိမ်းဖို့အတွက် နောက်ကွယ်က RDBMS တစ်ခုနဲ့ ချိတ်ဆက်ထားရလေ့ရှိတယ်။

Database Transactions, ACID and Concurrency

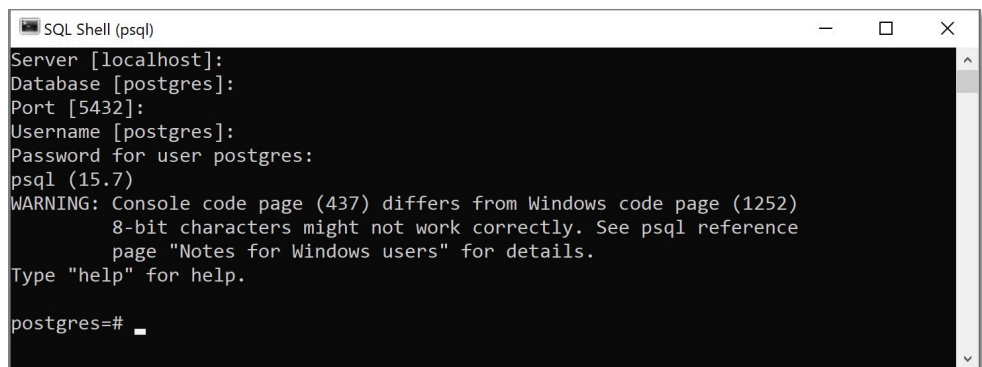
RDBMS တွေမှာ Transaction, ACID Properties (Atomicity, Consistency, Isolation, Durability) နဲ့ Concurrency စတဲ့ သဘောတရားတွေလည်း အရေးပါတယ်။ ဒါတွေနဲ့ ပါတ်သက် ပြီးတော့လည်း အခြေခံအဆင့် စာအုပ်တစ်အုပ်အနေနဲ့ လက်ရှိအခန်း နောက်ပိုင်းမှာ အတန်အသင့် ခြုံငုံမိ အောင် ဖော်ပြပေးထားပါတယ်။

၁.၂ PostgreSQL နှင့် SQL မိတ်ဆက်

ဆော့ဖ်ဝဲရေးကွက်မှာ Relational Database Management System (RDBMS) တွေ ရွေးချယ်စရာ အတော်များတယ်။ Oracle, Microsoft SQL, MySQL, PostgreSQL တို့ဟာ လူသိများ။ အသုံးများတဲ့ RDBMS တွေပါ။ PostgreSQL နဲ့ MySQL က ပိုက်ဆံ ပေးစရာမလိုဘဲ အသုံးပြုနိုင်ပြီး Oracle နဲ့ Microsoft SQL တို့ကတော့ လိုင်စင်ဝယ်သုံးရတာတွေ ဖြစ်ကြပေမဲ့ အလကား သုံးလို့ရတဲ့ developer version တွေလည်း ပေးထားတယ်။ ဒါကြောင့် ကိုယ်လေ့လာချင်တဲ့ ဒေတာဘေ့စ်ကို လေ့လာနိုင်ဖို့ လိုင်စင်အနေနဲ့ စိတ်ပူစရာမရှိဘူး။

PostgreSQL နဲ့ MySQL က အလကားရတာမို့လို့ ဝယ်သုံးရတာတွေလောက် အရည်အသွေးကောင်းမှာ မဟုတ်ဘူး ထင်ကောင်းထင်နိုင်ပါတယ်။ အမှန်တကယ်က အဲ့လိုပြောလို့ မရပါဘူး။ နှစ်ခုလုံးက လိုင်စင်ဝယ်သုံးရတဲ့ ဟာတွေလိုပဲ အရည်သွေးရော ပါဝင်တဲ့ ဖီချာတွေ စုံလင်မှုအရပါ အားကောင်းကြတယ်။ PostgreSQL အသုံးပြုတဲ့ မှတ်သားဖွယ် ကုမ္ပဏီ/အဖွဲ့အစည်းတွေကို Wikipedia မှာ ဖော်ပြထားတာ တွေ့နိုင်တယ် (https://en.wikipedia.org/wiki/PostgreSQL#Notable_users)။ MySQL ကလည်း သူ့ရဲ့ နံမည်ကြီး customer တွေကို သူ့ဝဘ်ဆိုဒ်မှာ ဖော်ပြထားတာ တွေ့ရတယ် (<https://www.mysql.com/customers>)။

ဒီစာအုပ်မှာ PostgreSQL RDBMS အသုံးပြုပါမယ်။ PostgreSQL ဘယ်လို အင်စတောလ် လုပ်ရမလဲ စာမျက်နှာ ၄၇ နောက်ဆက်တွဲ (က) မှာ ကြည့်ပါ။ အင်စတောလ်လုပ်ပြီးရင် psql ဖွင့်ပြီး root user (postgres) အနေနဲ့ PostgreSQL ဒေတာဘေ့စ်ကို ချိတ်ဆက်ထားပါ။



```
SQL Shell (psql)
Server [localhost]:
Database [postgres]:
Port [5432]:
Username [postgres]:
Password for user postgres:
psql (15.7)
WARNING: Console code page (437) differs from Windows code page (1252)
         8-bit characters might not work correctly. See psql reference
         page "Notes for Windows users" for details.
Type "help" for help.

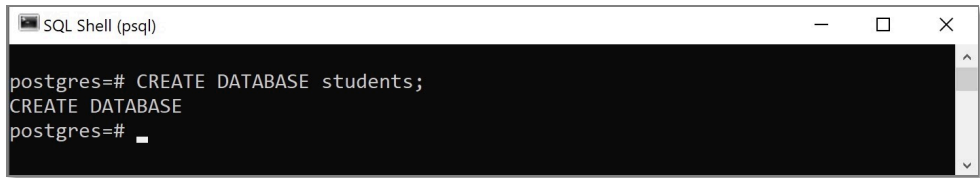
postgres=#
```

ပုံ ၁.၁ psql မှ PostgreSQL ကို စတင်ချိတ်ဆက်ပုံ

psql ဖွင့်ပြီး စတင်ချင်းမှာ ဒေတာဘေ့စ်နဲ့ ချိတ်ဆက်ဖို့ လိုအပ်တဲ့ အချက်အလက်တွေ ထည့်ပေးရပါမယ်။ ပုံ (၁.၁) မှာ ကြည့်ပါ။

- Server [localhost]: (ချိတ်ဆက်မဲ့ ဆာဗာရဲ့ Host Name/IP Address)
- Database [postgres]: (ချိတ်ဆက်မဲ့ ဒေတာဘေ့စ်နံမည်)
- Port [5432]: (PostgreSQL port နံပါတ်)
- Username [postgres]: (ဒေတာဘေ့စ် ချိတ်ဆက်အသုံးပြုမဲ့ user)
- Password for user postgres: (ဒေတာဘေ့စ်ကို ချိတ်ဆက်အသုံးပြုမဲ့ user ရဲ့ password)

လေးထောင့်ကွင်းထဲက default တန်ဖိုးတွေပါ။ တကူးတက ဘာမှမထည့်နေပဲ Enter နှိပ်ရင် အဲဒီတန်ဖိုးတွေ ထည့်တာနဲ့ တူတူပါပဲ။ စာမျက်နှာ ၄၇ နောက်ဆက်တွဲ (က) မှာ ဖော်ပြထားတဲ့အတိုင်း အင်စတောလ်လုပ်ထားတာဆိုရင် password တစ်ခုပဲ ထည့်ပေးဖို့လိုတယ်။ ကျန်တာက default အတိုင်း



```
SQL Shell (psql)
postgres=# CREATE DATABASE students;
CREATE DATABASE
postgres=#
```

ပုံ ၁.၂ psql ကနေ 'CREATE DATABASE' SQL run တာ

ထားပြီး Enter နှိပ်သွားလို့ရတယ်။ Password က အင်စတောလ်လုပ်တုန်းက ပေးခဲ့တဲ့ password ကို ထည့်ပေးရမှာပါ။

psql ဆိုတာဘာလဲ။ psql ဟာ PostgreSQL ကို ချိတ်ဆက်အသုံးပြုဖို့ သုံးတဲ့ command-line ပရိုဂရမ်တစ်ခုပါ။ Client ပရိုဂရမ် အနေနဲ့ အသုံးပြုရတာပါ။ ဒေတာဘေ့စ် ချိတ်ဆက်ခြင်း၊ ဒေတာဘေ့စ် ဆီကို SQL ကွန်မန်းတွေ ပေးပို့လုပ်ဆောင်စေခြင်း၊ ဒေတာဘေ့စ် ပြန်ပို့ပေးတဲ့ ရလဒ်တွေပြပေးခြင်း၊ ဒေတာဘေ့စ် စီမံခန့်ခွဲခြင်း (database administration) စတဲ့ ကိစ္စတွေအတွက် အသုံးပြုနိုင်တယ်။ ရိုးရိုးရှင်းရှင်းပေမဲ့ အစွမ်းထက်တဲ့ tool တစ်ခုဖြစ်ပါတယ်။

ဒေတာဘေ့စ် အသစ်ဆောက်ခြင်း

ဒေတာဘေ့စ် အသစ်တစ်ခု ဆောက်မယ်ဆိုရင် CREATE DATABASE SQL ကွန်မန်း သုံးပါတယ်။

```
CREATE DATABASE database_name;
```

SQL language ဟာ စာလုံး အကြီးအသေး မခွဲဘူး။ ဒီစာအုပ်မှာ SQL keyword တွေဆိုရင် အက္ခရာ အကြီးနဲ့ ရေးပါမယ်။ Database, table, column, function စတာတွေရဲ့ နံမည် (identifiers) တွေက အက္ခရာအသေးနဲ့ ဖြစ်မယ်။ student ဒေတာဘေ့စ် အတွက် ဒီ SQL ကို

```
CREATE DATABASE students;
```

psql ကနေ run ပေးပါ [ပုံ (၁.၂)]။ SQL စတိတ်မန့် တစ်ကြောင်း အဆုံးမှာ ဆီမီးကော်လံ (;) ထည့်ပေးရပါမယ်။

psql ကနေ \l (သို့) \list ကွန်မန်းနဲ့ PostgreSQL မှာ ရှိတဲ့ ဒေတာဘေ့စ်တွေကို ထုတ်ကြည့်နိုင်ပါတယ်။ ဒီလို \ (backslash) နဲ့ စတဲ့ ကွန်မန်းတွေဟာ psql သီးသန့်ဖြစ်တယ်။ SQL မဟုတ်တဲ့ အတွက် run ရင် ; မထည့်ရဘူး။ \l run လိုက်ရင် စာရင်းထဲမှာ students ဒေတာဘေ့စ် တွေ့ရမှာပါ [ပုံ (၁.၃)]။

psql မှာ ဒေတာဘေ့စ် ပြောင်းချိတ်မယ်ဆိုရင် \c (သို့) \connect နဲ့ ပြောင်းရပါမယ်။ students ဒေတာဘေ့စ်ကို အခုလို connect လုပ်ပါ

```
\c students
```

psql က ဒီလို ပြပါလိမ့်မယ်

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

```
Select SQL Shell (psql)
postgres=# \l

          List of databases
  Name | Owner | Encoding | Collate | Ctype | ICU Locale | Locale Provider | Access privileges
-----+-----+-----+-----+-----+-----+-----+-----
postgres | postgres | UTF8 | English_United States.1252 | English_United States.1252 | English_United States.1252 | English_United States.1252 | 
students | postgres | UTF8 | English_United States.1252 | English_United States.1252 | English_United States.1252 | English_United States.1252 | 
template0 | postgres | UTF8 | English_United States.1252 | English_United States.1252 | English_United States.1252 | English_United States.1252 | 
template1 | postgres | UTF8 | English_United States.1252 | English_United States.1252 | English_United States.1252 | English_United States.1252 | 
(4 rows)

postgres=#
```

ပုံ ၁.၃ psql ကနေ ဒေတာဘေ့စ်တွေ list ထုတ်ကြည့်တာ

SQL ကွန်မန်းကို psql က လုပ်ဆောင်ပေးတာ မဟုတ်ပါဘူး။ psql က SQL ကွန်မန်းကို ချိတ်ဆက်ထားတဲ့ ဒေတာဘေ့စ်ဆီ ပို့ပဲ ပို့ပေးတာ။ လက်ခံရရှိတဲ့ ဒေတာဘေ့စ်က အဲဒီ SQL ကို လုပ်ဆောင်ပေးတာပါ။ ဒီအချက်ကို ကွဲကွဲပြားပြား နားလည်ဖို့ လိုပါတယ်။

Table ဆောက်ခြင်း

CREATE TABLE က table ဆောက်တဲ့ SQL ကွန်မန်းပါ။ students ဒေတာဘေ့စ်ထဲမှာ student table ဆောက်ပါအတိုင်း ဆောက်ပါမယ်

```
-- create a table named student
CREATE TABLE student (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    age INT,
    grade VARCHAR(2)
);
```

psql မှာ SQL run ရင် လက်ရှိချိတ်ထားတဲ့ ဒေတာဘေ့စ်ကို အဲဒီ SQL ပေးပို့ လုပ်ဆောင်ခိုင်းပါတယ်။ အခုချိတ်ထားတာ students ဒေတာဘေ့စ်ဆိုတော့ table ကို အဲဒီ ဒေတာဘေ့စ်ထဲ ဆောက်ပေးသွားမှာပါ။ ဒီ table မှာ id, name, age နဲ့ grade column လေးခုရှိမယ်။ -- ကို တစ်ကြောင်းတည်း ကွန်းမန့် အတွက် သုံးတယ်။ တစ်ကြောင်းထက်ပိုရင် /* နဲ့ ဖွင့် */ နဲ့ ပိတ် ရေးလေ့ရှိတယ်။ ဥပမာ

```
/* This is a
   multiline comment */
```

Column တစ်ခုစီမှာ data type ရှိပါမယ်။ VARCHAR(100) က အများဆုံး ကာရက်တာ အလုံးတစ်ရာ သိမ်းဆည်းနိုင်တယ်။ သိမ်းတဲ့ ကာရက်တာ အရေအတွက်ပေါ် မူတည်ပြီး နေရာယူတာ အနည်းအများ ကွာတယ်။ ငါ့သိမ်းရင် ငါးခုစာ၊ ဆယ်ခုသိမ်းရင် ဆယ်ခုစာပဲ နေရာကုန်မှာပါ။ အမြဲ အလုံးတစ်ရာစာ နေရာကုန်တာ မဟုတ်ဘူး။ VARCHAR(2) ဆိုရင် အများဆုံး ကာရက်တာ နှစ်လုံး သိမ်းလို့ရမယ်။ SQL VARCHAR က Python str နဲ့ အလားတူတယ်။ INT ကတော့ integer ပါ။

id column က ထူးခြားပြီး နည်းနည်းပိုရှင်းပြဖို့ လိုတယ်။ SERIAL က data type အနေနဲ့

INT နဲ့ တူတူပဲ။ သူ့ရဲ့ ထူးခြားချက်က ဂဏန်းတွေကို အစဉ်အတိုင်း တစ်ခုပြီးတစ်ခု ထုတ်ပေးနိုင်တာပါ။ 1, 2, 3, ... စသည်ဖြင့် နောက်ဆုံးတန်ဖိုးကို အလိုအလျောက် တစ်တိုးတိုးပြီး ထုတ်ပေးသွားမှာ ဖြစ်တယ်။ id column က Primary Key လည်းဖြစ်တယ်။ Column တစ်ခုကို Primary Key အဖြစ် ထားချင်ရင် PRIMARY KEY လို့ သတ်မှတ်ရပါမယ်။ Primary Key ဆိုရင် column တန်ဖိုး ထပ် (duplicate) လို့မရဘူး၊ unique ဖြစ်ရပါမယ်။ အခု သိပ်နားမလည်သေးရင်လည်း table မှာ ကျောင်းသား record တွေထည့်တာ ဆက်ကြည့်ရင် ကောင်းကောင်း နားလည်သွားမှာပါ။

INSERT

Relational Data Model အခြေခံတဲ့ RDBMS တွေဟာ ဒေတာတွေကို table ပုံစံနဲ့ သိမ်းဆည်းတယ်။ ကျောင်းသူ/သား တစ်ယောက်ချင်းစီအတွက် အချက်အလက်ကို student table မှာ row တစ်ခုစီနဲ့ ထည့်သွင်း သိမ်းဆည်းပါမယ်။ Row ကို record လို့လည်း သုံးနှုန်းလေ့ရှိတယ်။ Record အသစ် ထည့်သွင်းမယ်ဆိုရင် SQL INSERT ကို သုံးရပါတယ်။

```
INSERT INTO student (name, age, grade) VALUES ('Amy', 20, 'A');
INSERT INTO student (name, age, grade) VALUES ('Kathy', 22, 'B');
INSERT INTO student (name, age, grade) VALUES ('Waiyan', 21, 'C');
```

အေမီ၊ ကေသီ နဲ့ ဝေယံ ကျောင်းသား သုံးယောက်အတွက် record သုံးခု ထည့်သွင်းတာပါ။ Column နံမည်တွေ ဝိုက်ကွင်းထဲမှာ ထည့်ပြီး အဲဒီ column တွေအတွက် တန်ဖိုးအသီးသီးကို အစဉ်အတိုင်း ထည့်ပေးရပါတယ်။ age နဲ့ grade ရှေ့နောက် ဖလှယ်လိုက်မယ်ဆိုရင် အခုလို

```
INSERT INTO student (name, grade, age) VALUES ('Amy', 'A', 20);
```

insert လုပ်ရမှာပါ။

student table မှာ column က လေးခု ရှိတာပါ။ အခု INSERT တွေမှာကျတော့ သုံးခုပဲတွေ့ရပြီး id မပါဘူး။ ဘာကြောင့်လဲ။ INSERT လုပ်တဲ့အခါ SERIAL column အတွက် တန်ဖိုးကို ဒေတာဘေ့စ်က အလိုအလျောက် ထည့်ပေးသွားတာ။ ကိုယ်တိုင်ထည့်ဖို့ မလိုဘူး။ ဒါကြောင့် id column ကို *auto-incrementing* primary key column လို့ ခေါ်တယ်။ Auto-increment ဖြစ်ဖို့ အခြားနည်းလမ်းတွေလည်း ရှိပါတယ်။ SERIAL ကတော့ ဒီကိစ္စအတွက် လွယ်အောင် လုပ်ပေးထားတာပါ။ စောစောက INSERT သုံးကြောင်းကို psql မှာ run ပါ။ အေမီ၊ ကေသီ နဲ့ ဝေယံတို့အတွက် record အသီးသီးကို id နံပါတ် 1, 2, 3 အစဉ်နဲ့ student table ထဲ ထည့်သွင်းသွားမှာဖြစ်တယ်။ နောက်ထပ် record တစ်ခု ထပ်ထည့်ရင် id နံပါတ် 4 ဖြစ်မှာပါ။

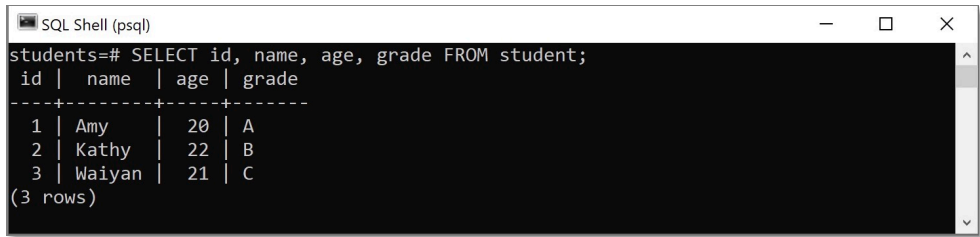
SELECT

Table ဒေတာတွေ ထုတ်ယူကြည့်ဖို့ အသုံးပြုတဲ့ SQL ဖြစ်ပါတယ်။ Student table ထဲက record အားလုံးကို ကြည့်မယ်ဆို အခုလို

```
SELECT id, name, age, grade FROM student;
```

