Basis Data dan Penelusuran Data

Bakti Siregar, M.Sc

2023-09-04

Contents

Kata Pengantar

Selamat datang dalam modul praktikum mengenai basis data dan penelusuran data. Dalam era digital yang semakin maju, pengelolaan informasi dan akses terhadap data sangatlah penting. Basis data merupakan fondasi utama dalam pengelolaan data yang efisien dan terstruktur, sedangkan penelusuran data memungkinkan kita untuk menggali wawasan berharga dari kumpulan informasi yang tersedia. Dalam modul ini, kita akan menjelajahi konsep-konsep dasar dalam basis data, termasuk jenis-jenis basis data, model data, bahasa kueri, dan praktik terbaik dalam merancang basis data yang optimal. Secara khusus, mudul ini

Selain itu, penelusuran basis data yang menjadi fokus penting adalah menggunakan R Programing dan SQL dalam membuat data analytics system. Penelusuran data melibatkan teknik-teknik dan alat-alat untuk menggali informasi yang berharga dari kumpulan data yang besar dan kompleks. Dengan adanya kemajuan dalam analisis data dan kecerdasan buatan, penelusuran data telah menjadi aspek penting dalam pengambilan keputusan dan inovasi. Penulis berharap bimbingan ini akan memberikan pemahaman yang kokoh tentang basis data dan penelusuran data, serta memberi Anda wawasan yang berguna dalam mengelola data dan mengambil informasi berharga dari sumber daya yang ada. Selamat belajar!

Ringkasan Materi

Adapun isi pembelajaran dalam modul ini adalah sebagai berikut:

- Bab 1
- Bab 2
- Bab 3
- Dst

Penulis

- **Bakti Siregar, M.Sc** adalah Ketua Program Studi di Jurusan Statistika Universitas Matana. Lulusan Magister Matematika Terapan dari National Sun Yat Sen University, Taiwan. Beliau juga merupakan dosen dan konsultan Data Scientist di perusahaan-perusahaan ternama seperti JNE, Samora Group, Pertamina, dan lainnya. Beliau memiliki antusiasme khusus dalam mengajar Big Data Analytics, Machine Learning, Optimisasi, dan Analisis Time Series di bidang keuangan dan investasi. Keahliannya juga terlihat dalam penggunaan bahasa pemrograman Statistik seperti R Studio dan Python. Beliau mengaplikasikan sistem basis data MySQL/NoSQL dalam pembelajaran manajemen data, serta mahir dalam menggunakan tools Big Data seperti Spark dan Hadoop. Beberapa project beliau dapat dilihat di link berikut: Rpubs, Github, Website, dan Kaggle.

Asisten Lab

- **Yonathan Anggraiwan, S.Stat** adalah seorang alumni Statistika yang bersemangat dalam dunia pemrograman dan analisis data. Lahir di Tangerang, minatnya terhadap teknologi dan komputer muncul sejak usia dini. Ia tumbuh dengan rasa ingin tahu yang kuat terhadap bahasa pemrograman, dan ini membawanya menuju dunia analisis data menggunakan bahasa pemrograman R dan Python. Selama menjalankan tugas sebagai asisten lab, Yonathan Anggraiwan berperan dalam membantu mahasiswa dalam memahami konsep-konsep dasar dan kompleks dalam pemrograman R dan Python. Ia memberikan penjelasan yang jelas dan dukungan kepada mahasiswa yang mengalami kesulitan. Selain itu, ia juga terlibat dalam merancang tugas dan ujian praktikum, serta memberikan umpan balik konstruktif kepada para mahasiswa. Dalam perjalanan waktu, Yonathan Anggraiwan mulai mengambil tanggung jawab lebih besar dalam laboratorium. Ia membantu mengembangkan materi pembelajaran tambahan, seperti tutorial online tentang analisis data menggunakan R dan Python. Ia juga aktif dalam berbagai proyek penelitian di bawah bimbingan dosen, yang melibatkan pengolahan data besar untuk analisis statistik dan visualisasi. Dengan semangat yang tinggi, dedikasi, dan keterampilan yang dimilikinya, Yonathan Anggraiwan adalah contoh nyata dari seorang mahasiswa yang berhasil menggabungkan minatnya dalam pemrograman R dan Python dengan peran yang produktif sebagai asisten laboratorium dan kontributor dalam dunia analisis data.