Table မှာ ရှိသမျှ column အကုန်လုံး ပါချင်ရင် SELECT * သုံးလို့လည်းရတယ်။ SELECT * ကို 'Select All' လို့ ဖတ်တယ်။

```
SELECT * FROM student;
```



```
SQL Shell (psql)
students=# SELECT id, name, age, grade FROM student;
 id | name  | age | grade
-----+-----+-----+-----
  1 | Amy   |  20 | A
  2 | Kathy |  22 | B
  3 | Waiyan | 21 | C
(3 rows)
```

ပုံ ၁.၄ psql ကနေ select လုပ်တာ

Column အကုန်မထုတ်ဘဲ ကိုယ်လိုချင်တာပဲ ရွေးပြီး select လုပ်ချင်လည်း ရတယ်။ အောက်ပါတို့ကို psql မှာ စမ်းကြည့်ပါ။

```
SELECT name, grade FROM student;
SELECT name, age, id FROM student;
```

WHERE

Grade A ရတဲ့ ကျောင်းသားတွေကိုပဲ ရွေးထုတ်ကြည့်မယ် ဆိုပါစို့။ ဒီအတွက် SQL မှာ WHERE ရှိပါတယ်။ ဥပမာ

```
SELECT * FROM student WHERE grade = 'A';
```

WHERE နောက်မှာ ဘူလီယန် အိပ်စ်ပရက်ရှင်တစ်ခု လိုက်ရပါမယ်။ WHERE ကွန်ဒီရှင် (Condition) လို့ခေါ်တယ်။ ရေးပုံရေးနည်း နည်းနည်းကွာပေမဲ့ SQL WHERE ကွန်ဒီရှင် က Python ဘူလီယန် အိပ်စ်ပရက်ရှင်နဲ့ သဘောတရားအားဖြင့် တူပါတယ်။ grade = 'A' က grade column တန်ဖိုး 'A' နဲ့ ညီလား စစ်တာ။ ညီတဲ့ record တွေကိုပဲ WHERE က စစ်ထုတ်ပေးမှာပါ။ ဒါ့ကြောင့် အပေါ်က select က record တစ်ကြောင်းပဲ ထွက်မှာပါ။ အဲဒီတစ်ယောက်ပဲ A ရပါတယ်။ WHERE နဲ့ပါတ်သက်ပြီး လက်တွေ့စမ်းကြည့်ရအောင် အောက်ပါအတိုင်း ကျောင်းသား record လေးခု ထပ်ထည့်ပါမယ်။

```
INSERT INTO student (name, age, grade) VALUES ('Sandy', 19, 'A');
INSERT INTO student (name, age, grade) VALUES ('Thida', 21, 'B');
INSERT INTO student (name, age, grade) VALUES ('Peter', 21, 'B');
INSERT INTO student (name, age, grade) VALUES ('Haymar', 18, NULL);
```

Grade A သို့ B ရတဲ့ ကျောင်းသား record တွေ select လုပ်ဖို့ OR သုံးထားတာပါ။ psql မှာ စမ်းကြည့်ပါ။ Amy, Kathy, Sandy, Thida, Peter တို့ A သို့ B ရကြတယ်။

```
SELECT * FROM student WHERE grade = 'A' OR grade = 'B';
```

Grade A သို့ B မရတဲ့ ကျောင်းသားတွေ ထုတ်ချင်ရင် NOT နဲ့ အခုလို ရတယ်

```
SELECT name FROM student WHERE NOT(grade = 'A' OR grade = 'B');
```

ဝေယံ တစ်ယောက်ပဲ ရလဒ်မှာတွေ့ရမှာပါ။ ဟေမာ ဘာကြောင့် မပါရတာလဲ။ စဉ်းစားကြည့်ရင် သူမ A လည်းမရ၊ B လည်းမရဘူး။ ဒါ့ကြောင့် ပါသင့်တယ် ယူဆကောင်း ယူဆနိုင်တယ်။ NULL ဟာ မရှိခြင်း၊ မသိခြင်း ကိုဖော်ပြဖို့ SQL မှာ အသုံးပြုတဲ့ special value တစ်ခု ဖြစ်ပါတယ်။ ဟေမာ့ grade က

NULL ဖြစ်နေတယ်။ ဆိုလိုတာက သူ့ grade ကို မသိဘူး။

NULL နဲ့ အခြားတန်ဖိုးတစ်ခုခု ညီ/မညီ စစ်တဲ့အခါ ရလဒ်က NULL ပဲ ဖြစ်ပါတယ်။ အဓိပ္ပါယ်က ညီ/မညီ 'မသိဘူး' ဆိုတဲ့ အဓိပ္ပါယ်။ ဒါကြောင့် ဘူလီယန် အိပ်စ်ပရက်ရှင် 'A' = NULL ရဲ့ အဖြေ NULL ဖြစ်သလို 'A' <> NULL ရဲ့ အဖြေလည်း NULL ပဲ ဖြစ်တယ်။ <> က မညီဘူးလား စစ်တာ၊ = နဲ့ ပြောင်းပြန်။

Select လုပ်တဲ့အခါ WHERE ကွန်ဒီရှင် true ဖြစ်တဲ့ record တွေကို ရွေးထုတ်ပေးတယ်။ WHERE ကွန်ဒီရှင် ရလဒ်တန်ဖိုး NULL ဖြစ်ရင် အဲဒီ record ကို ထုတ်ပေးမှာ မဟုတ်ဘူး။ စောစောက select ရလဒ်မှာ ဟေမာ ဘာကြောင့် မပါလဲ အောက်ပါအတိုင်း စဉ်းစားကြည့်နိုင်ပါတယ်

WHERE NOT(NULL = 'A' OR NULL = 'B')
 ⇒ WHERE NOT(NULL OR NULL)
 ⇒ WHERE NOT(NULL)
 ⇒ WHERE NULL

Grade C မဟုတ်တဲ့ ကျောင်းသားတွေကို အောက်ပါအတိုင်း နည်းလမ်းနှစ်မျိုးနဲ့ select လုပ်ကြည့်ပါ။ အခုတစ်ခါလည်း ဟေမာ ရလဒ်မှာ မပါတာကို သတိပြုပါ။

```
SELECT * FROM student WHERE grade <> 'C';
```

```
SELECT * FROM student WHERE NOT(grade = 'C');
```

ဒီတစ်ခု ထပ်စမ်းကြည့်ပါ။ ရှင်းပြဖို့မလိုဘဲ အဓိပ္ပါယ် နားလည်မယ် ထင်ပါတယ်။

```
SELECT * FROM student WHERE grade = 'B' AND id <= 5;
```

NULL ဟုတ်/မဟုတ် စစ်ချင်ရင် SQL မှာ IS NULL (သို့) IS NOT NULL သုံးရပါတယ်။ = နဲ့ <> ကို သုံးလို့မရဘူး။ မှားယွင်း အသုံးပြုမိတတ်လို့ ဒီအချက်ကို အထူးဂရုပြုရပါမယ်။ Grade NULL ဖြစ်တဲ့ record တွေနဲ့ NULL မဟုတ်တဲ့ record တွေကို အခုလို ရွေးထုတ်နိုင်ပါတယ်။

```
SELECT * FROM student WHERE grade IS NULL;  
SELECT * FROM student WHERE grade IS NOT NULL;
```

Grade NULL ဖြစ်တာ ဟေမာတစ်ယောက်ပဲ ရှိတာမို့လို့ ပထမ select က record တစ်ကြောင်းပဲ ထွက်မှာပါ။ အောက်ပါအတိုင်း တစ်ဆင့်ချင်း စဉ်းစားကြည့်ပါ

WHERE grade IS NULL
 ⇒ WHERE NULL IS NULL
 ⇒ WHERE TRUE

ဒုတိယ select မှာ ကျတော့ ဘာကြောင့် ဟေမာ မပါလဲ။ ကျန်တဲ့သူတွေကရော ဘာကြောင့်ပါလဲ။ အောက်ပါအတိုင်း တစ်ဆင့်ချင်း စဉ်းစားကြည့်ပါ။ ဟေမာ record အတွက်

WHERE grade IS NOT NULL
 ⇒ WHERE NULL IS NOT NULL
 ⇒ WHERE FALSE

A ရထားတဲ့ ကျောင်းသား record ဆိုရင် ဒီလို

WHERE grade IS NOT NULL
 ⇒ WHERE `A` IS NOT NULL
 ⇒ WHERE TRUE

အခြား NULL မဟုတ်တဲ့ grade အားလုံးအတွက် အလားတူဖြစ်မယ်။ ဒါ့ကြောင့် ဒုတိယ select ရလဒ်မှာ ဟေမာကလွဲလို့ ကျန်တဲ့သူအားလုံး ပါလာတာဖြစ်တယ်။

ORDER BY

ORDER BY က record တွေကို column တန်ဖိုးပေါ် မူတည်ပြီး ‘အစဉ်အတိုင်းစီခြင်း’ (sorting) အတွက်ပါ။

SELECT * FROM student ORDER BY grade;

Grade အလိုက် order by လုပ်ထားတာပါ။ ရလဒ်ကို ပုံ (၁.၅) မှာ ကြည့်ပါ။ ကြီးစဉ်ငယ်လိုက် စီချင်ရင် DESC (descending) နဲ့ ရတယ်။

```

SQL Shell (psql)
students=# SELECT * FROM student ORDER BY grade;
 id | name  | age | grade
----+-----+----+-----
  1 | Amy   |  20 | A
  4 | Sandy |  19 | A
  6 | Peter |  21 | A
  2 | Kathy |  22 | B
  5 | Thida |  21 | B
  3 | Waiyan | 21 | C
  7 | Haymar | 18 | C
(7 rows)

```

ပုံ ၁.၅ order by နဲ့ grade အလိုက် စီထားတာ

SELECT * FROM student ORDER BY grade DESC;

သူ့နဲ့ default ကတော့ ASC (ascending) ပါ။

Column တစ်ခုမကနဲ့ စီချင်လည်း ရတယ်။ ORDER BY နောက်မှာ sort လုပ်ရမဲ့ column တွေ ထည့်ပေးရုံပဲ။ Age နဲ့ grade တွဲရက် sort လုပ်မယ်ဆိုရင်

SELECT * FROM student ORDER BY age, grade;

psql ရလဒ်မှာ အောက်ပါအတိုင်း တွေ့ရမှာပါ။ Thida, Peter, Waiyan တို့ကို သေချာ ဂရုပြုကြည့်ပါ။ Age တူရင် grade B ရတဲ့သူက အရင်လာတာကို တွေ့ရပါမယ်။ Grade C က နောက်မှာပါ။

Output:

```

 id | name  | age | grade
----+-----+----+-----
  7 | Haymar |  18 | 
  4 | Sandy |  19 | A

```

```

1 | Amy      | 20 | A
5 | Thida    | 21 | B
6 | Peter     | 21 | B
3 | Waiyan   | 21 | C
2 | Kathy     | 22 | B

```

(7 rows)

Name ထပ်ထည့်ကြည့်ပါ။ Peter က Thida ရဲ့ ရှေ့ရောက်သွားတာကလွဲလို့ ခုနက စီထားတာနဲ့ အားလုံး တူပါမယ်။

```
SELECT * FROM student ORDER BY age, grade, name;
```

Age နဲ့ grade ကို ရှေ့နောက် ပြောင်းကြည့်ပါ။

```
SELECT * FROM student ORDER BY grade, age;
```

Grade A, B, C အစဉ်အတိုင်း ဖြစ်မယ်။ Grade တူရင်တော့ age ငယ်တဲ့ record က ရှေ့ရောက် ပါတယ်။ အခုလို စီသွားမှာပါ။

Output:

```

id | name  | age | grade
----+-----+-----+-----
4 | Sandy | 19 | A
1 | Amy   | 20 | A
5 | Thida | 21 | B
6 | Peter | 21 | B
2 | Kathy | 22 | B
3 | Waiyan | 21 | C
7 | Haymar | 18 |

```

(7 rows)

Grade ကို DESC နဲ့ age ကို ASC ထားကြည့်ပါ။ ခုနကဟာနဲ့ ဘာကွာခြားလဲ သေချာရှုပြု လေ့လာ ကြည့်ပါ။

```
SELECT * FROM student ORDER BY grade DESC, age ASC;
```

Output:

```

id | name  | age | grade
----+-----+-----+-----
7 | Haymar | 18 |
3 | Waiyan | 21 | C
6 | Peter  | 21 | B
5 | Thida  | 21 | B
2 | Kathy  | 22 | B
4 | Sandy  | 19 | A
1 | Amy    | 20 | A

```

(7 rows)

ORDER BY နဲ့ စီတဲ့အခါ ORDER BY နောက်မှာ list လုပ်ထားတဲ့ column အစီအစဉ်နဲ့ ASC,

DESC တို့ကို လိုသလို အသုံးပြုပြီး လိုချင်တဲ့အတိုင်း ရအောင် sort လုပ်လို့ရပါတယ်။ ပရိုဂရမ်းမင်းမှာ ရော ဒေတာဘေ့စ်ပိုင်းမှာပါ sorting စီခြင်းဟာ အရေးကြီးတာကြောင့် ကျွမ်းကျင်အောင် လုပ်ထားသင့်ပါတယ်။

UPDATE

Table record တွေ update လုပ်ဖို့ အသုံးပြုတဲ့ SQL စတိတ်မန်န့် ဖြစ်ပါတယ်။ id နံပါတ် 7 နဲ့ record ရဲ့ grade နဲ့ age ကို update လုပ်မယ်ဆိုရင် ဒီလို

```
UPDATE student SET grade = 'A', age = 19 WHERE id = 7;
```

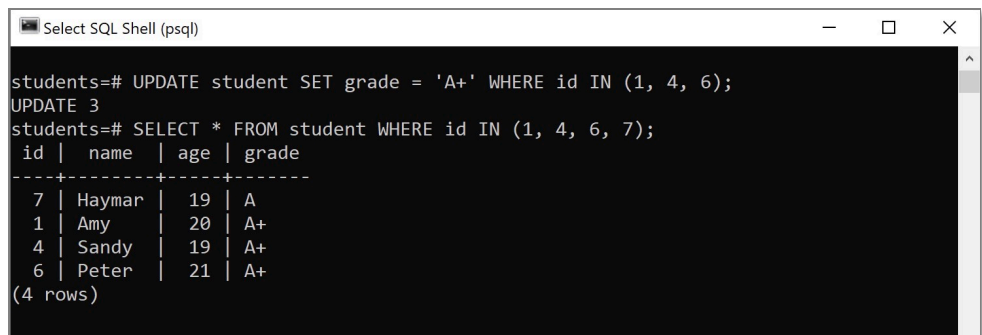
WHERE ပါရင် WHERE ကွန်ဒီရှင်နဲ့ ကိုက်ညီတဲ့ record ကိုပဲ update လုပ်တယ်။ အခု update စတိတ်မန်န့်က id 7 နဲ့ က ဟေမာ့ record တစ်ခုကိုပဲ update လုပ်မှာပါ။ အကယ်၍ WHERE မပါခဲ့ရင် table မှာရှိတဲ့ record တွေ အကုန်လုံးကို update လုပ် သွားလိမ့်မယ်။

```
UPDATE student SET grade = 'A+' WHERE id IN (1, 4, 6);
```

id နံပါတ်က (1,4,6) ထဲမှာပါရင် grade ကို A+ update လုပ်ထားတာပါ။ WHERE ကွန်ဒီရှင်မှာ IN အော်ပရိတ်တာ သုံးထားတယ်။ Column တစ်ခုရဲ့ တန်ဖိုးဟာ list လုပ်ထားတဲ့ တန်ဖိုးတွေထဲမှာ ပါ/မပါ စစ်ချင်ရင် IN သုံးပါတယ်။ Update လုပ်ထားတဲ့ record တွေကို select လုပ်ကြည့်ပါ။

```
SELECT * FROM student WHERE id IN (1, 4, 6, 7);
```

ပုံ (၁.၆) မှာလို တွေ့ရပါမယ်။



```
Select SQL Shell (psql)
students=# UPDATE student SET grade = 'A+' WHERE id IN (1, 4, 6);
UPDATE 3
students=# SELECT * FROM student WHERE id IN (1, 4, 6, 7);
 id | name  | age | grade
----+-----+----+-----
  7 | Haymar | 19  | A
  1 | Amy   | 20  | A+
  4 | Sandy | 19  | A+
  6 | Peter | 21  | A+
(4 rows)
```

ပုံ ၁.၆ record တချို့ကို update လုပ်ပြီးနောက် select ပြန်လုပ်ကြည့်တာ

DELETE

Table row တွေ ဖျက်ဖို့ အသုံးပြုတဲ့ စတိတ်မန်န့် ဖြစ်ပါတယ်။ Student table ထဲက record အားလုံး ဖျက်မယ်ဆိုရင် အခုလို ဖျက်ရမှာပါ

```
DELETE FROM students;
```

Development/testing ဒေတာဘေ့စ်မှာ table ဒေတာ အကုန်ဖျက်ပြီး အစမ်းဒေတာ (test data) ပြန်ထည့်ဖို့ ဒီလို လုပ်ရလေ့ရှိပေမဲ့ တကယ်သုံးနေတဲ့ production ဒေတာဘေ့စ်မှာတော့ table ဒေတာ အကုန်ဖျက်ပစ်ရမဲ့ အခြေအနေဆိုတာ ကြိုတောင့်ကြံခဲ့ပါ။ ဖျက်ရမဲ့ record ကို WHERE နဲ့ရွေး ဖျက်

တာက ပိုပြီး သဘာဝကျပါတယ်။ ဝေယံ ကျောင်းထွက်သွားလို့ သူ့ record ကို သိမ်းထားဖို့ မလိုတော့ဘူး ဆိုပါစို့။ အခုလို delete လုပ်နိုင်ပါတယ်။

```
DELETE FROM student WHERE id = 3; -- id 3 is Waiyan
```

၁.၃ Table Relationships

ဒေတာဘေ့စ် တစ်ခုမှာ table တစ်ခုမက (multiple tables) ပါဝင် ဖွဲ့စည်းထားလေ့ ရှိတယ်။ Table တွေရဲ့ ၎င်းတို့ကြား *relationship* ဟာ Relational Data Model ရဲ့ အဓိကကျတဲ့ သဘောတရားဖြစ်တယ်။ သက်ဆိုင်ရာ table အသီးသီးမှာ အချက်အလက်တွေ စုစည်းသိမ်းဆည်းပုံ သိမ်းဆည်းနည်း စနစ်ကို လေ့လာကြရအောင်။

ဘဏ်တစ်ခုကို စိတ်ကူးကြည့်ပါ။ ဘဏ်အကောင့်၊ အကောင့်ပိုင်ရှင်နဲ့ ငွေဝင်ငွေထွက် စာရင်း အချက်အလက်တွေ သိမ်းဆည်းမယ် ဆိုပါစို့။ Table တစ်ခုတည်းနဲ့ အားလုံးသိမ်းလို့ ရပေမဲ့ Relational Data Model အရ table သုံးခုခွဲပြီး သီးခြားစီသိမ်းတာ ပိုကောင်းပါတယ်။ အကောင့်ပိုင်ရှင်နဲ့ သက်ဆိုင်တဲ့ အချက်အလက်တွေအတွက် table တစ်ခု ရှိပါမယ်။

```
CREATE TABLE account_holder (
    holder_id SERIAL PRIMARY KEY,
    fname VARCHAR(50),
    lname VARCHAR(50),
    dob DATE,
    address TEXT
);
```

အခု table တွေကို ဒေတာဘေ့စ် အသစ်တစ်ခုထဲမှာထားတာ ပိုသင့်တော်မယ်။ (ကျောင်းသား ကိစ္စနဲ့ဆိုင်တဲ့ table တွေကို students ဒေတာဘေ့စ်၊ ဘဏ်နဲ့သက်ဆိုင်တာကို bank ဒေတာဘေ့စ် သတ်သတ်စီခွဲထားရင် ပိုပြီး စနစ်ကျတာကြောင့်ပါ။ အစမ်းလေ့လာတာမို့ သိပ်တော့ အရေးမကြီးပါ။) psql မှာ အောက်ပါအတိုင်း တစ်ကြောင်းချင်း run ပါ

```
\c postgres
CREATE DATABASE bank;
\c bank
```

ပြီးတော့မှ account_holder table အတွက် အပေါ်က CREATE TABLE ကို ဆက် run ပါ။ အခု ချိတ်ထားတာက bank ဒေတာဘေ့စ် (\c bank run ထားတာ သတိပြုပါ)၊ ဆိုတော့ table က အဲဒီထဲမှာ ဆောက်မှာပါ။

အကောင့်နဲ့ သက်ဆိုင်တဲ့ အချက်အလက်တွေအတွက် table တစ်ခု ထပ်ဆောက်ပါမယ်။ (အခု table နှစ်ခုကို အရင်ကြည့်ရအောင်၊ ငွေဝင်ငွေထွက် စာရင်းနဲ့ ဆိုင်တဲ့ တတိယ table ကို နောက်ပိုင်းမှာ တွေ့ရမှာပါ။)

```
CREATE TABLE account (
    acc_id SERIAL PRIMARY KEY,
    /* အခု holder_id က account_holder table ရဲ့ holder_id ကို
    reference လုပ်ထားတယ် */
    holder_id INT REFERENCES account_holder(holder_id),
    acc_no VARCHAR(20) UNIQUE,
```

```
acc_type VARCHAR(20),
balance NUMERIC(12, 2) DEFAULT 0.00
);
```

account table မှာ holder_id column က account_holder table ရဲ့ holder_id column ကို ရည်ညွှန်းထားပါတယ်။ အကောင့်နဲ့ အကောင့် ပိုင်ရှင် အချက်အလက်တွေကို ဒီ table နှစ်ခုမှာ ဘယ်လို ဆက်စပ် သိမ်းဆည်းလဲ နားလည်အောင် နမူနာ record အနည်းငယ်ထည့်ပြီး ရှင်းပြပါမယ်။ အောက်ပါအတိုင်း insert လုပ်ပါ။

```
INSERT INTO account_holder (fname, lname, dob, address)
VALUES
('Amy', 'Moe', '1985-02-15', '123 Main St, Sanchaung'),
('Sandy', 'Soe', '1990-06-23', '456 Oak St, Kamayaut');
```

Insert လုပ်ပြီးရင် အေမီနဲ့ စန္ဒီ holder_id နံပါတ်က 1 နဲ့ 2 အသီးသီး ဖြစ်မယ်။ အေမီဖွင့်ထားတဲ့ အကောင့်နှစ်ခုနဲ့ စန္ဒီရဲ့ အကောင့်တစ်ခုကို အောက်ပါအတိုင်း account table မှာ သိမ်းနိုင်ပါတယ်။ holder_id 1 နဲ့ 2 ကို အထူး ဂရုပြုပါ။

```
INSERT INTO account (holder_id, acc_no, acc_type, balance)
VALUES
(1, '0086-6002-1111', 'Savings', 500000.00),
(1, '0088-6005-1122', 'Current', 800000.00),
(2, '0086-6002-3311', 'Savings', 400000.00);
```

ဒီ table မှာ ကြည့်ရင် အကောင့်ပိုင်ရှင် အသေးစိတ်အချက်အလက်ကို မတွေ့ရပါဘူး။ အကောင့် record တစ်ခုစီအတွက် ပိုင်ရှင်ရဲ့ holder_id ကိုပဲ တွေ့ရမှာပါ။ ဒီ holder_id ဟာ account_holder table ထဲက record တစ်ခုရဲ့ holder_id ကို ရည်ညွှန်းရပါမယ်။

ပထမ အကောင့်နှစ်ခု holder_id က 1 ဖြစ်တယ်။ account_holder table မှာပြန်ကြည့်ရင် holder_id 1 က အေမီ။ ဒါကြောင့် ဒီအကောင့်နှစ်ခုဟာ အေမီရဲ့ အကောင့်ဖြစ်တယ်။ ထိုနည်းတူစွာ holder_id 2 က စန္ဒီဖြစ်တဲ့အတွက် တတိယအကောင့်ဟာ သူမရဲ့ အကောင့်ဖြစ်တယ်လို့ သိနိုင်ပါတယ်။ အခု ဖော်ပြခဲ့သလို အချက်အလက် သိမ်းဆည်းပုံ နည်းစနစ်ဟာ Relational Model ရဲ့ အဓိကကျတဲ့ အခြေခံ သဘောတရားလို့ ဆိုရမှာပါ။

Table JOIN

Table နှစ်ခုကို ပေါင်းစပ်ကြည့်ရင် အကောင့်ရော အကောင့်ပိုင်ရှင် အချက်အလက်ကိုပါ အပြည့်အစုံ သိနိုင်မှာပါ။ ဥပမာ အခုလို select လုပ်ကြည့်ရင် အေမီနဲ့ သူမ၏အကောင့် အသေးစိတ် အချက်အလက်တွေကို တွေ့ရပါမယ်

```
SELECT * FROM account_holder WHERE holder_id = 1;
SELECT * FROM account WHERE holder_id = 1;
```

Output:

holder_id	fname	lname	dob	address
1	Amy	Moe	1985-02-15	123 Main St, Sanchaung

(1 row)

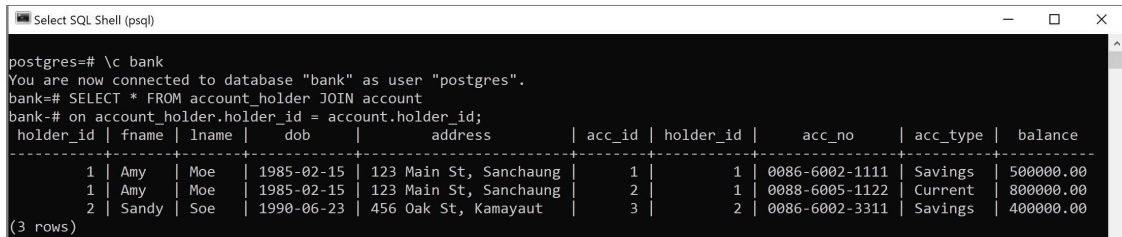
acc_id	holder_id	acc_no	acc_type	balance
1	1	0086-6002-1111	Savings	500000.00
2	1	0088-6005-1122	Current	800000.00

(2 rows)

Table နှစ်ခု ချိတ်ဆက်ပြီး အကောင့်နဲ့ အကောင့်ပိုင်ရှင် တစ်ဆက်တည်း ထုတ်ကြည့်မယ် ဆိုရင် တော့ ဒီအတွက် SQL JOIN ရှိပါတယ်။

```
SELECT * FROM account_holder JOIN account
ON account_holder.holder_id = account.holder_id;
```

Psql မှာ စမ်းကြည့်ရင် အခုလို ရမှာပါ



```

postgres=# \c bank
You are now connected to database "bank" as user "postgres".
bank=# SELECT * FROM account_holder JOIN account
bank=# ON account_holder.holder_id = account.holder_id;

```

holder_id	fname	lname	dob	address	acc_id	holder_id	acc_no	acc_type	balance
1	Amy	Moe	1985-02-15	123 Main St, Sanchaung	1	1	0086-6002-1111	Savings	500000.00
1	Amy	Moe	1985-02-15	123 Main St, Sanchaung	2	1	0088-6005-1122	Current	800000.00
2	Sandy	Soe	1990-06-23	456 Oak St, Kamayaut	3	2	0086-6002-3311	Savings	400000.00

(3 rows)

ပုံ ၁.၇ table နှစ်ခု join ပြီး select လုပ်ထားတာ

account_holder table ရဲ့ holder_id နဲ့ account table ရဲ့ holder_id တူရင် JOIN က record တွေကို တွဲဆက်ပေးတာ တွေ့ရမှာပါ။ တွဲဆက်ပေးရမဲ့ ကွန်ဒီရှင်ကို ON နောက်မှာ အခုလို ထည့် ပေးထားတယ်

```
ON account_holder.holder_id = account.holder_id;
```

အောက်မှာ နောက်ထပ် ပုံစံတစ်မျိုး ရေးထားတာကို ကြည့်ပါ။ account_holder table ကို t1 ၊ account ကို t2 နဲ့ ရည်ညွှန်းပါတယ်။ Alias လုပ်တာလို့ ခေါ်ပါတယ်။

```

SELECT
    t1.*,
    t2.*
FROM account_holder t1 JOIN account t2
    ON t1.holder_id = t2.holder_id;
```

Table နှစ်ခုကနေ လိုချင်တဲ့ column ကိုပဲ ရွေးထုတ်လည်း ရတယ်။ ဥပမာ

```

SELECT
    t2.*,
    t1.fname,
    t1.lname
```

```
FROM account_holder t1 JOIN account t2
ON t1.holder_id = t2.holder_id;
```

Referential Integrity

Relational Model ဟာ *referential integrity* ကို မပျက်ယွင်းအောင် အလေးအနက်ထား ထိန်းသိမ်းပေးပါတယ်။ Record တစ်ခုကို ဖျက်တဲ့အခါ အဲဒီဖျက်လိုက်တဲ့ record ကို အခြား table မှာရှိတဲ့ record တွေက ရည်ညွှန်းထားမယ်ဆိုရင် ပြဿနာရှိတယ်။ ဥပမာ account_holder table မှာ အကောင့်ပိုင်ရှင် အမှီ record ကို ဖျက်လိုက်တယ် ဆိုပါစို့။ ဒီလိုဆိုရင် account table ထဲက holder_id 1 နဲ့ အကောင့်နှစ်ခု ရည်ညွှန်းထားတဲ့ အကောင့်ပိုင်ရှင် record ရှိမှာ မဟုတ်တော့ဘူး။ ဒါဟာ referential integrity ကို ချိုးဖောက်တာ ဖြစ်တဲ့အတွက် Relational Model က အဲဒီလို ဖျက်ခွင့်ပေးမှာ မဟုတ်ပါဘူး။ Record တစ်ခုကို အခြား record တွေက reference လုပ်ထားတာ ရှိနေသမျှ Relational Model က အဲဒီ record ပေးမဖျက်ဘူး။ အခုလို စမ်းပြီး ဖျက်ကြည့်ပါ။ ဖျက်ခွင့်မပေးတာကို

```
Select SQL Shell (psql)
bank=# DELETE FROM account_holder WHERE holder_id = 1;
ERROR: update or delete on table "account_holder" violates foreign key constraint "
account_holder_id_fkey" on table "account"
DETAIL: Key (holder_id)=(1) is still referenced from table "account".
bank=#
```

ပုံ ၁.၈ referential integrity အရ ပေးမဖျက်တာ

တွေ့ရပါလိမ့်မယ်။

holder_id 1 နဲ့ record ကို ဖျက်ချင်ရင် ၎င်းကို ရည်ညွှန်းတဲ့ record တွေကို အရင်ဖျက်ရပါမယ်။ ဒါမှမဟုတ် နောက်ထပ်နည်းလမ်းတစ်ခုက parent record ကိုဖျက်ရင် ဆက်စပ်နေတဲ့ child record တွေကိုပါ အလိုအလျောက် ဖျက်အောင် ON DELETE CASCADE option အသုံးပြုတာပါ။ ရည်ညွှန်းတဲ့ table/record ကို child table/record လို့ ယူဆပါ။ ရည်ညွှန်းခြင်း ခံရတဲ့ table/record ကို parent table/record လို့ ယူဆပါ။ ON DELETE CASCADE ကို child table ဆောက်တဲ့အခါ holder_id column မှာ အခုလို သတ်မှတ်ပေးရမှာပါ။

```
CREATE TABLE account (
    acc_id SERIAL PRIMARY KEY,
    holder_id INT REFERENCES account_holder(holder_id) ON DELETE CASCADE,
    ...
);
```

Referential integrity နဲ့ ပါတ်သက်ပြီး နားလည်ထားဖို့ လိုပါတယ်။ ဒါကြောင့် အကျဉ်းချုံး ရှင်းပြထားတာပါ။ ON DELETE CASCADE စမ်းကြည့်မယ်ဆို ဒေတာဘေ့စ် ကိုဖျက်ပြီး table ပြန်ဆောက် (သို့) ဒေတာဘေ့စ် အသစ်တစ်ခု ဆောက်ပြီး စမ်းကြည့်ပါ။ ရှိပြီးသား table ကို ON DELETE CASCADE ဖြစ်အောင် လုပ်လိုရပေမဲ့ နည်းနည်းပိုရှုပ်ထွေးပါတယ်။

Relationship Between account and account_transaction Tables

Table relationship နဲ့ JOIN ကို ပိုပြီး သဘောပေါက်အောင် အားဖြည့်တဲ့အနေနဲ့ နောက်ထပ် ဥပမာ တစ်ခု ကြည့်ရအောင်။ အကောင့် ငွေသွင်း/ထုတ်စာရင်း (account transaction) table ပါ။

```
CREATE TABLE account_transaction (
    txn_id SERIAL PRIMARY KEY,
    -- reference to account table by acc_id
    acc_id INT REFERENCES account(acc_id),
    txn_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    txn_type VARCHAR(20),
    amount NUMERIC(12, 2),
    balance_after NUMERIC(12, 2)
);
```

ဒီ table မှာ acc_id column က account table ရဲ့ acc_id column ကို reference လုပ်ထားတာ ဂရုပြုကြည့်ပါ။ အောက်ပါ ငွေသွင်း/ထုတ် စာရင်း (transaction) ၅ ခု ထည့်ပါ

```
INSERT INTO account_transaction
    (acc_id, txn_date, txn_type, amount, balance_after)
VALUES
    (1, '2024-08-01 09:00:00', 'Deposit', 100000.00, 600000.00),
    (1, '2024-08-05 14:30:00', 'Withdrawal', 50000.00, 550000.00),
    (2, '2024-08-02 10:00:00', 'Deposit', 200000.00, 1000000.00),
    (2, '2024-08-03 16:00:00', 'Withdrawal', 300000.00, 700000.00),
    (3, '2024-08-04 11:00:00', 'Deposit', 50000.00, 450000.00);
```

Table နှစ်ခုကို acc_id နဲ့ ဘယ်လို ချိတ်ဆက်ထားလဲ နားလည်ဖို့ အရေးကြီးတယ်။ ပထမ transaction နှစ်ခုနဲ့ သက်ဆိုင်တဲ့ အကောင့် အချက်အလက် အသေးစိတ်ကို သိချင်ရင် account table မှာ acc_id နံပါတ် 1 နဲ့ record ကို ကြည့်ရမှာပါ

```
SELECT * FROM account WHERE acc_id = 1;
```

Output:

acc_id	holder_id	acc_no	acc_type	balance
1	1	0086-6002-1111	Savings	500000.00

account_transaction table မှာ transaction record တွေ insert လုပ်တဲ့အခါမှာလည်း ၎င်းတို့နှင့် သက်ဆိုင်တဲ့ acc_id ကို မှန်ကန်အောင် သေချာစိစစ်ဖို့ လိုတယ်။ Table နှစ်ခုက အချက်အလက်တွေကို JOIN နဲ့ ချိတ်ဆက် ထုတ်ယူနိုင်တယ်။

```
SELECT
    *
FROM account t1 JOIN account_transaction t2
    ON t1.acc_id = t2.acc_id;
```

Table နှစ်ခုမကလည်း JOIN လို့ရတယ်။ ဥပမာ

```
SELECT
    t1.holder_id,
    t1.fname,
    t1.lname,
```

```

        t2.acc_no,
        t2.acc_type,
        t3.txn_type,
        t3.amount,
        t3.balance_after
FROM account_holder t1 JOIN account t2
    ON t1.holder_id = t2.holder_id
JOIN account_transaction t3
    ON t2.acc_id = t3.acc_id;

```

Table သုံးခုကြား relationship ကို နားလည်အောင် နောက်ဆုံးတစ်ခါ အောက်ပါတို့ကို ဆက်စပ်ကြည့်ပါ။ Insert တစ်ခါလုပ်ပြီး auto-generated id နံပါတ်တွေ select လုပ်ကြည့်ပါ။

```

INSERT INTO account_holder (fname, lname, dob, address)
VALUES
('Waiyan', 'Phyo', '1991-07-22', '45 Bawga St, Yankin');

```

```

-- Before insert, make sure Waiyan's holder_id is 3
INSERT INTO account (holder_id, acc_no, acc_type, balance)
VALUES
(3, '0086-6002-4411', 'Savings', 700000.00);

```

```

-- Before insert, make sure Waiyan's acc_id is 4
INSERT INTO account_transaction
    (acc_id, txn_date, txn_type, amount, balance_after)
VALUES
(4, '2024-08-05 15:45:00', 'Deposit', 200000.00, 900000.00);

```

၁.၄ SQL ဖန်ရှင်များ

SQL မှာ built-in ဖန်ရှင်တွေ ပါရှိပါတယ်။ to_char, upper နဲ့ concat သုံးထားတာ ကြည့်ပါ။ Column ကို alias ပေးလိုရတယ်။ Date_of_Birth နဲ့ Full_Name က alias တွေ။

```

SELECT
    to_char(dob, 'Mon DD YYYY') Date_of_Birth,
    upper(concat(fname, ' ', lname)) Full_Name
FROM account_holder;

```

Output:

```

date_of_birth | full_name
-----+-----
Feb 15 1985   | AMY MOE
Jun 23 1990   | SANDY SOE
Jul 22 1991   | WAIYAN PHYO
(3 rows)

```

Format Code	Format
YYYY	Year (4 digits)
YY	Year (last 2 digits)
MM	Month (01-12)
MON	Abbreviated month name (e.g., AUG)
MONTH	Full month name (e.g., AUGUST)
DD	Day of the month (01-31)
HH24	Hour (24-hour clock, 00-23)
HH12	Hour (12-hour clock, 01-12)
MI	Minute (00-59)
SS	Second (00-59)
AM/PM	Meridian indicator

ဇယား ၁.၁ PostgreSQL Date and Time Format Codes

SQL date (သို့) datetime data type ကနေ လိုချင်တဲ့ format ကို to_char ဖန်ရှင်နဲ့ ပြောင်းလို့ရတယ်။ 'Mon DD YYYY' မှာ Mon က month ကို Feb, Jun, Jul အတိုကောက်ပြဖို့။ Format codes တွေကို ဇယား (၁.၄) တွင် ကြည့်ပါ။

အောက်ပါအတိုင်း အလွယ်တကူ စမ်းကြည့်လို့ ရပါတယ်။ လက်ရှိအချိန်ကို now() နဲ့ ယူတယ်။ ဒီလို format 15/08/2024 10:41:36 နဲ့ ထုတ်ပေးမှာပါ။

```
SELECT to_char(now(), 'DD/MM/YYYY HH24:MI:SS') formatted_datetime;
```

အခြား ဖန်ရှင်တွေ အများကြီး ရှိပါသေးတယ်။ ဒီစာအုပ်မှာတော့ ဒီလောက်ပဲ အကျဉ်း ဖော်ပြပေးနိုင်ပါတယ်။ ဘီဂင်နာတွေ ဆက်လက်လေ့လာဖို့ စာအုပ်၊ YouTube နဲ့ tutorial လင့်ခ်တွေကို ဒီအခန်းအဆုံးမှာ ပေးထားတယ်။