Ucapan Terima Kasih

Saya ingin mengucapkan terima kasih yang tulus kepada semua yang telah mendukung dan berkontribusi dalam perjalanan pembuatan modul "Basis Data dan Penelusuran Data". Modul ini tidak akan mungkin menjadi kenyataan tanpa kerja keras, semangat, dan dukungan yang luar biasa dari berbagai pihak. Terima kasih juga kepada rekan-rekan dan kolega yang telah memberikan masukan, saran, dan diskusi berharga sepanjang perjalanan penulisan modul ini. Kontribusi kalian telah membantu memperkaya isi modul dan menghadirkan sudut pandang yang beragam. Tentu saja,modul ini tidak akan lengkap tanpa rasa terima kasih kepada para peneliti dan praktisi di bidang basis data dan penelusuran data yang telah menciptakan landasan pengetahuan yang menjadi dasar dari modul ini. Pengalaman dan pengetahuan yang kalian bagikan sangat berharga. Saya juga ingin mengucapkan terima kasih kepada keluarga dan teman-teman saya atas dukungan, pengertian, dan dorongan yang tak hentihentinya. Tanpa dukungan kalian, perjalanan menulis modul ini pastinya tidak akan semudah ini.

Akhir kata, semoga modul ini dapat memberikan manfaat dan wawasan baru kepada para pembaca yang ingin mendalami dunia basis data dan penelusuran data. Ucapan terima kasih terakhir saya tujukan untuk semua yang telah berkontribusi, baik secara langsung maupun tidak langsung, dalam menghadirkan modul ini kepada para pembaca.

Masukan & Saran

Semua masukan dan tanggapan Anda sangat berarti bagi kami untuk memperbaiki template ini kedepannya. Bagi para pembaca/pengguna yang ingin menyampaikan masukan dan tanggapan, dipersilahkan melalui kontak dibawak ini!

Email: dsciencelabs@outlook.com

Chapter 1

Pendahuluan

Sejak tahun 1970, **Structured Query Language (SQL)** telah digunakan oleh para programmer untuk membangun dan mengakses **Sistem Basis Data (SBD)**. Banyak sekali perdebatan mengenai cara penyebutan SQL ini, namun pada kenyataannya, kita dapat melafalkannya sebagai "sequel" ataupun "S.Q.L". Mempelajari bahasa pemrograman umum seperti R adalah penting dan akan lebih baik jika memiliki kemampuan SQL dalam bidang pengolahan data.

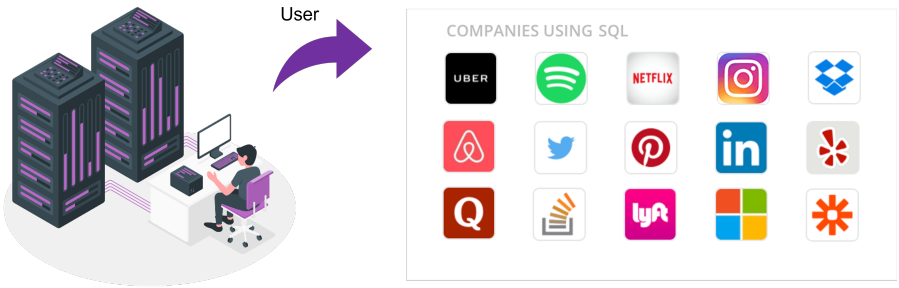

Figure 1.1: Beberapa Perusahaan Besar Pengguna SQL

Banyak perusahaan besar di bidang teknologi menggunakan SQL seperti Uber, Netflix, dan Airbnb. Bahkan dalam perusahaan seperti Facebook, Google dan Amazon, yang telah membuat sendiri **SBD** berkemampuan tinggi, tetap menggunakan SQL untuk melakukan query dan analisis data.

1.1 Apa itu SBD?

Secara umum **SBD** dapat didefinisikan sebagai berikut:

Figure 1.2: Definisi Sistem Basis Data

1.1.1 Komponen SBD

Adapun beberapa komponen dasar yang diperlukan dalam SBD adalah:

1.1.2 Manfaat SBD

Manfaat atau kegunaan penerapan SBD cukup banyak dan cakupannya pun luas dalam mendukung keberadaan lembaga atau organisasi maupun perusahaan, diantaranya:

1.1.3 Definisi SQL vs NoSQL

Sebenarnya perbedaan antara SQL dan NoSQL secara mendasar sudah dapat dijelaskan dari akronimnya.