၁.၅ Python နှင့် ဒေတာဘေ့စ် ပရိုဂရမ်မင်း

ဆော့ဖ်ဝဲ အပ်ပလီကေးရှင်းတွေဟာ ဒေတာဘေ့စ်နဲ့ ချိတ်ဆက်လုပ်ဆောင် ရလေ့ရှိတယ်။ 'End User' လို့ခေါ်တဲ့ အသုံးပြုသူတွေဟာ အပ်ပလီကေးရှင်း User Interface (UI) ကနေတစ်ဆင့် ဒေတာဘေ့စ်ကို သုံးကြတာပါ။ ပုံမှန်အားဖြင့် end user အများစုဟာ ဒေတာဘေ့စ်ကို တိုက်ရိုက် အသုံးမပြုကြပါဘူး။ ဒါကြောင့် end user တွေလိုအပ်မဲ့ ဒေတာသွင်းတာ၊ ပြန်ရှာ/ဖတ်တာ၊ အပ်ဒိတ်လုပ်တာ၊ ဖျက်တာ စတဲ့ ကိစ္စတွေအတွက် အပ်ပလီကေးရှင်းတွေကပဲ ထည့်သွင်းစဉ်းစား တည်ဆောက်ပေးရတာပါ။

ကျောင်းသားစာရင်းသွင်း ပရိုဂရမ်တစ်ခုကို စိတ်ကူးကြည့်ပါ။ ကျောင်းသားသစ် ကိုယ်ရေး အချက်အလက် UI နမူနာကို ပုံ (၁.၉) မှာ ပြထားတာ ကြည့်ပါ။ Submit နှိပ်လိုက်ရင် ဖြည့်ထားတဲ့ ကျောင်းသား အချက်အလက်တွေကို ဒေတာဘေ့စ်ထဲ သိမ်းရပါမယ်။ ထိုနည်းတူစွာ ပြန်ရှာတာ၊ ဖျက်တာ၊ update လုပ်တာ ကိုလည်း UI ကနေပဲ လုပ်လို့ရအောင် ပရိုဂရမ်က စီစဉ်ပေးထားရမှာပါ။

ဒီလို အပ်ပလီကေးရှင်းမျိုးတွေကို ဒေတာဘေ့စ် အပ်ပလီကေးရှင်းလို့ ခေါ်ပါတယ်။ အချက်အလက်တွေ create, read, update, and delete လုပ်တဲ့ အခြေခံလုပ်ဆောင်ချက် လေးခု ပါဝင်တာကြောင့် မို့ CRUD အပ်ပလီကေးရှင်းလို့လည်း ခေါ်ကြတယ်။

registration form

Form

Name: Sandy Soe

Course: Python Programming

Semester: First

Form No.: 2021

Contact No.: +955005433

Email id: sandy@gmail.com

Address: 133/A Kyaung St. Kyauk Myaung

Submit

ပုံ ၁.၉

Psycpg ဒေတာဘေ့ခ် ဒရိုက်ဗာ

Python နဲ့ PostgreSQL ချိတ်ဆက်လုပ်ဆောင်ဖို့အတွက် Psycpg (ဆိုက်ကော့ပ်ဂျီ) ဟာ လူကြိုက်အများဆုံး ဒေတာဘေ့ခ် ဒရိုက်ဗာ ဖြစ်ပါတယ်။ ဒေတာဘေ့ခ် ဒရိုက်ဗာဆိုတာ ဒေတာဘေ့ခ်နဲ့ programming language ကြား ပေါင်းကူးတံတားအဖြစ် ဆောင်ရွက်ပေးတဲ့ လိုက်ဘရီပါ။ ဒေတာဘေ့ခ် အဒပ်တာ (adapter) လို့လည်းခေါ်တယ်။

DBMS နဲ့ programming language အလိုက် သက်ဆိုင်ရာ ဒေတာဘေ့ခ် ဒရိုက်ဗာကို ရွေးချယ်အသုံးပြုရမှာပါ။ PostgreSQL အတွက် Python မှာ Psycpg 2 နဲ့ Psycpg 3 ရှိမယ်။ အခြားဟာတွေလည်း ရှိပါသေးတယ်။ MySQL အတွက် MySQL Connector/Python သုံးကြတယ်။ Microsoft SQL အတွက်ဆိုရင် pyodbc ဒရိုက်ဗာ။

Psycpg 3 က ဗားရှင်းပိုမြင့်ပေမဲ့ ဒီစာအုပ်မှာ Psycpg 2 ကိုပဲ အသုံးပြုပါမယ်။ Psycpg 2 သုံးတတ်ရင် Psycpg 3 ပြောင်းသုံးလည်း အခက်အခဲ သိပ်မရှိနိုင်ဘူး။ ဘီဂင်နာအတွက် စလေ့လာရတာ ပိုအဆင်ပြေမယ် ယူဆတာကြောင့် Psycpg 2 သုံးဖို့ ဆုံးဖြတ်ရတာပါ။ နောက်ပိုင်း အတွေ့အကြုံရှိလာရင် ဗားရှင်းအမြင့်ကို ဆက်လေ့လာလို့ရပါတယ်။ Psycpg 2 ကို အောက်ပါအတိုင်း pip ကွန်မန်းနဲ့ အင်စတောလ်လုပ်ပါ။

```
pip install psycpg2-binary
```

Connecting to PostgreSQL

ဒေတာဘေ့ခ် အပ်ပလီကေးရှင်းတွေဟာ client-server အပ်ပလီကေးရှင်းတွေလို့ ရှေ့ပိုင်းမှာ ပြောခဲ့တာမှတ်မိမယ်ထင်ပါတယ်။ အချက်အလက်တွေ အပြန်အလှန် ပေးပို့လုပ်ဆောင်နိုင်ဖို့ client နဲ့ server ကြား ကွန်နက်ရှင် ချိတ်ဆက်ဖို့လိုတယ်။ ဒီလိုချိတ်ဆက်တဲ့အခါ TCP/IP (Transmission Control Protocol/Internet Protocol) ကို အသုံးပြုရပါတယ်။ CRUD အော်ပရေးရှင်း တစ်ခုခု လုပ်မယ်ဆို ပထမဆုံး ကွန်နက်ရှင်အရင် ရှိထားရမှာပါ။ ကွန်နက်ရှင်မရရင် ဒေတာဘေ့ခ်နဲ့ ပါတ်သက်တဲ့ အခြားကိစ္စတွေ ဆက်လုပ်လို့ မရနိုင်ဘူး။

ပထမ ဥပမာအနေနဲ့ Python ပရိုဂရမ်ကနေ PostgreSQL ဒေတာဘေ့ခ် အသစ်တစ်ခု ဘယ်လို ဆောက်ရမလဲ ကြည့်ရအောင်။

```
# File: db_connect.py
import psycopg2

# Connect to the default PostgreSQL database to create
# the new "students" database
conn = psycopg2.connect(
    dbname="postgres",
    user="postgres",
    password="asdfgh",
    host="localhost",
    port="5432"
)

conn.autocommit = True
cur = conn.cursor()

# Create the "students" database
cur.execute("DROP DATABASE IF EXISTS students")
cur.execute("CREATE DATABASE students")

# Close the initial connection
cur.close()
conn.close()
```

ဒီပရိုဂရမ်ကို တစ်ခုချင်း ခွဲခြမ်းစိတ်ဖြာ ကြည့်ပါမယ်။ psycopg2 လိုက်ဘရီ ပထမဆုံး အင်ပို့ လုပ်ထားတယ်။ ပြီးတော့ connect ဖန်ရှင်နဲ့ ကွန်နက်ရှင်ယူတယ်။ connect ဖန်ရှင် ပါရာမီတာတွေက psycopg2 ကွန်နက်ရှင်လုပ်တဲ့အခါ ထည့်ပေးရတာတွေနဲ့ တူတူပါပဲ။

- dbname (ချိတ်ဆက်မဲ့ ဒေတာဘေ့စ်နံမည်)
- user (ဒေတာဘေ့စ် ချိတ်ဆက်အသုံးပြုမဲ့ user)
- password (ဒေတာဘေ့စ်ကို ချိတ်ဆက်အသုံးပြုမဲ့ user ရဲ့ password)
- host (ချိတ်ဆက်မဲ့ ဆာဗာရဲ့ Host Name/IP Address)
- port (PostgreSQL port နံပါတ်)

Connect လုပ်တာ အောင်မြင်ရင် connect ဖန်ရှင်က connection အော့ဘ်ဂျက်တစ်ခု ပြန်ရပါတယ်။ တကယ်လို့ ပြဿနာ တစ်ခုခုကြောင့် connect လုပ်လို့ မရရင်တော့ ဖြစ်ရတဲ့ အကြောင်းအရင်းပေါ် မူတည်ပြီး OperationalError, InternalError စတဲ့ exception တွေ တက်နိုင်တယ်။ ဒီ exception တွေက psycopg2.Error ရဲ့ subclass တွေပါ။

```
conn.autocommit = True
```

ဒါက ဒေတာဘေ့စ် auto-commit mode ကို on လုပ်ပေးဖို့။ ဒေတာဘေ့စ် transaction နဲ့ ဆိုင်တဲ့ setting တစ်ခုဖြစ်ပြီး နောက်ပိုင်းမှာ အသေးစိတ် ရှင်းပြမှာပါ။ Psycopg က သူ့နဂိုအတိုင်းဆိုရင် auto-commit mode ကို off လုပ်ထားတယ်။ ဒေတာဘေ့စ် အသစ်ဆောက်တဲ့ CREATE DATABASE လို တချို့ SQL စတိတ်မန်တွေက auto commit ကို on လုပ်ပေးရတယ်လို့ လောလောဆယ် သိထားရင် လုံလောက်ပါပြီ။ Auto-commit on ထားရင် SQL စတိတ်မန်တွေ execute လုပ်ပြီးရင် commit

ဖန်ရှင် ခေါ်ဖို့ မလိုဘူး။ (နောက် ဥပမာမှာ commit ဖန်ရှင် သုံးထားတာ တွေ့ရမှာပါ။)

SQL စတိတ်မန်တွေ ဒေတာဘေ့စ်ဆီ ပေးပို့လုပ်ဆောင်စေခြင်း၊ ဒေတာဘေ့စ်ဆီက ပြန်ရလာတဲ့ ရလဒ်တွေကို အသုံးပြုခြင်း၊ ဒေတာဘေ့စ် transaction စီမံခြင်း စတဲ့ ကိစ္စတွေအတွက် cursor က အဓိကကျတယ်။ Cursor အော့ဘ်ဂျက်ကို connection ကနေ တစ်ဆင့် အခုလို ယူရပါတယ်

```
cur = conn.cursor()
```

ချိတ်ဆက်ထားတဲ့ ဒေတာဘေ့စ်ဆီကို SQL စတိတ်မန်တွေ ပေးပို့ လုပ်ဆောင်ခိုင်းဖို့ execute ဖန်ရှင် အသုံးပြုတယ်။

```
cur.execute("DROP DATABASE IF EXISTS students")
cur.execute("CREATE DATABASE students")
```

ပထမ တစ်ကြောင်းက students ဒေတာဘေ့စ် ရှိပြီးသားဆိုရင် ဖျက်ပစ်မှာပါ။ ပြီးတော့မှ ဒုတိယ တစ်ကြောင်းမှာ ဒေတာဘေ့စ် အသစ်ဆောက်ပါတယ်။ ရှိပြီးသားဆိုရင် ဖျက်ပြီး အသစ်ပြန်ဆောက်တဲ့ သဘောပါ။ psql နဲ့ဆိုရင် SQL စတိတ်မန်အဆုံးမှာ ; ထည့်ပြီး run ရမှာ ဖြစ်ပေမဲ့ Psycpg မှာတော့ ထည့်စရာ မလိုဘူး။ ထည့်ပေးလည်း ပြဿနာတော့ မရှိဘူး။

Cursor နဲ့ connection ကို အသုံးပြုပြီးသွားရင် ပိတ်ပေးရပါမယ်။ Exception handle လုပ်မယ်ဆိုရင် finally ထဲမှာ ပိတ်သင့်တယ်။ ဒီဥပမာမှာတော့ ဒီတိုင်းပဲ ပိတ်ထားပါတယ်။

```
cur.close()
conn.close()
```

ဒီပရိုဂရမ် run ပြီး error လည်း မတက်ဘူးဆိုရင် students ဒေတာဘေ့စ် ဆောက်ပြီးသွားမှာပါ။ psql ကနေ \1 ကွန်မန်းနဲ့ စစ်ကြည့်နိုင်ပါတယ်။

ဒေတာဘေ့စ်ထဲမှာ table တွေ ဆောက်ပါမယ်။ လက်ရှိအသုံးပြုမဲ့ ဒေတာဘေ့စ်နဲ့ connection အရင်ယူရမယ်။ students ဒေတာဘေ့စ်မှာ table ဆောက်မှာ။ ဒီတော့ ချိတ်ဆက်မဲ့ dbname က students ဖြစ်တယ်။ ကျန်တာတွေက ရှေ့ကနဲ့ တူတူပဲ။

```
# File: db_create_student_tbl.py
import psycopg2

# Now, connect to the "students" database to create the "student" table
conn = psycopg2.connect(
    dbname="students",
    user="postgres",
    password="asdfgh",
    host="localhost",
    port="5432"
)
conn.autocommit = False
cur = conn.cursor()

# Create the "student" table
cur.execute("""
    CREATE TABLE student (
```



```

        id SERIAL PRIMARY KEY,
        name VARCHAR(100),
        age INT,
        grade VARCHAR(2)
    )
"""

# Commit changes and close the connection
conn.commit()
cur.close()
conn.close()

```

ကွန်နက်ရှင် ရပြီး နောက်မှာ

```
conn.autocommit = False
```

နဲ့ auto-commit mode ကို off လုပ်တယ်။ Psycopg default က auto-commit off ဖြစ်တဲ့ အတွက် ဒါမပါရင်လည်း off ဖြစ်နေမှာပါ။ ဒါဆို ဘာလို့ တကူးတက ထည့်ထားလည်းဆိုတော့ ကုဒ်ကို ဖတ်တဲ့အခါ auto-commit off ထားတယ်ဆိုတာ သိသာစေချင်တာရော ဒရိုက်ဗာရဲ့ default က on/off ဘာလဲ မသေချာမှာ စိုးလို့ပါ။

CREATE TABLE စတိတ်မန်က auto-commit mode on ဖြစ်ဖြစ်၊ off ဖြစ်ဖြစ် run လို့ရ တယ်။ အခု off လုပ်ထားတာက ရှေ့ကဥပမာမှာ on လုပ်ထားတာနဲ့ ကွာခြားချက်တချို့ကို သိအောင်လို့ ပါ။ အခြားထူးခြားတဲ့ အကြောင်းအရင်း မရှိပါဘူး။ နောက်ပိုင်း database transaction အပိုင်းမှာ off လုပ်ကို လုပ်ရတဲ့ အကြောင်းအရင်းကို ရှင်းပြပါမယ်။

Auto-commit off ထားရင် SQL စတိတ်မန် execute လုပ်ပြီး commit လုပ်ပေးဖို့ လိုတယ်။ ဒီအတွက်

```
conn.commit()
```

လုပ်ရပါမယ်။ commit မလုပ်မီရင် execute လုပ်ထားတဲ့ SQL က သက်ရောက်မှု ရှိမှာမဟုတ်ပါဘူး။ ဆိုလိုတာက student table ဆောက်တဲ့ကိစ္စက အတည်ဖြစ်သွားဘူး (rollback ဖြစ်သွားတယ်လို့ ခေါ်တယ်)။ Commit နဲ့ rollback အကြောင်း transaction အပိုင်းမှာ ရှင်းပြပါမယ်။

အကယ်၍ Auto-commit on ထားရင်တော့ ကိုယ်တိုင် conn.commit() လုပ်မပေးရဘူး၊ ဒေ တာဘေ့စ်က SQL ကွန်မန်း တစ်ခု execute လုပ်ပြီးတိုင်း အလိုအလျောက် commit လုပ်ပေးမှာပါ။ Off ထားရင် commit လုပ်ပေးရမယ်၊ on ဆိုရင် မလုပ်ပေးရဘူးလို့ မှတ်နိုင်ပါတယ်။

၁.၆ Parameterized SQL

SQL စတိတ်မန်တွေမှာ ပါရာမီတာ ထည့်လို့ရအောင် ဒေတာဘေ့စ် ဒရိုက်ဗာတွေက ထောက်ပံ့ပေးထားပါ တယ်။ ပါရာမီတာပါတဲ့ SQL ကို *Parameterized SQL* လို့ခေါ်တယ်။ ပုံမှန် ရိုးရိုး SQL နဲ့ param-eterized SQL အသုံးပြုပုံ နှစ်ခုယှဉ်ကြည့်ပါ

```

# normal SQL, no parameter
cur.execute("SELECT * FROM student WHERE age = 19 AND grade = 'A'")

```

```
# File: db_parameterized_sql.py
# parameterized SQL
age = int(input("Age: "))
grade = input("Grade: ")
cur.execute("SELECT * FROM student WHERE age = %s AND grade = %s",
            (age, grade))
```

%s က ပါရာမီတာ၊ ပါရာမီတာတွေနေရာ ရှိရမဲ့ တန်ဖိုးအသီးသီးကို tuple နဲ့ ထည့်ပေးရပါမယ်။ execute လုပ်တဲ့အခါ ပါရာမီတာတစ်ခုစီကို သက်ဆိုင်ရာတန်ဖိုး အစားထိုးပေးမှာပါ။ Tuple ဟာ list နဲ့ဆင်တူတဲ့ စထရက်ချာ တစ်မျိုးပါပဲ။ ကွာခြားချက် တချို့ကတော့ list ကို [] သုံးတယ်၊ tuple အတွက် () သုံးတယ်။ Tuple က immutable ဖြစ်ပြီး item တစ်ခုပဲဆိုရင် ကော်မာထည့်ပေးရပါမယ်။ `val1` တစ်ခုပဲဆို (`val1,`) လို့ ရေးရမှာပါ။ Parameterized SQL မှာ ပါရာမီတာ တစ်ခုတည်းဆိုရင် ဒီအချက်ကို သတိပြုဖို့ လိုပါတယ်။ အောက်မှာ (`stu_id,`) ဖြစ်ရပါမယ်။ ကော်မာ မထည့်ဘဲ (`stu_id`) လို့ ရေးမိရင် အယ်ရာတက်မှာပါ။

```
stu_id = int(input("ID: "))
cur.execute("SELECT * FROM student WHERE id = %s", (stu_id,))
```

Parameterized SQL နဲ့ insert ဘယ်လိုလုပ်ထားလဲ အောက်မှာ လေ့လာကြည့်ပါ။ students list ထဲမှာ ကျောင်းသားတစ်ယောက်ချင်းအတွက် tuple တွေ ထည့်ထားပြီး for loop နဲ့ တစ်ခုချင်း insert လုပ်သွားတာကို ဂရုစိုက် နားလည်အောင် ကြည့်ပါ။

```
# File: db_insert1.py
import psycopg2

conn = psycopg2.connect(dbname="students", user="postgres",
                        password="asdfgh", host="localhost", port="5432")
conn.autocommit = False
cur = conn.cursor()

# Insert records into the student table one at a time
students = [
    ('Amy', 20, 'A'),
    ('Sandy', 22, 'B'),
    ('Kathy', 21, 'C')
]

for student in students:
    cur.execute("""
        INSERT INTO student (name, age, grade)
        VALUES (%s, %s, %s)
    """, student)

conn.commit()
cur.close()
conn.close()
```

ရှေ့က ဥပမာဟာ အောက်ပါ insert စတိတ်မန့် သုံးခုကို တစ်ခုပြီးတစ်ခု ဒေတာဘေ့ခ်ဆီ ပေးပို့ လုပ်ဆောင်စေမှာပါ။

```
INSERT INTO student (name, age, grade) VALUES ('Amy', 20, 'A');
INSERT INTO student (name, age, grade) VALUES ('Sandy', 22, 'B');
INSERT INTO student (name, age, grade) VALUES ('Kathy', 21, 'C');
```

Insert သုံးခုလုံး တစ်ခါတည်းနဲ့ ပေးပို့ လုပ်ဆောင်စေချင်ရင် executemany ကို သုံးပါတယ်။ Tuple array တစ်ခု ထည့်ပေးရပါမယ်။ Array ထဲက tuple တစ်ခုချင်းအတွက် SQL စတိတ်မန့် တစ်ခုစီ ထုတ်ပေးပြီး အသုတ်လိုက် ဒေတာဘေ့ခ်ဆီ ပို့ပေးပါတယ်။

```
# File: db_insert2.py
students = [
    ('Amy', 20, 'A'),
    ('Sandy', 22, 'B'),
    ('Kathy', 21, 'C')
]

cur.executemany("""
    INSERT INTO student (name, age, grade)
    VALUES (%s, %s, %s)
""", students)
```

စောစောကဥပမာနဲ့ ရလဒ်အားဖြင့်တော့ တူတူပါပဲ။ တစ်ကြောင်းချင်း ပို့တာနဲ့ သုံးကြောင်းလုံး အသုတ် လိုက် တစ်ခါတည်း ပို့တာပဲ ကွာပါတယ်။ အသုတ်လိုက် ပို့တာက နက်ဝပ်အသွားအပြန် (network round-trip) လုပ်ရတာ မများတဲ့အတွက် record တွေ အများကြီး insert လုပ်တဲ့အခါ သိသိသာသာ ပိုပြီးတော့မြန်မှာပါ။ တစ်ကြောင်းချင်း ပို့တာက တစ်ခါပို့တိုင်း နက်ဝပ်အသွားအပြန် ရှိတဲ့အတွက် ပိုကြာ မယ်။ နှစ်မျိုးလုံး သူ့နေရာနဲ့သူ ချင့်ချိန်အသုံးပြုရမှာပါ။ ဒါနဲ့ပါတ်သက်လို့ အခုဆက်လက် ဆွေးနွေးမှာ မဟုတ်ပေမဲ့ ဘယ်လိုအခြေအနေမျိုးမှာ ဘယ်ဟာသုံးသင့်လဲ ဆုံးဖြတ်တတ်အောင် ဆက်လက်လေ့လာဖို့ လိုပါလိမ့်မယ်။

၁.၇ fetch-ing and Using Results

Select စတိတ်မန့်ရဲ့ ရလဒ် record တွေကို fetchall, fetchmany, fetchone ဖန်ရှင်တွေနဲ့ ရယူ အသုံးပြုနိုင်ပါတယ်။

fetchall က record အကုန်လုံးကို တစ်ခါတည်းနဲ့ ရယူပေးမှာပါ။ Select လုပ်တဲ့ record အရေအတွက် များလွန်းရင် ဒေတာဘေ့ခ်ကနေ Python ပရိုဂရမ်ဘက်ကို ဒေတာအားလုံး လွှဲပြောင်း ရောက်ရှိချိန် ကြာနိုင်တဲ့အပြင် ရောက်ရှိလာတဲ့ record တွေအတွက် ကွန်ပျူတာ မမ်မိုရီသုံးရတာက လည်း ပမာဏ များနိုင်ပါတယ်။ fetchall သုံးမယ်ဆိုရင် record အရေအတွက်ကို ထည့်စဉ်းစားဖို့ လိုပါမယ်။

```
# File: db_select_and_fetch.py
cur.execute("SELECT * FROM student")

rows = cur.fetchall()
for row in rows:
```

```
print(row)
```

fetchone ကိုတော့ record တစ်ခုပဲ ပြန်ရမှာ သေချာတဲ့ အခြေအနေမျိုးမှာ သုံးပါတယ်။ ဥပမာ primary key နဲ့ select လုပ်တဲ့အခါ record တစ်ခုထက် ပိုမရနိုင်ဘူး။

```
# File: db_select_and_fetch.py
cur.execute("SELECT * FROM student WHERE id = 1")
row = cur.fetchone()
print(row)
```

fetchmany ကတော့ လိုသလောက် record အရေအတွက်ကိုပဲ ရယူဖို့အတွက်ပါ။ Record အခု ၁၀၀ ရှိတာကို တစ်ခေါက်ကို ၂၀ နဲ့ ၅ ခါ ခွဲပြချင်တဲ့ အခါမျိုးမှာ သုံးနိုင်ပါတယ်။

```
rows = cur.fetchmany(20)
```

Pagination

Record တွေ အများကြီး တစ်ခါတည်းနဲ့ ပြမဲ့အစား နည်းနည်းချင်း ခွဲပြီး ပြပေးတဲ့နည်းကို အပ်ပလီကေးရှင်းတွေမှာ အသုံးများတာ တွေ့ရပါတယ်။ Pagination လို့ ခေါ်ပါတယ်။ Page တစ်ခုမှာ သတ်မှတ်ထားတဲ့ record အရေအတွက်ကို ပြပေးပြီး user က နောက် page တစ်ခုချင်း ဆက်ကြည့်လို့ရအောင် စီစဉ်ပေးထားတာပါ။ အောက်ပါ ဥပမာမှာ fetchmany သုံးထားတာ လေ့လာကြည့်ပါ။

```
# File: db_pagination1.py
# Function to fetch a page of student records using fetchmany
def fetch_page(page_size):
    # Fetch rows in batches using fetchmany
    rows = cur.fetchmany(page_size)
    return rows

cur.execute("SELECT * FROM student ORDER BY id")

page_size = 2    # Number of records per page
page_number = 1  # Start with page 1

while True:
    students = fetch_page(page_size)

    if not students:
        break # No more records, exit loop

    print(f"Page {page_number}:", students)
    page_number += 1
```

fetchmany နဲ့ နည်းလမ်းအပြင် SQL language ရဲ့ LIMIT နဲ့ OFFSET အသုံးပြုပြီး pagination လုပ်နိုင်ပါတယ်။

```
# File: db_pagination2.py
```

```
# Function to fetch a page of student records
def fetch_page(page_size, page_number):
    offset = (page_number - 1) * page_size

    # SQL query with LIMIT and OFFSET for pagination
    select_query = """
        SELECT * FROM student
        ORDER BY id
        LIMIT %s OFFSET %s
    """

    cur.execute(select_query, (page_size, offset))
    rows = cur.fetchall()

    return rows
```

OFFSET နဲ့ LIMIT ဟာ “page နံပါတ်” နဲ့ “တစ် page မှာ ပါရှိရမဲ့ record အရေအတွက်” ကို ကိုယ်စားပြုတာလို့ အကြမ်းဖျဉ်း ပြောလို့ရပါတယ်။ ဒီနည်းက record အရမ်းများပြီး page အရေအတွက် များတယ်၊ page တစ်ခုချင်း အစဉ်အတိုင်းမဟုတ်ဘဲ ဟိုကျော်ဒီကျော် ကြည့်လို့ရအောင် လုပ်ဖို့လိုတုံ့ အခြေအနေမျိုးမှာ ပိုပြီးအဆင်ပြေနိုင်ပါတယ်။ ပရိုဂရမ်ကုန် အပြည့်အစုံကို သက်ဆိုင်ရာ ဖိုင်မှာ ကြည့် နိုင်ပါတယ်။

rowcount Attribute

Insert, update, delete လုပ်တာဆိုရင်တော့ insert, update, delete လုပ်လိုက်တဲ့ record အရေအတွက်ကို rowcount attribute နဲ့ သိနိုင်ပါတယ်။

```
# File: db_update_and_rowcount.py
cur.execute("""
    UPDATE student
    SET grade = %s
    WHERE name = %s
""", ('A+', 'Amy'))
print(cur.rowcount)
```

၁.၈ Python နဲ့ SQL အကြား ဒေတာတိုက်ပ် ပြောင်းလဲခြင်း

ဒေတာဖော်ပြပုံ မတူကွဲပြားချက်တွေကြောင့် Python နဲ့ SQL language နှစ်ခုကြား ဒေတာအမျိုး အစား လိုက်လျောညီထွေဖြစ်အောင် ပြောင်းလဲမှုတွေ အပြန်အလှန် လုပ်ဆောင်ဖို့ လိုအပ်ပါတယ်။ SQL ပါရာမီတာနေရာမှာ Python ဒေတာတန်ဖိုး အစားထိုးတဲ့အခါ ဒေတာတိုက်ပ် ပြောင်းလဲခြင်းကို Psycopg က လုပ်ဆောင်ပေးပါတယ်။

```
# File: db_datatype_conv.py
# select account transactions between two dates
cur.execute("""
    SELECT * FROM account_transaction
```

```
WHERE txn_date BETWEEN %s AND %s
""", (datetime(2024, 8, 1), datetime(2024, 8, 31)))
rows = cur.fetchall()
```

Python datetime ကို သင့်တော်တဲ့ SQL ဒေတာအဖြစ် ပြောင်းလဲပေးဖို့ လိုမယ်ဆိုတာ မြင်နိုင်ပါတယ်။ ဒေတာဘေ့ခ်ဆီ ပေးပို့မဲ့ SQL ကို အခုလို mogrify ဖန်ရှင်နဲ့ ထုတ်ကြည့်လို့ ရတယ်

```
sql_str = cur.mogrify("""
    SELECT * FROM account_transaction
    WHERE txn_date BETWEEN %s AND %s
""", (datetime(2024, 8, 1), datetime(2024, 8, 31)))
print(sql_str.decode('utf-8'))
```

ဒီလို ပြောင်းပေးထားတာ တွေ့ရမှာပါ

```
SELECT * FROM account_transaction
    WHERE txn_date BETWEEN '2024-08-01T00:00:00'::timestamp
    AND '2024-08-31T00:00:00'::timestamp
```

ဒီတိုင်း စမ်းကြည့်ပါ

```
print(cur.mogrify("SELECT %s, %s, %s;", (None, True, False))
      .decode('utf-8'))
# Output: 'SELECT NULL, true, false;'
print(cur.mogrify("SELECT %s, %s, %s, %s;",
                  (10, 25 ** 25, 10.0, Decimal("10.00"))).decode('utf-8'))
# Output: 'SELECT 10, 88817841970012523233890533447265625, 10.0, 10.00;'
```

None ကို NULL ပြောင်းပေးပါတယ်။ bool, int, long, Decimal စတာတွေကိုလည်း သင့်တော်တဲ့ SQL တိုက်ပ်အဖြစ် ပြောင်းပေးတယ်။

Select လုပ်တဲ့ ရလဒ်ဒေတာတွေကိုလည်း သင့်တော်တဲ့ Python ဒေတာတိုက်ပ် ပြောင်းပေးမှာပါ။ Select လုပ်ရင် row တွေကို list တစ်ခုအနေနဲ့ ရတယ်။ အဲဒီ list ထဲမှာ row တစ်ခုကို tuple တစ်ခုနဲ့ ဖော်ပြတယ်။ Row ကို tuple အနေနဲ့ ဖော်ပြတဲ့အခါ column တန်ဖိုး တစ်ခုချင်းကို SQL ဒေတာတိုက်ပ် အပေါ်မူတည်ပြီး သက်ဆိုင်ရာ Python ဒေတာတိုက်ပ် ပြောင်းပေးပါတယ်။ Python နဲ့ SQL ဒေတာတိုက်ပ် တစ်ခုချင်းအတွက် လိုက်လျောညီထွေဖြစ်အောင် အပြန်အလှန်တွဲဖက်ပေးထားပုံကို Psycopg documentation စာမျက်နှာမှာ အသေးစိတ် ဖော်ပြထားပါတယ်

<https://www.psycopg.org/docs/usage.html#python-types-adaptation>

၁.၉ Dynamic SQL

ပရိုဂရမ်ကုဒ်ထဲမှာ ပုံသေရေးထားတာ မဟုတ်ဘဲ ပရိုဂရမ် run တဲ့ အချိန်ကျမှ လိုသလို တည်ဆောက်ယူတဲ့ SQL ကို Dynamic SQL လို့ ခေါ်ပါတယ်။ Table နံမည် ထည့်ပေးလိုက်၊ အဲ့ဒီ table ထဲက record တွေကို ထုတ်ပြရမယ်ဆိုပါစို့။ Select စတိတ်မန်မှာ table နံမည်က ပုံသေဖြစ်လို့ မရတော့ဘူး။ ပထမဆုံး စဉ်းစားမိတာက SQL ကို string နှစ်ခုဆက်ပြီး ထုတ်မယ်ပေါ့။

```
# Never do like this!!!
tbl_name = input("Enter table name: ")
select_tbl_sql = "SELECT * FROM " + tbl_name
```

ဒါဟာ လူပြိန်းနည်းနဲ့ dynamic SQL ထုတ်တဲ့ အရိုးရှင်းဆုံး ဥပမာတစ်ခုပဲ။ ဒီနည်းနဲ့ ရတယ်ဆိုပေမဲ့ ရှားရှားပါးပါး ကြုံရခဲတဲ့ ချင်းချက်အခြေအနေ တချို့ကလွဲလို့ string ဆက်ပြီး SQL ထုတ်တာက လုံးဝ ကို မလုပ်သင့်တဲ့ အရာပါ။ ဒါတင် မဟုတ်သေးဘူး။ f-strings, `str.format()`, template string, ပုံစံဟောင်း % အော်ပရိတ်တာ စတာတွေကိုလည်း dynamic SQL ထုတ်ဖို့ မသုံးသင့်ဘူး။ (Python မှာ *string interpolation* ပုံစံအမျိုးမျိုးရှိတယ်။ f-strings ကို စာမျက်နှာ ?? အခန်း ?? မှာ ဖော်ပြခဲ့ဖူးတယ်။ Python string interpolation အကြောင်း <https://tinyurl.com/y4jj285> မှာ ပြည့်ပြည့်စုံစုံ ရှင်းပြထားတာ တွေ့နိုင်တယ်။) ဘာကြောင့် မသုံးသင့်တာလဲ အကြောင်းအရင်းကို ဒီအခန်း နောက်ဆုံး ပိုင်းမှာ အားနည်းချက်/ပြဿနာတွေကို ဖော်ပြထားတာ ဖတ်ပြီးရင် နားလည်သွားပါလိမ့်မယ်။

Dynamic SQL အတွက် Psycopg မှာ sql မော်ဒူး ရှိပါတယ်။ String interpolation နဲ့ SQL ထုတ်ရင် ဖြစ်နိုင်တဲ့ ပြဿနာတွေကို ဖြေရှင်းပေးထားတယ်။ Table နံမည် dynamic SQL ကို အခုလို ရေးရမယ်။

```
# File: db_dynamic_sql1.py
# require sql module
from psycopg2 import sql

tbl_name = input("Enter table name: ")
query = sql.SQL("SELECT * FROM {f}").format(sql.Identifier(tbl_name))
cur.execute(query)
```

SQL string အစိတ်အပိုင်းတစ်ခုကို ကိုယ်စားပြုဖော်ပြဖို့အတွက် `sql.SQL` အောက်ဂျက် သုံးရပါတယ်။ ၎င်းအောက်ဂျက် ကိုယ်စားပြုတဲ့ SQL အစိတ်အပိုင်းထဲမှာ `{f}` ကို ဖြည့်ပေးရမဲ့ နေရာအဖြစ် ယာယီ သတ်မှတ်ပေးနိုင်တယ်။ အဲဒီနေရာကို ဖြည့်ပေးဖို့ `format` မက်သဒ်ကို သုံးရပါမယ်။ အပေါ်က `SELECT SQL` မှာ `{f}` ကို table နေရာမှာ ပထမ ထည့်ထားတယ်။ ပြီးတော့မှ `format` မက်သဒ်ခေါ်ပြီး အဲ့ဒီ နေရာမှာ `tbl_name` ဖြည့်တယ်။ `{f}` နေရာမှာ string တိုက်ရိုက် အစားမထိုးဘဲ `sql.Identifier` အောက်ဂျက် ပြောင်းပေးဖို့တော့ လိုတယ်။

`{f}` နဲ့ `%s` နှစ်ခုလုံးကို ဖြည့်ပေးရမဲ့ နေရာတွေအတွက် သုံးတယ် ဆိုပေမဲ့ `{f}` က table နံမည်၊ column နံမည် စတဲ့ identifier တွေအတွက်၊ `%s` ကိုက ဒေတာတန်ဖိုးတွေအတွက် သုံးတာ၊ ရည်ရွယ်ချက် ချင်းမတူပါဘူး။ အောက်ပါ ဥပမာကို ကြည့်ပါ။

```
# File: db_dynamic_sql2.py
table = 'student'
col1 = 'grade'
col2 = 'age'
query = sql.SQL("SELECT * FROM {f} WHERE {f} = %s AND {f} = %s").format(
    sql.Identifier(table), sql.Identifier(col1), sql.Identifier(col2))
# see the formatted result
print(query.as_string(conn))
cur.execute(query, ('A+', 20))
```

`format` မက်သဒ်နဲ့ table နဲ့ column နှစ်ခုနေရာကို `table`, `col1`, `col2` အစားထိုးပေးပါတယ်။

WHERE အပိုင်းမှာ column တစ်ခုချင်းအတွက် တန်ဖိုးကို %s ထည့်ထားတယ်။ query ကို execute လုပ်တဲ့အခါမှ % နေရာမှာ tuple ('A', 20) ထဲက တန်ဖိုးနဲ့ အစားထိုးမှာပါ။

Dynamic SQL အတွက် sql.SQL အောက်ဂျက်မှာ join မက်သစ်ကလည်း အသုံးဝင်တယ်။ နံမည်က join ဆိုတဲ့အတိုင်း column နံမည်တွေ ကော်မာနဲ့ဆက်တာ၊ ကွန်ဒီရှင်တွေ AND/OR နဲ့ ဆက်တာ စတဲ့ကိစ္စတွေအတွက် ကူညီပေးတဲ့ မက်သစ်ဖြစ်ပါတယ်။ အသုံးပြုပုံ လေ့လာကြည့်ပါ။

```
col_names = ['id', 'name', 'age', 'grade']
col_identifiers = []
for col in col_names:
    col_identifiers.append(sql.Identifier(col))
sql_frag1 = sql.SQL(", ").join(col_identifiers)
print(sql_frag1.as_string(conn)) # Output: "id", "name", "age", "grade"

# File: db_dynamic_sql2.py

# generating a full sql statement with WHERE conditions
conditions = []
for col in col_names:
    conditions.append(sql.SQL("{} = %s").format(sql.Identifier(col)))
sql_frag2 = sql.SQL(" AND ").join(conditions)
print(sql_frag2.as_string(conn))

# generate a full sql statement
sql_full = (sql.SQL("SELECT ")
            + sql.SQL(", ").join(col_identifiers)
            + sql.SQL(" FROM student WHERE ")
            + sql.SQL(" AND ").join(conditions))
print(sql_full.as_string(conn))
cur.execute(sql_full, (1, 'Amy', 20, 'A+'))
print(cur.fetchone())
```

sql.SQL အောက်ဂျက်တွေကို + အော်ပရေတာနဲ့ ဆက်ထားတာကိုလည်း သတိထားမိမှာပါ။

Psycopg ဒရိုက်ဗာ သုံးရင် အခုဖော်ပြခဲ့တဲ့ နည်းတွေနဲ့ပဲ dynamic SQL ထုတ်ရမယ်ကတော့ ဟုတ်ပါပြီ။ ဘာကြောင့် ဒီလောက် အလုပ်ရှုပ်ခဲပြီး dynamic SQL ထုတ်နေရတာလဲ၊ တစ်ခါတည်း ကုန်ထဲမှာ SQL string ကိုပဲ အပြည့်အစုံရေးထားလို့ မရဘူးလား စသည်ဖြင့် မေးခွန်းထုတ်စရာ ရှိပါတယ်။ တကယ့်လက်တွေ့ အပ်ပလီကေးရှင်းတွေမှာ လိုအပ်ချက်အများစုဟာ dynamic SQL သုံးရတာပါ။ ပုံသေရေးထားလို့ရတဲ့ static SQL သုံးရတာ နည်းတယ်။