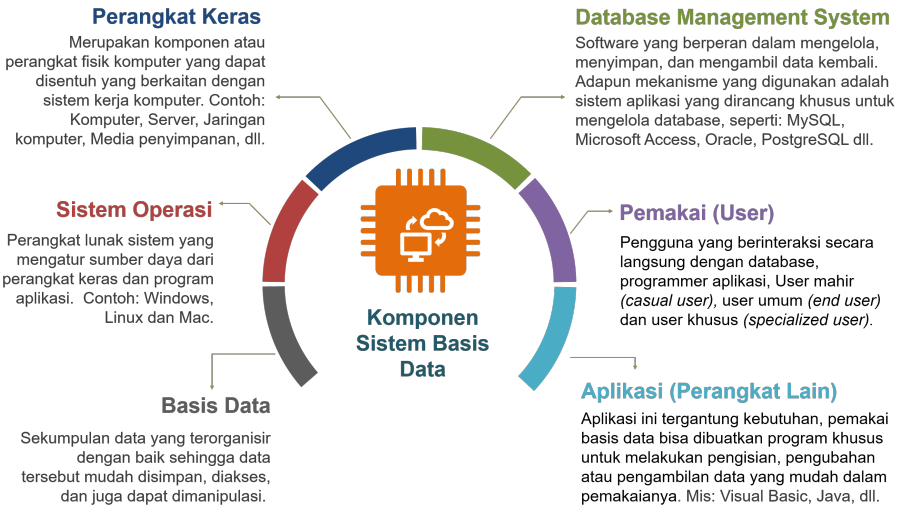

Figure 1.3: Komponen SBD

Figure 1.4: Manfaat SBD

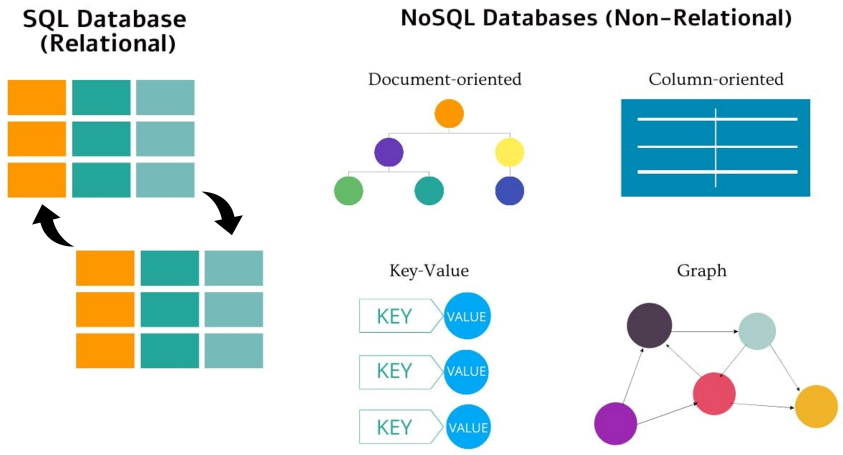

Figure 1.5: SQL vs NoSQL

SQL basis data relasional yang menggunakan 'relasi' (yang biasanya disebut tabel) untuk menyimpan data dan mencocokkan data tersebut dengan memakai karakteristik umum di setiap dataset. Sedangkan, *NoSQL* adalah database yang menggunakan format JSON untuk setiap dokumennya sehingga mudah dibaca dan dimengerti. NoSQL banyak diminati karena memiliki performa yang tinggi dan bersifat non-relasional sehingga dapat memakai berbagai model data.

1.1.4 Perbedaan SQL vs NoSQL

Sebenarnya banyak perbedaan yang dimiliki di antara dua database tersebut tapi inilah perbedaan yang paling mencolok antara SQL dan NoSQL:

1.1.5 Top 7 SQL

Tercatat sampai bulan Februari 2020 ada 334 jenis database menurut db-engines.com. Berikut ini saya merangkum daftar 7 database terpopuler yang menggunakan SQL (Relasional):

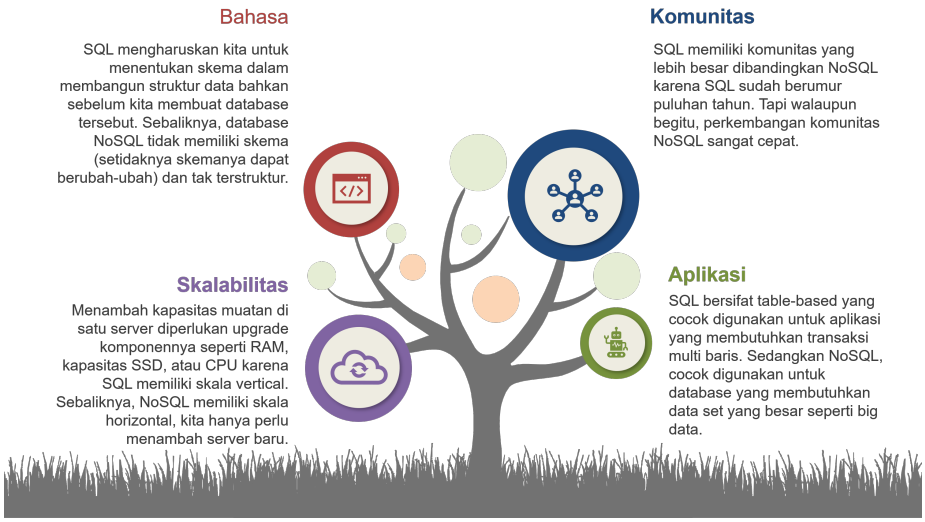

Figure 1.6: Perbedaan SQL vs NoSQL

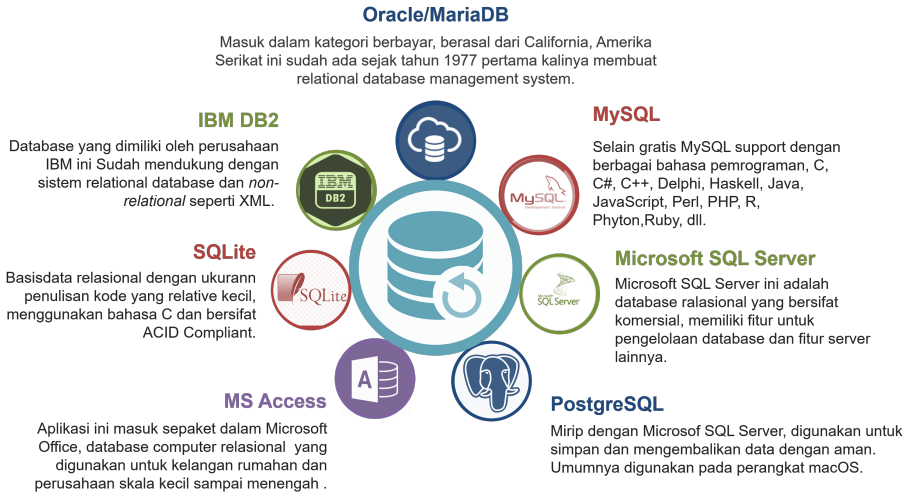

Figure 1.7: Top 7 Perangkat Lunak SQL

1.1.6 Top 8 NoSQL

Kebanyakan basis data NoSQL digunakan dalam dunia aplikasi web waktu nyata (real-time web app). Berikut ini adalah ulasan 8 jenis basis data NoSQL yang paling populer digunakan diseluruh dunia:

Figure 1.8: Top 8 Perangkat Lunak NoSQL

1.2 Mengapa R & SQL?

Menggunakan R dan SQL merupakan kombinasi yang kuat untuk analisis data dan pengelolaan basis data. Keduanya memiliki peran yang berbeda dalam proses analisis dan pengelolaan data. Berikut adalah beberapa alasan mengapa menggunakan R dan SQL bersama:

- **Kekuatan Analisis R**

R adalah bahasa pemrograman yang khusus dirancang untuk analisis statistik dan visualisasi data. R memiliki berbagai paket (packages) yang menawarkan fungsi statistik dan analisis yang kuat, termasuk regresi, pengelompokan, analisis deret waktu, dan banyak lagi. Visualisasi yang dapat dihasilkan dengan R sangat bervariasi, dari grafik sederhana hingga visualisasi interaktif yang kompleks.