Multi-field Search

ကျောင်းသား record တွေ search လုပ်တဲ့ ပရိုဂရမ် အစိတ်အပိုင်းတစ်ခုကို စိတ်ကူးကြည့်ပါ။ တစ်ခါတစ်ရံ ကျောင်းသားနံမည်နဲ့ ရှာတယ်။ တစ်ခါတစ်ရံမှာတော့ grade နဲ့ ရှာချင်ရှာမယ် (ဥပမာ ဘယ်သူတွေ grade A ရကြလဲ)။ ဒါမှမဟုတ် နံမည်နဲ့ grade နှစ်ခုလုံးနဲ့ ရှာတာလည်း ဖြစ်နိုင်တာပဲ။ Dynamic SQL မသုံးပဲ အခုလို စမ်းကြည့်နိုင်တယ်။


```
# File: multi_field_search1.py

print("Key in the value for each attribute."
      "Or press enter to ignore the attribute.")
name = input("Name: ")
grade = input("Grade: ")

sql0 = "SELECT * FROM student"
sql1 = "SELECT * FROM student WHERE name = %s"
sql2 = "SELECT * FROM student WHERE grade = %s"
sql3 = "SELECT * FROM student WHERE name = %s AND grade = %s"

if name.strip() and grade.strip():
    cur.execute(sql3, (name, grade.strip()))
elif name.strip():
    print(name.strip())
    cur.execute(sql1, (name.strip(),))
elif grade.strip():
    cur.execute(sql2, (grade.strip(),))
else:
    cur.execute(sql0)
```

နံမည်နဲ့ grade နှစ်ခုဆိုရင်တောင် if စတိတ်မန့်က အတော်လေး ရှုပ်နေပါပြီ။ SQL စတိတ်မန့် လေး ခုကိုလည်း ကြိုတင်သတ်မှတ်ထားရသေးတယ်။ Age နဲ့ ရှာလိုရအောင် ထပ်ဖြည့်မယ် ဆိုပါစို့။ အောက်ပါ အတိုင်း ဖြစ်နိုင်ခြေ ၈ ခု တွဲလိုရမှာပါ။

```
(name, age, grade),
(name, age), (name, grade), (grade, age),
(name), (age), (grade),
()
```

သုံးခုနဲ့တင် အတော်လေး ရှုပ်နေပါပြီ။ ပါရာမီတာတွေသာ ထပ်တိုးလာတာနဲ့အမျှ SQL တွေ၊ ဖြစ်နိုင်တဲ့ အတွဲတွေနဲ့ စစ်ရမဲ့ ကွန်ဒီရှင်တွေကလည်း ဖေါင်းပွလာမှာ။ ဒီလိုအခြေအနေမျိုးမှာ Dynamic SQL က အများကြီး အဆင်ပြေစေတယ်။ အောက်မှာ ရေးထားတဲ့ ပရိုဂရမ်ကုဒ်ကို လေ့လာကြည့်ပါ။

```
# File: multi_field_search2.py
# This program illustrates searching students by multiple columns

name = input("Name: ")
grade = input("Grade: ")
age = input("Age: ")

search_params = {
    'name': name.strip() if name.strip() else None,
    'age': int(age.strip()) if age.strip() else None,
    'grade': grade.strip() if grade.strip() else None
}
```

```

query = sql.SQL("SELECT * FROM student")

# List to hold the WHERE conditions
conditions = []
# List to hold the values
values = []

# Build conditions dynamically based on user input
for column, value in search_params.items():
    if value is not None:
        conditions.append(sql.SQL("{} = %s").format(sql.Identifier(column)))
        values.append(value)

# If there are any conditions, add them to the query
if conditions:
    query = query + sql.SQL(" WHERE ") + sql.SQL(" AND ").join(conditions)

```

နောက်ထပ် ပါရာမီတာတစ်ခု ထပ်ထည့်လည်း နည်းနည်းပဲ ပြင်ဖို့လိုတယ်။ ကျောင်းသား id နဲ့လည်း ရှာ လို့ရချင်ရင် input ဖတ်တာနဲ့ ပါရာမီတာတွေ ထည့်တဲ့ map မှာ ထပ်ထည့်ရုံပဲ။

```

# others inputs
id = input("ID: ")

search_params = {
    # others params
    'id': int(id.strip()) if id.strip() else None
}

```

အခုတွေ့ခဲ့တာက search ကို နမူနာအနေနဲ့ ပြထားတာပါ။ CRUD လုပ်တဲ့အခါ dynamic SQL ဟာ insert, select, update, delete လေးခုလုံးအတွက် သုံးရပါတယ်။ (Psycopg မဟုတ်တဲ့) အခြား ဒေတာဘေ့စ် လိုက်ဘရီတွေမှာလည်း dynamic SQL အတွက် သူနည်းသူဟန်နဲ့ ထောက်ပံ့ပေး ထားတာ တွေ့ရမှာဖြစ်ပြီး လိုက်ဘရီ မတူတဲ့အတွက် အသုံးပြုပုံလည်း ကွာခြားမယ်ဆိုပေမဲ့ အခုဖော်ပြခဲ့ တဲ့ အခြေခံ သဘောတရားတွေ နားလည်ရင် သိပ်အခက်အခဲမရှိဘဲ ဆက်လက်လေ့လာလို့ရမှာပါ။

၁.၁၀ Database Transactions

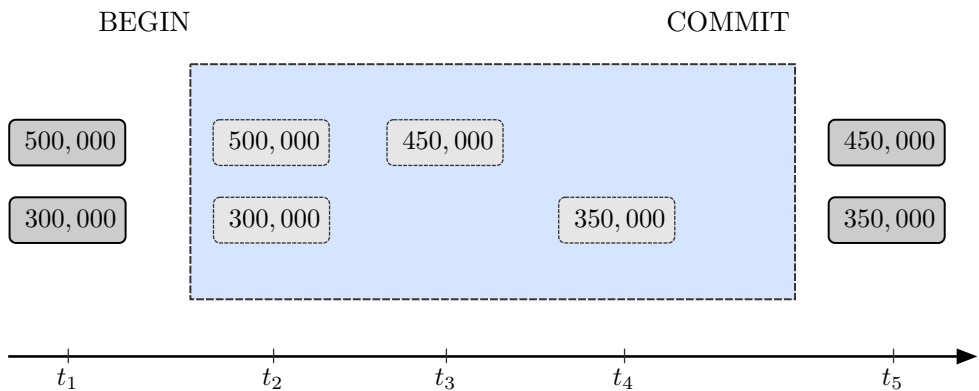
အောက်ဖော်ပြပါ လုပ်ငန်းကိစ္စတွေကို စဉ်းစားကြည့်ပါ။

- ဘဏ်အကောင့် ငွေလွှဲခြင်း
- ဟိုတယ် booking
- Online မှ ပစ္စည်းဝယ်ယူခြင်း

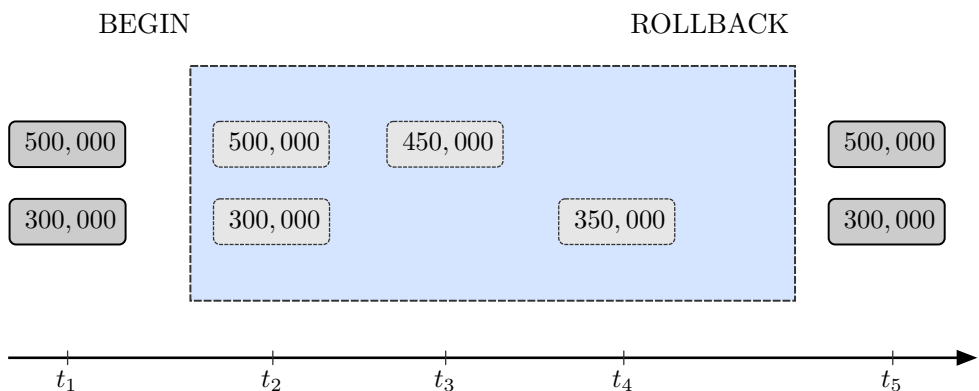
ဒီလို လုပ်ငန်းကိစ္စတစ်ခု ပြီးမြောက်ဖို့အတွက် ဆောင်ရွက်ပေးရတဲ့ လုပ်ငန်းစဉ်မှာ အဆင့်ဆင့် ပါဝင်တယ်။ ဘဏ်အကောင့် တစ်ခုနဲ့ တစ်ခု ငွေလွှဲတဲ့အခါ လွှဲသူ အကောင့်ကနေ ငွေကြေးပမာဏတစ်ခု နှုတ်ရမှာဖြစ် ပြီး လက်ခံအကောင့်ထဲကို ပေါင်းပေးရပါမယ်။ ဟိုတယ် booking လုပ်ရင်လည်း ကတ်စတမ်မာ တည်းမဲ့ ရက် အခန်းရနိုင်/မရနိုင် စစ်ကြည့်ပြီး အခန်း ဖယ်ထားပေးရပါမယ်။ အခန်းခ စရံငွေအတွက် ကတ်စတမ်

မာ အကောင့်ကနေ ပိုက်ဆံပေးချေရမယ်။ နောက်ဆုံးမှာ booking လုပ်ထားပြီးကြောင်း ကွန်ဖန်းလုပ်ဖို့ အီးမေးလ်ပို့ပေးရပါမယ်။ ထိုနည်းတူစွာ online ကနေ ပစ္စည်းဝယ်တာကို ခွဲခြမ်းစိတ်ဖြာ ကြည့်ရင်လည်း အဆင့် တစ်ခုမက ပါဝင်နေတာ တွေ့ရမှာပါ။

ဖော်ပြပါ ဥပမာတစ်ခုစီကို ကြည့်ရင် လုပ်ငန်းစဉ်တစ်ခုလုံး ပြီးမြောက်ဖို့အတွက် ၎င်းလုပ်ငန်းစဉ်မှာ ပါဝင်တဲ့အဆင့်အားလုံး ပြဿနာ တစ်စုံတစ်ရာ မရှိဘဲ အောင်မြင်ပြီးစီးအောင် ဆောင်ရွက်ရမှာပါ။ ငွေလွှဲ တဲ့အခါ ကိုယ့်ဆီကပဲ ပိုက်ဆံဖြတ်သွားပြီး တစ်ဖက်လူဆီ မရောက်မှာ စိုးရိမ်မိတာ ကျွန်တော်တို့အားလုံး ဖြစ်ဖူးမှာပါ။ ဟိုတယ် booking လုပ်တဲ့ ကိစ္စမှာဆိုရင် အခန်းကို ကတ်စတင်မာအတွက် ချန်ထားပြီးခါမှ အကြောင်းတစ်ခုခုကြောင့် စရံငွေဖြတ်လို့မရရင် ဟိုတယ်အနေနဲ့ ပြဿနာရှိပါတယ်။ အဲဒီအခန်းကို အခြား သူအတွက်လည်း ငှားလို့မရ ဖြစ်နေမှာပါ။



(က)



(ခ)

ပုံ ၁.၁၀ Database transaction သဘောတရား။ Dashed စတုရန်းအကြီးဟာ transaction စတင်၊ ထောင့်လုံး စတုရန်း အသေးတွေက အကောင့်နှစ်ခုရဲ့ လက်ကျန်ငွေ အချိန်နဲ့အမျှ ပြောင်းလဲနေပုံ၊ မြားဟာ အချိန် စီးဆင်းရာ။ (က) Transaction ကို commit လုပ်တဲ့အခါ ၎င်း transaction စတင် အတွင်း update လုပ်ထားတာတွေက အတည်ဖြစ်သွားပါတယ်။ (ခ) Rollback လုပ်ရင်တော့ transaction စတင်ထဲရှိ update တွေ ပျက်ပြယ်သွားပြီး လက်ကျန်ငွေ မပြောင်းလဲဘဲ နဂိုအတိုင်း ရှိနေမှာပါ။

အခုလို ပြဿနာမျိုးတွေ မဖြစ်အောင် ကာကွယ်ဖို့ ဒေတာဘေ့စ်တွေမှာ *transaction* သဘောတရား ကို ထည့်သွင်းတည်ဆောက် ပေးထားပါတယ်။ Transaction ဆိုတာ လုပ်ငန်းကိစ္စတစ်ခုအနေနဲ့ တစ် ပေါင်းတစ်စည်းတည်း ရှုမြင်တဲ့ ‘လုပ်ငန်းစဉ် အဆင့်ဆင့်’ လို့ ယူဆနိုင်တယ်။ အဲဒီ လုပ်ငန်းစဉ် အဆင့်ဆင့် ဟာ ဒေတာဘေ့စ် read/write/update အော်ပရေးရှင်းတွေ ပါဝင်နိုင်တယ်။ Transaction အတွင်း လုပ်ငန်းစဉ် အဆင့်အားလုံး အောင်မြင်ပြီးစီးသွားတာ၊ ဒါမှမဟုတ် ဘယ်တစ်ခုကိုမှ မလုပ်ဆောင်ဖြစ်တာ

နှစ်မျိုးပဲ ဖြစ်နိုင်ပါတယ်။ တချို့ပဲ လုပ်ဖြစ်သွားပြီး အကြောင်း တစ်ခုခုကြောင့် တချို့ကို မလုပ်ဖြစ်ဘဲ ကျန်ခဲ့တယ်ဆိုတာမျိုး လုံးဝမဖြစ်နိုင်ဘူး (ဥပမာ transaction အတွင်းမှာ ငွေလွှဲတဲ့ ကိစ္စလုပ်ရင် အကောင့်တစ်ခုကနေပဲ ပိုက်ဆံဖြတ်သွားပြီး တစ်ဖက်အကောင့်မှာ မဝင်သွားတာ မဖြစ်နိုင်တော့ဘူး)။ Transaction တစ်ခု အတွင်းက အချက်အလက် အပြောင်းအလဲ မှန်သမျှဟာ ၎င်း transaction ကို commit မလုပ်မချင်း အတည်မဖြစ်သေး၊ ယာယီသာဖြစ်တယ်။ ၎င်းတို့ကို အတည်ဖြစ်စေချင်ရင် commit လုပ်ရပါမယ်၊ အကယ်၍ ပယ်ဖျက်ပြီး အားလုံးနိမ့်အတိုင်း ပြန်ရှိစေချင်ရင် rollback လုပ်နိုင်ပါတယ်။ ပုံ (၁.၁၀) မှာ commit နဲ့ rollback အလုပ်လုပ်ပုံ သဘောတရားကို တွေ့ရပါမယ်။ အချက်အလက် မှန်ကန်စိတ်ချရခြင်း (data integrity) အတွက် အာမခံချက် အပြည့်အဝ ပေးနိုင်ပြီး တစ်ဝက်တစ်ပျက် လုပ်ဆောင်မှုတွေကြောင့် ဒေတာဘေ့စ် မှန်ကန်ကိုက်ညီမှု မရှိခြင်း အခြေအနေကို ကာကွယ်ဖို့ရာအတွက် transaction သဘောတရားဟာ အလွန်အရေးပါတယ်။

Transaction နဲ့ ပတ်သက်ပြီး နောက်ထပ်အရေးပါတဲ့ အချက်တစ်ခုက isolation သဘောတရားပါ။ Transaction တစ်ခုအတွင်း လုပ်ဆောင်ထားတဲ့ ဒေတာအပြောင်းအလဲတွေဟာ (update, insert, delete ကို ဆိုလိုတာ) အဲဒီ transaction မပြီးမချင်း ဘယ်သူကမှ မမြင်ရဘူး။ တစ်နည်းအားဖြင့် transaction တစ်ခု commit မလုပ်ရသေးခင်မှာ ၎င်းအတွင်းရှိ ကြားကာလ အခြေအနေကို အခြား transaction တွေကနေ တွေ့မြင်ရမှာ မဟုတ်ပါဘူး (Isolation level ပေါ်မှာတော့ မူတည်တယ်)။ Isolation ဟာ ဒေတာမှန်ကန်ကိုက်ညီခြင်း အတွက် အရေးကြီးပါတယ်။ Transaction မပြီးစီးခင် ကြားကာလဟာ မှန်ကန်ကိုက်ညီခြင်း မရှိသေးတဲ့ အခြေအနေမှာ ရှိနေနိုင်ပါတယ် (ပုံ ၁.၁၀ ရဲ့ t_3 နဲ့ t_4 ကြားကာလကို စဉ်းစားကြည့်ပါ)။ အဲဒီလို အနေအထားကို ပြင်ပကနေ မမြင်နိုင်အောင် isolation နဲ့ ကာကွယ်ပေးထားတာပါ။

ဒေတာဘေ့စ်တွေမှာ isolation level (၄) မျိုး ရှိပါတယ်။ Isolation အလျော့အတင်း အလိုက် level အခုလို စဉ်ထားပါတယ်

- Read Uncommitted
- Read Committed (default)
- Repeatable Read
- Serializable

ဒေတာဘေ့စ် အများစုမှာ Read Committed က default ပါ။ Transaction တစ်ခု commit မလုပ်ရသေးတာတွေကို အခြားကနေ မြင်ရနိုင်တဲ့ Read Uncommitted ကိုတော့ အသုံးပြုတာ မတွေ့ရသလောက်ပါပဲ။ Isolation level မြင့်လေ concurrency အားနည်းလေလို့ ယေဘုယျ ပြောနိုင်ပါတယ်။

Transaction အကြောင်း အတန်အသင့်နားလည်ပြီး SQL နဲ့ Psycopg မှာ ဘယ်လို ဖန်တီးအသုံးပြုရလဲ ကြည့်ရအောင်။ BEGIN, COMMIT, ROLLBACK တို့ဟာ transaction နဲ့ဆိုင်တဲ့ SQL ကွန်မန်းတွေ။ BEGIN က transaction အစပြုပေးတဲ့ ကွန်မန်းဖြစ်တယ်။ COMMIT လုပ်ရင် transaction တစ်ခုလုံးကို (ပါဝင်တဲ့ အဆင့်အားလုံးကိုဆိုလို) အတည်ပြုပေးမှာဖြစ်ပြီး ROLLBACK လုပ်ရင် transaction တစ်ခုလုံးကို ပြန်လည်ရုတ်သိမ်းပေးမှာပါ။ COMMIT (သို့) ROLLBACK လုပ်တဲ့အခါ transaction လည်း ပြီးဆုံးပါတယ်။ နောက်ထပ် အသစ်တစ်ခုလိုအပ်ရင် BEGIN နဲ့ ပြန်စရပါမယ်။

psql မှာ စမ်းကြည့်မယ်ဆိုရင် AUTOCOMMIT ကို off ပေးရပါမယ် (default က on ထားတယ်)။ \set ကွန်မန်းနဲ့ အခုလို run ထားပါ။

```
\set AUTOCOMMIT off
```

အခုလို ပြန်စစ်ကြည့်ပါ။ Off ဖြစ်နေသင့်ပါတယ် (ပုံ ၁.၁၁ မှာ ကြည့်ပါ)။

```
\echo :AUTOCOMMIT
```