- **Manipulasi dan Pengelolaan Data dengan SQL**

SQL digunakan untuk mengelola dan mengambil data dari basis data terstruktur. SQL menyediakan cara efisien untuk membuat, mengubah, menghapus, dan memanipulasi data dalam basis data. SQL memiliki fitur untuk menggabungkan data dari berbagai tabel, melakukan agregasi, dan menyaring data.

- **Integrasi Antara R dan SQL**

Banyak perpustakaan R yang mendukung koneksi ke basis data menggunakan SQL. Anda dapat menggunakan perintah SQL dalam skrip R untuk mengambil data dari basis data, memanipulasi data di dalam R, dan kemudian menerapkan analisis statistik menggunakan paket R. Integrasi ini memungkinkan Anda menggabungkan kekuatan analisis statistik R dengan kemampuan pengelolaan data SQL.

- **Skalabilitas dan Efisiensi**

Menggunakan SQL untuk mengambil dan memanipulasi data dalam basis data bisa lebih efisien daripada melakukannya dalam R, terutama untuk dataset besar. SQL memungkinkan query yang dioptimalkan dan penggunaan indeks untuk kinerja yang lebih baik.

- **Data Preprocessing**

Sebelum menerapkan analisis di R, Anda mungkin perlu melakukan prapemrosesan pada data, seperti membersihkan data, menggabungkan tabel, dan mengisi data yang hilang. SQL dapat membantu dalam melakukan tugas-tugas ini.

Jadi, menggunakan R dan SQL bersama memungkinkan Anda menggabungkan kekuatan analisis statistik R dengan kemampuan pengelolaan data SQL. Ini bisa sangat berguna ketika Anda ingin melakukan analisis data yang luas dan kompleks dari berbagai sumber data yang berbeda.

1.3 MySQL vs PostgreSQL

MySQL adalah sistem manajemen basis data relasional yang memungkinkan Anda untuk menyimpan data sebagai tabel dengan baris dan kolom. Sistem ini populer sehingga digunakan di banyak aplikasi web, situs web dinamis, dan sistem tertanam. PostgreSQL adalah sistem manajemen basis data relasional-objek yang menawarkan lebih banyak fitur daripada MySQL. Sistem ini memberi Anda lebih banyak fleksibilitas dalam tipe data, skalabilitas, konkurensi, dan integrasi data.

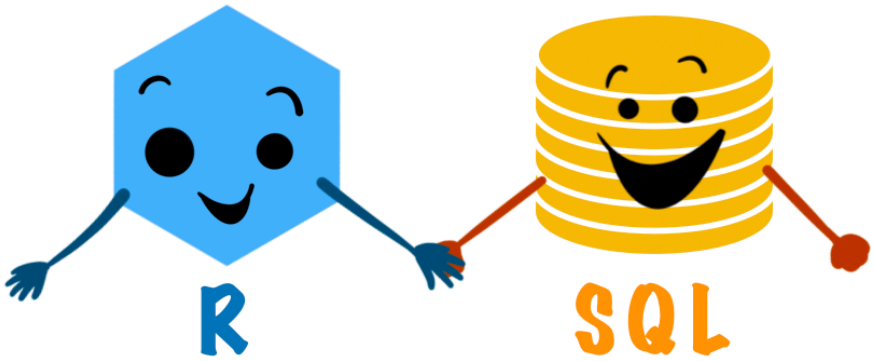

Figure 1.9: R dan SQL [https://irene.rbind.io](https://irene.rbind.io/post/using-sql-in-rstudio/)

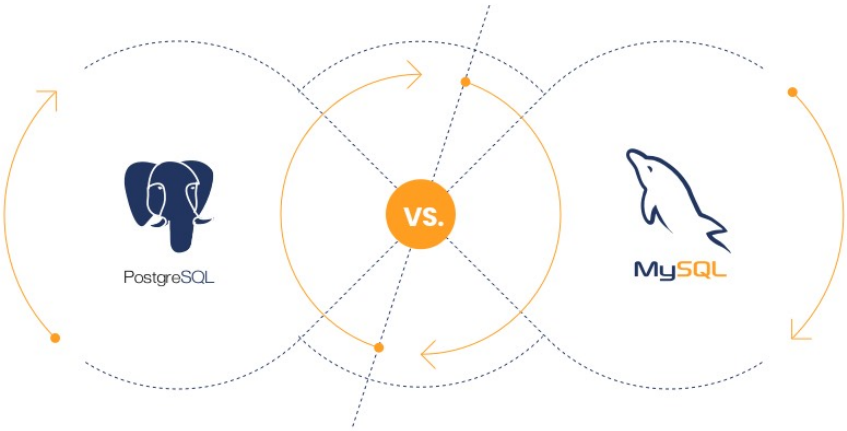

Figure 1.10: MySQL vs PostgreSQL [https://integrio.net/](https://integrio.net/blog/postgresql-vs-mysql)

MySQL dan PostgreSQL, Keduanya menyimpan data di dalam tabel yang terkait satu sama lain melalui nilai kolom umum. Namun keduanya sering dibandingkan karena terdapat beberapa perbedaan. Ingin mengenal lebih dalam? Simak penjelasan di bawah.

1.3.1 Kelebihan

MySQL	PostgreSQL
Integrasi bahasa pemrograman sangat luas;	Support framework website modern seperti Node.js dan Django; Support framework website modern seperti Node.js dan Django;
Aplikasi ringan, tidak membutuhkan spesifikasi hardware yang tinggi;	Dirilis dengan lisensi PostgreSQL sendiri;
Struktur tabel dengan fleksibilitas tinggi;	Bersifat open source dan gratis;
Dibekali banyak administrative tools;	Skala besar, mampu memuat hingga ribuan transaksi data;
Bersifat open source dan gratis (versi basic);	Memiliki banyak fitur yang mumpuni;
Meski open source, MySQL menjamin keamanan tingkat tinggi;	Memiliki banyak fitur yang mumpuni;
Mendukung berbagai variasi Data Type;	Performa sangat baik meski menuntut query yang lebih kompleks;
Dapat digunakan banyak pengguna karena mendukung multi user.	Kecepatan analisis (read-write) sangat cepat; Keamanan yang lebih ketat.

1.3.2 Kekurangan

MySQL	PostgreSQL
Sistem manajemen database kurang cocok untuk aplikasi mobile dan game;	PostgreSQL tidak mendukung semua stack development;
Technical support MySQL dinilai kurang baik;	Meski memiliki integrasi dan skalabilitas tinggi, kecepatan PostgreSQL kalah unggul dibandingkan RDBMS lain;

MySQL	PostgreSQL
Sulit diaplikasikan untuk manajemen database berskala besar.	Sistem kompatibilitas PostgreSQL menuntut pengguna untuk bekerja lebih keras dalam perbaikan dan perawatan.

1.4 Instalasi MySQL (XAMPP)

1.4.1 Download Aplikasi XAMPP

Silakan klik disini untuk mengunduh applikasi XAMPP, pilih salah satu saja sesuai Operating System pada Komputer anda.

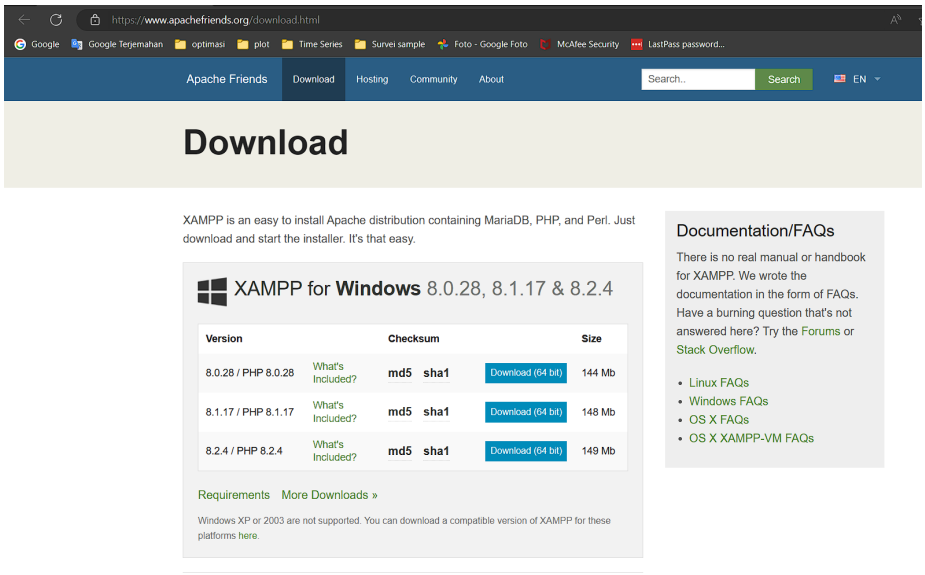

Figure 1.11: Langkah 1, Download XAMPP)