ဆက်လက်ပြီး အောက်ပါအတိုင်း စမ်းကြည့်ပါ။ BEGIN ကွန်မန်းနဲ့ transaction ကို စတင်ပါတယ်။ Transaction ထဲမှာ UPDATE နှစ်ခု run ပြီး COMMIT လုပ်တယ်။ UPDATE နှစ်ခုလုံး run ပြီးတော့ မှ COMMIT မလုပ်ဘဲ ROLLBACK လုပ်ရင် အကောင့်နှစ်ခုလုံး transaction မစခင် မူလအနေအထား အတိုင်း ရှိနေမှာပါ။

Isolation နဲ့ COMMIT/ROLLBACK အလုပ်လုပ်ပုံ သဘောတရားကို psql နောက်ထပ်တစ်ခု ဖွင့်ပြီး စစ်ဆေး စမ်းသပ် ကြည့်နိုင်ပါတယ်။ ပထမ psql မှာ SQL တစ်ခု run ပြီးတိုင်း နောက် psql တစ်ခုကနေ SELECT လုပ်ပြီး စောင့်ကြည့်လေ့လာပါ။ နမူနာ စမ်းကြည့်ထားတာကို ပုံ (၁.၁၂) မှာ တွေ့နိုင်တယ်။ ပထမ SELECT ရလဒ်တွေမှာ ဒေတာအပြောင်းအလဲ ဖြစ်တာ မတွေ့ရသေးပါဘူး။ နောက်ဆုံး SELECT က ပထမ psql မှာ COMMIT လုပ်ပြီးမှ run ထားတာပါ။ အဲဒီ ရလဒ်မှာ အကောင့် record နှစ်ခု အမှန်တကယ် update ဖြစ်သွားတာကို တွေ့ရမှာပါ။

```
/* Transaction in SQL */
BEGIN;
UPDATE account SET balance = balance - 50000.00
WHERE acc_no = '0086-6002-1111';
UPDATE account SET balance = balance + 50000.00
WHERE acc_no = '0086-6002-3311';
-- etc etc
COMMIT;
```

```
Select SQL Shell (psql)
bank=# \set AUTOCOMMIT off
bank=# \echo :AUTOCOMMIT
off
bank=# UPDATE account SET balance = balance - 50000.00 WHERE acc_no = '0086-6002-1111';
UPDATE 1
bank=# UPDATE account SET balance = balance + 50000.00 WHERE acc_no = '0086-6002-3311';
UPDATE 1
bank=# COMMIT;
COMMIT
bank=#
```

ပုံ ၁.၁၁ psql မှတစ်ဆင့် transaction တစ်ခု ဖန်တီးအသုံးပြုပုံ

Psycopg ဒရိုက်ဗာက သူ့ကိုအတိုင်းဆိုရင် AUTOCOMMIT ပိတ်ပြီးသားပါ။ ပထမဆုံး execute လုပ်တဲ့အခါ transaction ကို အလိုအလျောက် စပေးပါတယ် (execute လုပ်ပြီးဆိုတာနဲ့ BEGIN ကို အရင်လုပ်ပေးမှာပါ)။ COMMIT လုပ်ရင် conn.commit(), ROLLBACK ဆိုရင် conn.rollback() ခေါ်ပေးရပါမယ်။ commit() မလုပ်မီဘဲ ကွန်နက်ရှင် ပိတ်လိုက်ရင် transaction အတွင်း လုပ်ထား သမျှ ဒေတာအပြောင်းအလဲ အားလုံး အတည်မဖြစ်တော့ဘဲ အားလုံး ပျက်ပြယ်သွားပါမယ်။

Psycopg နဲ့ transaction စီမံတဲ့ နမူနာပုံစံကို အောက်မှာကြည့်ပါ။ Transaction စီမံတဲ့နေရာ မှာ exception handling က အရေးပါပါတယ်။ try ထဲမှာ transaction မှာ ပါဝင်တဲ့ အဆင့်တွေ ကို လုပ်ဆောင်ရလေ့ရှိတယ်။ အခု ဥပမာမှာ update နှစ်ခု လုပ်ထားတယ်။ နှစ်ခုလုံး ပြဿနာမရှိဘဲ ပြီး ရင် commit လုပ်သွားမယ်။ အကြောင်းတစ်ခုခုကြောင့် exception တက်ခဲ့ရင် except ဘလောက်ထဲ

```
Select SQL Shell (psql)
bank=# SELECT * FROM account;
 acc_id | holder_id | acc_no | acc_type | balance
-----+-----+-----+-----+-----
      1 |          1 | 0086-6002-1111 | Savings | 500000.00
      2 |          1 | 0088-6005-1122 | Current | 800000.00
      3 |          2 | 0086-6002-3311 | Savings | 400000.00
      4 |          3 | 0086-6002-4411 | Savings | 700000.00
(4 rows)

bank=# SELECT * FROM account;
 acc_id | holder_id | acc_no | acc_type | balance
-----+-----+-----+-----+-----
      1 |          1 | 0086-6002-1111 | Savings | 500000.00
      2 |          1 | 0088-6005-1122 | Current | 800000.00
      3 |          2 | 0086-6002-3311 | Savings | 400000.00
      4 |          3 | 0086-6002-4411 | Savings | 700000.00
(4 rows)

bank=# SELECT * FROM account;
 acc_id | holder_id | acc_no | acc_type | balance
-----+-----+-----+-----+-----
      2 |          1 | 0088-6005-1122 | Current | 800000.00
      4 |          3 | 0086-6002-4411 | Savings | 700000.00
      1 |          1 | 0086-6002-1111 | Savings | 450000.00
      3 |          2 | 0086-6002-3311 | Savings | 450000.00
(4 rows)
```

ပုံ ၁.၁၂ အခြား transaction တစ်ခုကို psql မှ စောင့်ကြည့်ပုံ

ရောက်ပြီး rollback ဖြစ်မှာပါ။

```
# File: db_transaction_eg1.py
# Transaction in Python with Psycopg
import psycopg2

conn = psycopg2.connect(dbname="bank", user="postgres",
                        password="asdfgh", host="localhost", port="5432")
cur = conn.cursor()

try:
    cur.execute("UPDATE account SET balance = balance - 50000.00 "
                "WHERE acc_no = '0086-6002-1111'")
    cur.execute("UPDATE account SET balance = balance + 50000.00 "
                "WHERE acc_no = '0086-6002-3311'")
    conn.commit()
except psycopg2.Error as e:
    conn.rollback()
    print("Database error: ", e)
except Exception as e:
    conn.rollback()
    print("Unknown error: ", e)
finally:
    cur.close()
```

```
conn.close()
```

except ဘလောက်တွေမှာ rollback လုပ်ဖြစ်အောင် လုပ်ဖို့ သေချာရရှိစိုက်ရပါမယ်။ မေ့ကျန်ခဲ့တာ ဖြစ်နိုင်တယ်။ ဒီလို မဖြစ်အောင် ကူညီပေးတဲ့ with စတိတ်မန်ကို ပိုအသုံးများတယ်။ Connection နဲ့ cursor ကို with နဲ့တွဲသုံးရင် commit, rollback နဲ့ cursor ပိတ် ကိစ္စတွေကို အလိုအလျောက် လုပ် ပေးမှာမို့လို့ မေ့ကျန်ခဲ့စရာ အကြောင်း သိပ်မရှိတော့ဘူး။ Connection ကိုပဲ သေချာရရှိစိုက် ပိတ်ပေးဖို့ လိုတယ်။

```
# File: db_transaction_eg2.py
```

```
# Using with statement
```

```
conn = None
```

```
try:
```

```
    conn = psycopg2.connect(dbname="bank", user="postgresql",
                           password="asdfgh", host="localhost", port="5432")
```

```
    with conn:
```

```
        with conn.cursor() as cur:
```

```
            cur.execute("UPDATE account SET balance = balance - 50000.00 "
                        "WHERE acc_no = '0086-6002-1111'")
```

```
            cur.execute("UPDATE account SET balance = balance + 50000.00 "
                        "WHERE acc_no = '0086-6002-3311'")
```

```
except psycopg2.Error as e:
```

```
    print("Database error: ", e)
```

```
except Exception as e:
```

```
    print("Unknown error: ", e)
```

```
finally:
```

```
    conn.close()
```

Psycopg ဗားရှင်း 2.5 ကစပြီး connection, cursor တို့ကို with စတိတ်မန်နဲ့ တွဲဖက် အသုံးပြု နိုင်ပါတယ်

```
conn = psycopg2.connect()
```

```
with conn:
```

```
    with conn.cursor() as cur:
```

```
        cur.execute(SQL1)
```

```
conn.close()
```

Connection နဲ့ဆိုင်တဲ့ with ဘလောက် (outer with) ပြီးဆုံးသွားတယ်၊ exception မဖြစ် ဘူးဆိုရင် transaction ကို commit လုပ်ပေးမှာဖြစ်ပြီး exception ဖြစ်ခဲ့ရင်တော့ rollback ဖြစ်သွားမှာပါ။ Cursor နဲ့ သက်ဆိုင်တဲ့ with ဘလောက် (inner with) ပြီးဆုံးရင် cursor ကို အလိုအလျောက် ပိတ်ပေးမှာဖြစ်ပေမဲ့ transaction က ဆက်ရှိနေအုံးမှာပါ။ Transaction com- mit/rollback က connection နဲ့ဆိုင်တဲ့ with ဘလောက် အဆုံးမှာ ဖြစ်တယ်။ with က connection ကို အလိုအလျောက် မပိတ်ပေးပါဘူး။ ဒါကြောင့် ကိုယ်တိုင် ပိတ်ပေးရပါမယ်။

၁.၁၁ Concurrency

ဒေတာဘေ့စ်တွေဟာ တစ်ချိန်တည်း user အများအပြား တစ်ပြိုင်နက် အသုံးပြုရင် ဖြစ်ပေါ်နိုင်တဲ့ concurrency problem တွေ မဖြစ်ပေါ်အောင် ကာကွယ်ဖို့အတွက် နည်းလမ်းတွေ ထောက်ပံ့ပေးထားပါတယ်။ *Concurrency* ဆိုတာ တစ်ခုထက်ပိုတဲ့ အလုပ်တွေ ကွန်ပျူတာက တစ်ချိန်တည်းမှာ လုပ်ဆောင်ပေးနေတာကို ဆိုလိုတာပါ။ တစ်ချိန်တည်း လုပ်ဆောင်ပေးတယ် ဆိုပေမဲ့ အားလုံး တစ်ပြိုင်နက်တည်း လုပ်ဆောင်တာ ဟုတ်ချင်မှ ဟုတ်မှာပါ။ အလုပ်တစ်ခုစီကို အလှည့်ကျ လုပ်ဆောင်ပေးပြီး တစ်ပြိုင်နက်ထဲ ဖြစ်ပျက်နေတယ်လို့ ထင်ရအောင် စီမံထားတာလည်း ဖြစ်နိုင်ပါတယ်။

ကလပ်စ် concurrency problem ဥပမာတစ်ခုကို ဒီအပိုင်းမှာ လေ့လာကြည့်ပါမယ်။ အောက်ဖော်ပြပါ ပရိုဂရမ်ကုဒ် အစိတ်အပိုင်းဟာ ငွေထုတ်ယူတဲ့ ကိစ္စအတွက် ဖြစ်ပါတယ်။

```
# Step 1: select account balance from database
cur.execute("""
    SELECT balance FROM account WHERE acc_no = %s
""", (acc_no,))
account_balance = cur.fetchone()

# Step 2: check if the balance is enough
if not account_balance:
    raise Exception("Account not found.")
elif account_balance[0] < amount:
    raise Exception("Insufficient funds in the account.")

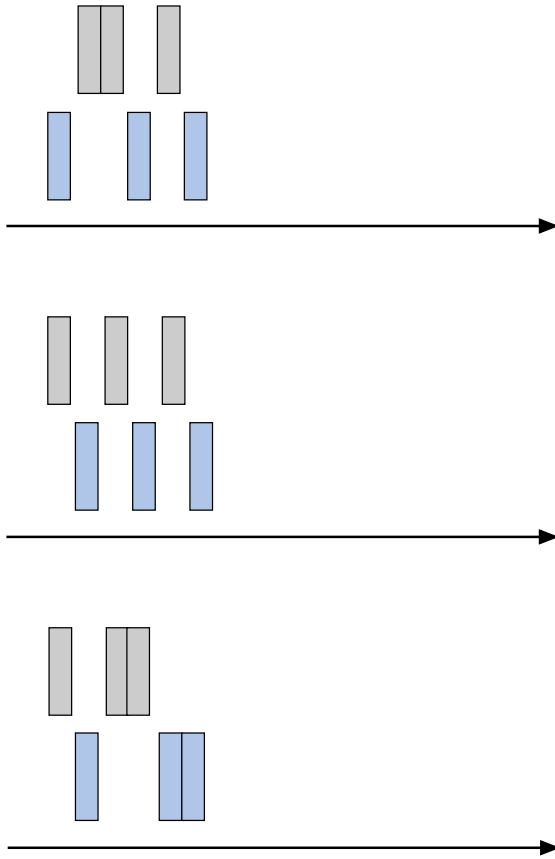
# Step 3: Debit the account
cur.execute("""
    UPDATE account
    SET balance = balance - %s
    WHERE acc_no = %s
""", (amount, acc_no))
conn.commit()
```

ငွေထုတ်တဲ့ကိစ္စမှာ အကြမ်းဖျဉ်း step သုံးခု ပါဝင်တာ တွေ့ရမှာပါ

- လက်ကျန်ငွေ select လုပ်ခြင်း
- လက်ကျန်ငွေ လုံလောက်မှု ရှိ/မရှိ စစ်ဆေးခြင်း
- လက်ကျန်ငွေ update လုပ်ခြင်း

(ဒါထက် အသေးစိတ် ထပ်ခွဲလို့ ရအုံးမှာပါ။ ရှေ့ဆက်ရှင်းပြမဲ့ ကိစ္စအတွက် ဒီလောက်နဲ့က နားလည်ရ ပိုလွယ်ပါမယ်။)

Concurrent ပရိုဂရမ်တစ်ခုဟာ ငွေထုတ်သူ တစ်ဦးချင်းစီအတွက် ဒီ အစဉ်လိုက် step သုံးခုကို လုပ်ဆောင်ပေးရမှာပါ။ တစ်ချိန်တည်း နှစ်ယောက်ထုတ်ရင် step တစ်ခုစီကို တစ်ယောက် တစ်လှည့် နှစ်ယောက်လုံးအတွက် တစ်ချိန်တည်းမှာ ဆောင်ရွက်ပေးရပါမယ်။ အနီးစပ်ဆုံး မြင်သာအောင် ပြောမယ်ဆိုရင် အဖျော်ဆရာတစ်ယောက် လက်ဖက်ရည်ဖျော်သလိုပဲ၊ တစ်ခွက်ပြီးမှ တစ်ခွက်ဖျော်တာ မဟုတ်ဘူး။ လက်ဖက်ရည်ခွက်တွေ ရှေ့မှာစီချထားပြီး အကျရည်ထည့်၊ နို့ဆီထည့်၊ နို့စိမ်းထည့် အားလုံး တစ်ပြိုင်တည်း နီးပါး အပြီးဖျော်တာ။ အလုပ်တွေကို တစ်ချိန်တည်း လုပ်တယ်ဆိုတာ ဒီသဘောကို ဆိုလိုတာ။



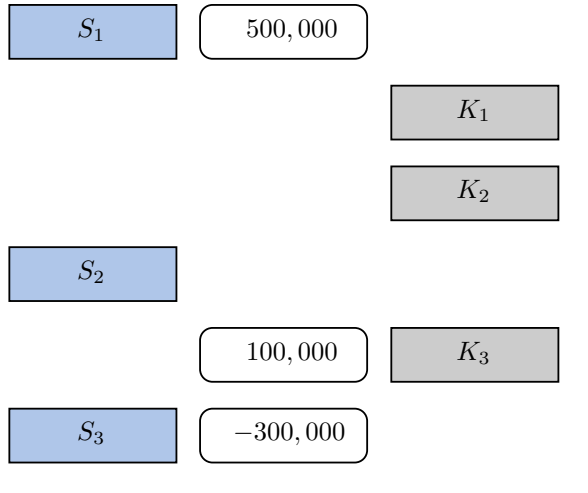
ပုံ ၁.၁၃ ငွေထုတ်ကိစ္စ နှစ်ခု တစ်ချိန်တည်း အလှည့်ကျ လုပ်ဆောင်ပုံ ဖြစ်နိုင်ခြေ ၃ ခု (အားလုံး မဟုတ်ပါ)။ concurrency သဘာဝအရ ဘယ်လို အလှည့်ကျမလဲ ဆိုတာက random ဝဲ၊ ပုံသေမရှိဘူး။ ပရိုဂရမ်မာက အလှည့်ကျ အစီအစဉ်ကို လိုသလို ထိန်းကွပ်လို့ မရနိုင်ဘူး။

အဖျော်ဆရာ အလုပ်လုပ်ပုံနဲ့ concurrent ပရိုဂရမ် လုံးဝမတူတဲ့ အချက်တစ်ခုရှိတယ်။ Concurrency သဘာဝအရ အလုပ်တစ်ခုစီကို အချိန်အနည်းငယ်ကြာ အလှည့်ကျ လုပ်ဆောင်ပေးပါတယ်။ ဒီလို အလှည့်ပေးစနစ်ကို operating system မှာ ပါဝင်တဲ့ scheduler က စီမံတာဖြစ်ပြီး ပရိုဂရမ်ရေးသားသူ လိုသလို စိတ်ကြိုက် ထိန်းကွပ်လို့ မရနိုင်ပါဘူး။ ဒီအတွက်ကြောင့် အလုပ်နှစ်ခုမှာပါဝင်တဲ့ step တွေ ဘယ်လိုအစဉ်နဲ့ အလှည့်ကျ လုပ်ဆောင်မလဲ ပုံသေတွက်လို့မရတော့ဘူး။

ပုံ (၁.၁၃) မှာ အလုပ်နှစ်ခု အလှည့်ကျ လုပ်ဆောင်ပုံ ဖြစ်နိုင်ခြေ သုံးခုကို တွေ့ရပါမယ်။ အလုပ်နှစ်ခုကို အထက်အောက် အရောင်ခွဲ ပြထားတယ်။ စတုဂံငယ် တစ်ခုစီက ပါဝင်တဲ့ step တစ်ခုချင်းကို ကိုယ်စားပြုတာ၊ မြားဟာ အချိန် စီးဆင်းရာ။ ပုံမှာပြထားတာ သုံးခုအပြင် အခြား ဖြစ်နိုင်တဲ့ အစီအစဉ်တွေ ကျန်ပါသေးတယ်။ သင်္ချာနည်းနည်းကျွမ်းတယ်၊ စိတ်ဝင်စားတယ်ဆိုရင် ဖြစ်နိုင်ခြေ အားလုံး တွက်ကြည့်လို့ရပါတယ် (*Permutations with repetition or combinatorics* သဘောတရားနဲ့ တွက်ရမှာပါ။ အခုကိစ္စအတွက် ဖြစ်နိုင်ခြေ အားလုံး အခု ၂၀ ရှိပါမယ်)။

Concurrent အလုပ်တွေကြားမှာ အချက်အလက် မျှဝေသုံးစွဲတာ မရှိရင် ထူးထူးခြားခြား ပြဿနာ မရှိပါဘူး။ ပရိုဂရမ် ရေးသားရတာလည်း ပုံမှန်ထက် အများကြီး မခက်ဘူး။ ဒါပေမဲ့ အချက်အလက် မျှဝေသုံးစွဲတာ ရှိခဲ့ရင်တော့ ပြဿနာရှိလာပါတယ်။ ဥပမာ အကောင့်တစ်ခုတည်းကနေ တစ်ပြိုင်တည်း ငွေထုတ်တဲ့ ဖြစ်စဉ်ကို စဉ်းစားကြည့်ပါ။ စန္ဒီနဲ့ ကေသီ နှစ်ယောက်ပေါင်း အကောင့်တစ်ခု ဖွင့်ထားတယ်။ လက်ရှိ

အကောင့် လက်ကျန်ငွေ ၅ သိန်း ရှိပြီး သူတို့ နှစ်ယောက် တစ်နေရာစီကနေ ၄ သိန်း သီးခြား ထုတ်ယူကြတာ တိုက်တိုက်ဆိုင်ဆိုင် တစ်ချိန်တည်းဖြစ်သွားတယ်လို့ စိတ်ကူးကြည့်ပါ။ ဒီဖြစ်စဉ်မှာ အကောင့် record ဟာ shared data ဖြစ်ပြီး အလုပ်နှစ်ခုက တူညီတဲ့ record တစ်ခုတည်းကို တစ်ချိန်တည်းမှာ update လုပ်ဖို့အတွက် ကြိုးစားကြတာကို တွေ့ရမှာပါ။



ပုံ ၁.၁၄ Concurrency problem ဥပမာ။ ၅ သိန်းရှိတဲ့ အကောင့် တစ်ခုတည်းကနေ တစ်ယောက် ၄ သိန်း၊ နှစ်ယောက် တစ်ချိန်တည်း ငွေထုတ်တဲ့အခါ step တစ်ခုချင်းအလိုက် လက်ကျန်ငွေ ပြောင်းလဲပုံ

ပုံ (၁.၁၃) အပေါ်ဆုံးကအတိုင်း အလှည့်ကျ လုပ်ဆောင်တယ် ဆိုပါစို့။ Step တစ်ခုချင်းအလိုက် လက်ကျန်ငွေ balance ပြောင်းလဲပုံကို ပုံ (၁.၁၄) မှာ ပြထားတယ်။ $K_1, K_2, K_3, S_1, S_2, S_3$ တို့ဟာ ကေသီနဲ့ စန္ဒီအတွက် step သုံးခုစီလို့ ယူဆပါ (K for Kathy, S for Sandy)။ S_1, K_1, K_2, S_2 လုပ်ဆောင်ပြီးချိန်အထိ လက်ကျန်ငွေ ၅ သိန်းဟာ နဂိုအတိုင်း မပြောင်းလဲသေးဘူး။

K_3 အပြီးမှာ လက်ကျန်ငွေ balance ဟာ ၁ သိန်း ဖြစ်သွားပြီ။ ဒီအတိုင်းဆိုရင် နောက်ထပ် ၄ သိန်း ထုတ်လို့ မရသင့်တော့ဘူး။ S_2 မှာ လက်ကျန်ငွေ လောက်/မလောက် စစ်ခဲ့ချိန်က ဒီလိုမဟုတ်သေးဘူး။ အဲ့တုန်းက ၅ သိန်းရှိခဲ့တာ။ ဆိုတော့ S_2 အရဆိုရင် S_3 ကို ဆက်လက်လုပ်ဆောင်ရမှာပဲ။ S_3 ပြီးသွားတဲ့အခါ balance ဟာ အနှုတ် ၃ သိန်းဖြစ်သွားပါတယ်။ ကေသီနဲ့ အေမီဟာ သူတို့မှာ ရှိတဲ့ငွေထက် ဘဏ်ကနေ ၃ သိန်း အပိုထုတ်လို့ရသွားတာ ဖြစ်ပါတယ်။ ပုံ (၁.၁၃) က နောက်ထပ် ဖြစ်နိုင်ခြေ နှစ်ခုမှာလည်း ဒီပြဿနာ တွေ့ရမှာပါ။

ဘုကြောင့် ဒီလို ဖြစ်ရတာလဲ၊ မဖြစ်အောင် ဘယ်လို ကာကွယ်ရမလဲ။ တစ်ချိန်တည်းမှာ ထုတ်ယူကြပေမဲ့ အကောင့်တစ်ခုတည်းကနေ မဟုတ်ရင် ဒီလိုပြဿနာ မဖြစ်နိုင်ဘူး။ တစ်နည်းအားဖြင့် မတူညီတဲ့ သီးခြားအကောင့်တစ်ခုစီကနေ တစ်ပြိုင်နက် ငွေထုတ်ယူတာဟာ concurrency problem မဖြစ်စေနိုင်ဘူး။ တစ်ချိန်တည်း၊ အကောင့်တစ်ခုတည်းကနေ ငွေထုတ်တဲ့ကိစ္စနှစ်ခု တိုက်ဆိုင်တဲ့အခါ concurrency problem ရှိနိုင်တာပါ။ ဖြေရှင်းဖို့ နည်းလမ်းကတော့ အကောင့်တစ်ခုကို တစ်ချိန်တည်းမှာ အလုပ်တစ်ခုကပဲ update လုပ်လို့ရအောင် ကာကွယ်ပေးထားရပါမယ်။ Transaction တစ်ခုဟာ ၎င်း update လုပ်ဖို့ ရည်ရွယ်တဲ့ record ကို အခြား transaction တွေကနေ update လုပ်လို့မရအောင် FOR UPDATE နဲ့ တားဆီးနိုင်ပါတယ်။ SELECT လုပ်တဲ့အချိန်မှာ အခုလို တွဲသုံးရမှာပါ

```
# File: db_transaction_and_concurrency.py
```

```
# Step 1: select account balance from database
```

```
cur.execute("""
    SELECT balance FROM account WHERE acc_no = %s FOR UPDATE
""", (acc_no,))
account_balance = cur.fetchone()
```

Concurrency စကားနဲ့ ပြောရင် FOR UPDATE ဟာ select လုပ်လိုက်တဲ့ row တွေအပေါ်မှာ exclusive lock ရယူတာဖြစ်တယ်။ ဒါနဲ့ပါတ်သက်တဲ့ အသေးစိတ်ကို concurrency အခန်းမှာ သီးခြား ရှင်းပြမှာပါ။

Concurrency ဟာ ကျယ်ပြန့်ပြီး သီးခြားအထူးပြု လေ့လာရမဲ့ အပိုင်းဖြစ်ပါတယ်။ စာမျက်နှာ ၄၅ အခန်း (၂) မှာ အခြေခံ concurrency အကြောင်း ဖော်ပြပေးထားတယ်။ အခု တွေ့ခဲ့ရတဲ့ ဥပမာက ဒေတာဘေ့စ် အပ်ပလီကေးရှင်းတွေမှာ ကြုံတွေ့ရနိုင်တဲ့ concurrency problem တွေ အများကြီးထဲက မှ အခြေခံ တစ်ခုလေးပဲ ရွေးထုတ်ထားတာ။ ဒေတာဘေ့စ် concurrency နဲ့ ပါတ်သက်ပြီး ပေးထားတဲ့ ကိုးကားစာအုပ်တွေ၊ ဒါမှမဟုတ် အခြား တစ်နေရာကနေ ဖြည့်စွက်လေ့လာဖို့ တိုက်တွန်းပါတယ်။

၁.၁၂ SQL Injection and Dynamic SQL

String interpolation နည်းလမ်းနဲ့ Dynamic SQL မထုတ်သင့်ဘူး ပြောခဲ့ပေမဲ့ ဘာကြောင့်လဲ အကြောင်းအရင်းကို မရှင်းပြခဲ့ဘူး။ ဒါနဲ့ ပါတ်သက်ပြီး အကြောင်းအချက် တချို့ကို ဆက်လက် လေ့လာ ကြပါမယ်။ အသိသာဆုံး ပြဿနာတစ်ခုက SQL နဲ့ Python တို့ဟာ string ကို ဖော်ပြပုံ မတူတာပါ။ နောက်ဆုံးအမည် 0'Brian နဲ့ အကောင့်ပိုင်ရှင်ကို SQL မှာ အခုလို select လုပ်ရပါတယ်။

```
SELECT * FROM account_holder WHERE lname = '0'Brian';
```

စာသားကို single quote နှစ်ခုအတွင်းမှာ ရေးတယ်။ စာသားထဲမှာ ' ပါနေရင် '' (single quote နှစ်ခု) နဲ့ escape လုပ်ပေးရပါမယ်။

SQL identifier တွေမှာ စပေ့စ်၊ ဟိုက်ဖန် (သို့) အခြား ထူးခြားသင်္ကေတတွေ ပါဝင်နေတဲ့အခါ double quotes (") နှစ်ခုအတွင်း ထည့်ရေးလေ့ရှိပါတယ်။ Identifier က SQL reserved keyword ဖြစ်နေရင်လည်း အလားတူပဲ double quotes သုံးရတယ်။ (မှတ်ချက်။ Identifier ဆိုတာ table, column, function, variable စတာတွေရဲ့ နံမည်ကို ဆိုလိုတာပါ)။

```
-- # and - are special characters
SELECT fname "#1st-name" FROM account_holder;
-- contains space in the column alias
SELECT concat(fname, ' ', lname) "Full Name" FROM account_holder;
-- order is one of the reserved keywords
CREATE TABLE "order" (
    id SERIAL PRIMARY KEY
);
```

Identifier ထဲမှာ " ပါနေရင် "" (double quote နှစ်ခု) နဲ့ escape လုပ်ရပါမယ်။

```
-- Column alias contains double quotes, Students(only "Best")
SELECT name "Students(only ""Best"*)" FROM student
WHERE grade = 'A' OR grade = 'A+';
```

အောက်မှာ ရေးထားတဲ့အတိုင်း စမ်းကြည့်ရင် SQL ဆင်းတက်စံအယ်ရာ ဖြစ်မှာပါ။ နံမည်မှာပါတဲ့

' ကို ' ' ပြောင်းပေးဖို့ လိုတာက တစ်ကြောင်း၊ နောက်တစ်ချက်က နံမည်ဟာ စာသားဖြစ်တဲ့အတွက် SQL ထဲမှာ '0''Brian' ဖြစ်ရပါမယ်။ အခုလို ဆက်ထားရင်

```
# File: db_problem_of_str_interpolation.py
last_name = "0'Brian"
# this will cause SQL syntax error!!!
cur.execute("SELECT * FROM account_holder WHERE lname = " + last_name)
```

SQL string က

```
"SELECT * FROM account_holder WHERE lname = 0'Brian"
```

ဖြစ်နေတာကြောင့် ဆင်းတက်စစ်မမှန်ဘူး။ အမှန်ဖြစ်အောင်က ဒီလို ဆက်ရမှာပါ

```
cur.execute("SELECT * FROM account_holder "
            "WHERE lname = '" + last_name.replace("'", "'") + "'")
```

SQL ကုဒ်ထဲမှာ string က dynamic အပိုင်းဖြစ်နေတဲ့အခါ မမှားအောင် ဂရုစိုက်ရပြီး အတော်လေး ကရိုကထများတယ်။ Identifier တွေက dynamic ဖြစ်နေတယ်၊ double quote လုပ်ရမဲ့ဟာ ဖြစ်နေရင်လည်း အလားတူပြဿနာမျိုး ကြုံရမှာပါ။ ရှေ့ပိုင်းမှာ ဖော်ပြခဲ့တဲ့ နည်းတွေက ဒီလိုကိစ္စတွေကို ကြိုတင်စဉ်းစား ဖြေရှင်းပေးထားတာမို့လို့ ပရိုဂရမ်မှာ သိပ်ခေါင်းစားစရာ မလိုတော့ဘူး

```
cur.execute("SELECT * FROM account_holder WHERE lname = %s", (last_name,))
```

%s နေရာမှာ အစားထိုးပြီး ရမဲ့ SQL ကို mogrify ဖန်ရှင်သုံးပြီး ထုတ်ကြည့်ပါ

```
sql_full = cur.mogrify("SELECT * FROM account_holder "
                       "WHERE lname = %s", (last_name,))
print(sql_full.decode('utf-8'))
```

ဖြစ်သင့်တဲ့အတိုင်း SQL အမှန် တွေ့ရပါလိမ့်မယ်

```
SELECT * FROM account_holder WHERE lname = '0''Brian'
```

SQL Injection

Dynamic SQL ကို string interpolation နည်းတွေနဲ့ ထုတ်တဲ့အခါ ရှေ့မှာဖော်ပြခဲ့တဲ့ အခက်အခဲတွေအပြင် ပိုပြီး နက်ရှိုင်းတဲ့ ပြဿနာတစ်ခု ကြုံရနိုင်ပါတယ်။ အဲ့ဒါကတော့ SQL Injection လို့ခေါ်တဲ့ နည်းလမ်းတစ်မျိုးနဲ့ ဒေတာဘေ့စ် စီကျူရတီပိုင်း တိုက်ခိုက် ခံရနိုင်ခြင်းပါပဲ။ SQL Injection ဆိုတာ ၎င်းလုပ်ဆောင်စေချင်တဲ့ SQL ကုဒ်တွေကို ဟက်ကာက ပရိုဂရမ် input ကနေတစ်ဆင့် ထည့်သွင်းတဲ့ နည်းလမ်းလို့ အကြမ်းဖျဉ်း ယူဆနိုင်တယ်။

အောက်ပါ ပရိုဂရမ်ကုဒ် ကောက်နုတ်ချက်မှာ အသုံးပြုသူ user က last_name ကို input ထည့်ပေးမယ်လို့ ယူဆပါ။ SQL နားလည်ကျွမ်းကျင်တဲ့ ဟက်ကာဟာ သူ့ရဲ့ အကောင့် လက်ကျန်ငွေစာရင်းကို update ဖြစ်သွားစေမဲ့ input string ကို မှန်းဆနိုင်ပါတယ်။

```
# File: db_sql_inj1.py

# SQL injection example
```

```
# ဒီအတိုင်း ထည့်ပေးမယ်လို့ ယူဆပါ
last_name = input("Enter last name: ")
sql = ("SELECT * FROM account_holder "
      "WHERE lname = '" + last_name + "'")
print(sql)
cur.execute(sql)
```

Input ကို အခုလို ထည့်လိုက်မယ် ဆိုပါစို့

```
'; UPDATE account SET balance = 10000000.00 WHERE acc_id = 1;--
```

ဒီလိုသာဆိုရင် ဟက်ကာဟာ သူ့ရဲ့အကောင့်မှာ 10000000.00 ရသွားပါပြီ (!)။ သူ့ input ကြောင့် SQL က အခုလို

```
SELECT * FROM account_holder WHERE lname = '';
UPDATE account SET balance = 10000000.00 WHERE acc_id = 1;--'
```

ဖြစ်သွားတာ တွေ့ရမှာပါ။ နဂိုရည်ရွယ်တာက အကောင့်ပိုင်ရှင်ကို last name နဲ့ ရှာဖွေပေမဲ့ ဟက်ကာရဲ့ input ကြောင့် update လုပ်တဲ့ SQL ပါ တွဲရက် ပါသွားတယ်။ အဆုံးမှာ --' ကို သတိပြုပါ။ -- ဟာ SQL ကွန်းမန်ဖြစ်တဲ့အတွက် နောက်မှာ ဘာပဲရှိရှိ အရေးမကြီးတော့ဘူး (နောက်ဆုံး single quote ကို အယ်ရာ မဖြစ်အောင် ကွန်းမန်ပါ ပိတ်ပေးလိုက်တာ)။ တကယ့်လက်တွေ့မှာ SQL Injection ဟာ ဒီထက် ရှုပ်ထွေးခက်ခဲကောင်း ခက်ခဲနိုင်ပေမဲ့ အခုဥပမာကနေ အခြေခံ သဘောတရား နားလည်နိုင်မယ် မျှော်လင့်ပါတယ်။

SQL injection ကြောင့် ကတ်စတန်မာ ကိုယ်ရေးကိုယ်တာ အချက်အလက်တွေ ပေါက်ကြားကုန် နိုင်ပါတယ်။ ရက်စွဲအလိုက် ငွေဝင်ငွေထွက်စာရင်း စစ်လိုရတယ် ယူဆပါ။ ရက်စွဲကို အောက်ပါအတိုင်း ထည့်ပေးလိုက်ရင် ဟက်ကာဟာ အခြားသူ စာရင်းတွေကိုပါ ကြည့်လိုရသွားပါမယ်။

```
# File: db_sql_inj1.py

acc_id = 1

txn_date = "2024-08-01" OR 1 = 1 --"
sql = ("SELECT * FROM account_transaction WHERE date(txn_date) = '"
      + txn_date + "' AND acc_id = " + str(acc_id))
print(sql)
cur.execute(sql)
transactions = cur.fetchall()

for tx in transactions:
    print(tx)
```

ထွက်လာတဲ့ SQL ကို ကြည့်တဲ့အခါ

```
SELECT * FROM account_transaction
WHERE date(txn_date) = '2024-08-01' OR 1 = 1 --' AND acc_id = 1
```

1 = 1 က true ဖြစ်မယ်။ ဒီတော့ OR သုံးထားတဲ့ WHERE ကွန်ဒီရှင်တစ်ခုလုံးကလည်း ဘယ်တော့မှဆို

true ဖြစ်နေမှာပါ။ နောက်ပိုင်းက ကွန်မန်ဖြစ်သွားတော့ AND acc_id = 1 ကလည်း ထူးခြားမှု မရှိတော့ဘူး။ ဒါကြောင့် အကျိုးသက်ရောက်မှုအရ WHERE မပါဘဲ table တစ်ခုလုံး select လုပ်တာနဲ့ တူတူဖြစ်သွားမှာပါ

```
SELECT * FROM account_transaction WHERE true;
-- , which is effectly the same as below:
SELECT * FROM account_transaction;
```

အခုတွေ့ခဲ့ရတဲ့ ဥပမာမျိုးတွေဟာ ဘယ်လုပ်ငန်းလို ကုမ္ပဏီကြီးတွေအတွက်ဆိုရင် အတော်ကို ပြဿနာကြီးတဲ့ စီကျူရတီ ကျိုးပေါက်မှု ဖြစ်ပါလိမ့်မယ်။ တရားစွဲဆို ခံရတာ ဖြစ်နိုင်တယ်။ ကတ်စာမိမာတွေ လက်လွှတ်ဆုံးရှုံးရနိုင်တယ်။ SQL injection နဲ့ပါတ်သက်လို့ လုံးဝ ပေါ့ဆလို့မရဘူး၊ အထူးဂရုစိုက်ဖို့ လိုမယ်ဆိုတာ ဒီလောက်ဆိုရင် သဘောပေါက်မယ် ထင်ပါတယ်။

ဖတ်ရှုလေ့လာသင့်သည့် စာအုပ်များနှင့် အခြား အရင်းအမြစ်များ

1. Viescas, J. L. *SQL Queries for Mere Mortals*, 4th Edition. Addison-Wesley, 2018.
2. Hernandez M. J. *Database Design for Mere Mortals*, 4th Edition. Addison-Wesley, 2021.
3. Connolly T. M., Begg C. E. *Database Systems: A Practical Approach to Design, Implementation, and Management*, 6th Edition. Pearson, 2015.
4. Silberschatz A., Korth H. F., Sudarshan S. *Database System Concepts*, 7th Edition. McGraw-Hill, 2020.

YouTube Tutorials

1. techTFQ. *Learn Complete SQL (Beginner to Advance)*

https://www.youtube.com/playlist?list=PLavw5C92dz9Ef4E-1Zi9KfCTXS_IN8gXZ

Tutorial က table နဲ့ sample data SQL script တွေကို ဖန်တီးသူရဲ့ blog page မှာ download လုပ်လို့ရပါတယ်

<https://techtfq.com/blog/sql-basics-tutorial-for-beginners>

အခန်း ၂

Basic Concurrency

[[Not completed yet!]]

နောက်ဆက်တွဲ က

PostgreSQL ဒေတာဘေ့စ် ဆာဗာဆော့ဖ်ဝဲ ထည့်သွင်းခြင်း