1.4.2 Install Aplikasi

Temukan file XAMPP.exe yang telah anda download, secara default biasanya disimpan di;

Selanjutnya, akan muncul Warning di klik **OK**

selajutunya klik next

Klik next lagi, karena sudah dipilih secara default oleh XAMPP

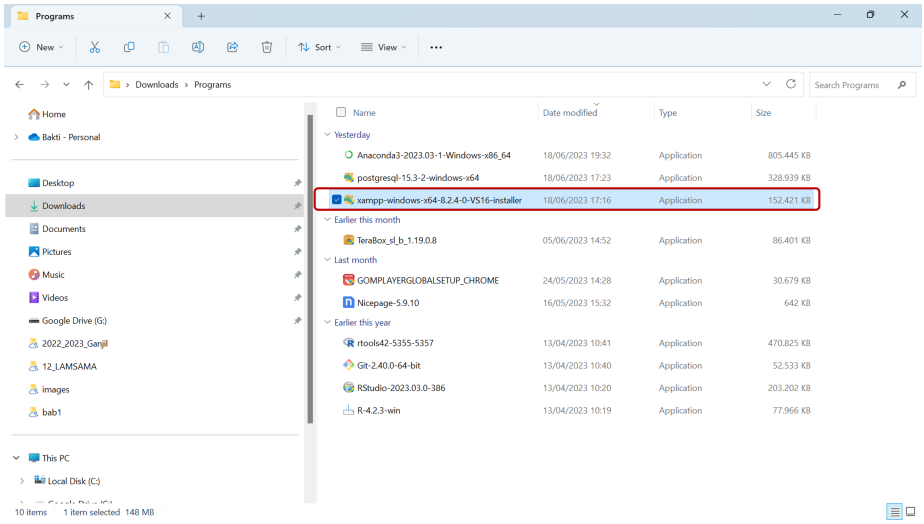

Figure 1.12: Langkah 2, Instalasi XAMPP)

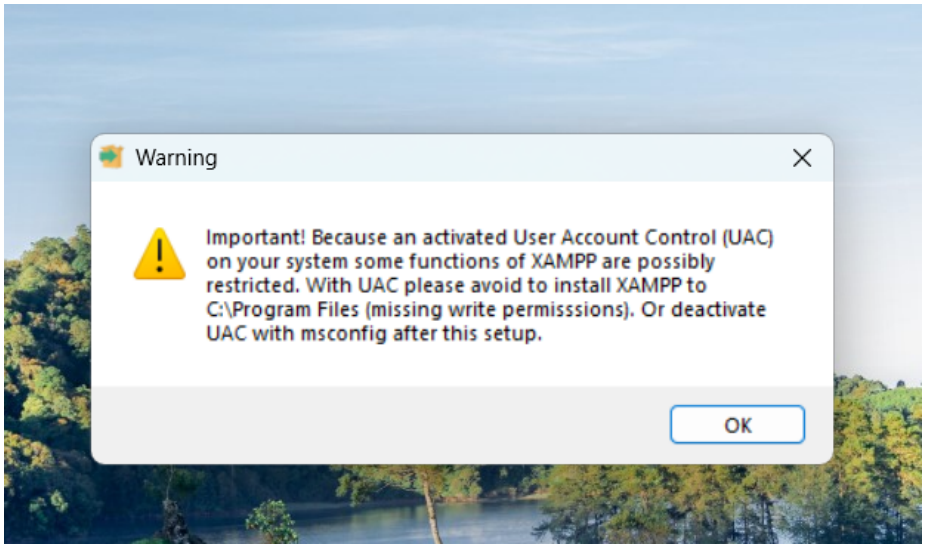

Figure 1.13: Langkah 3, Instalasi XAMPP)

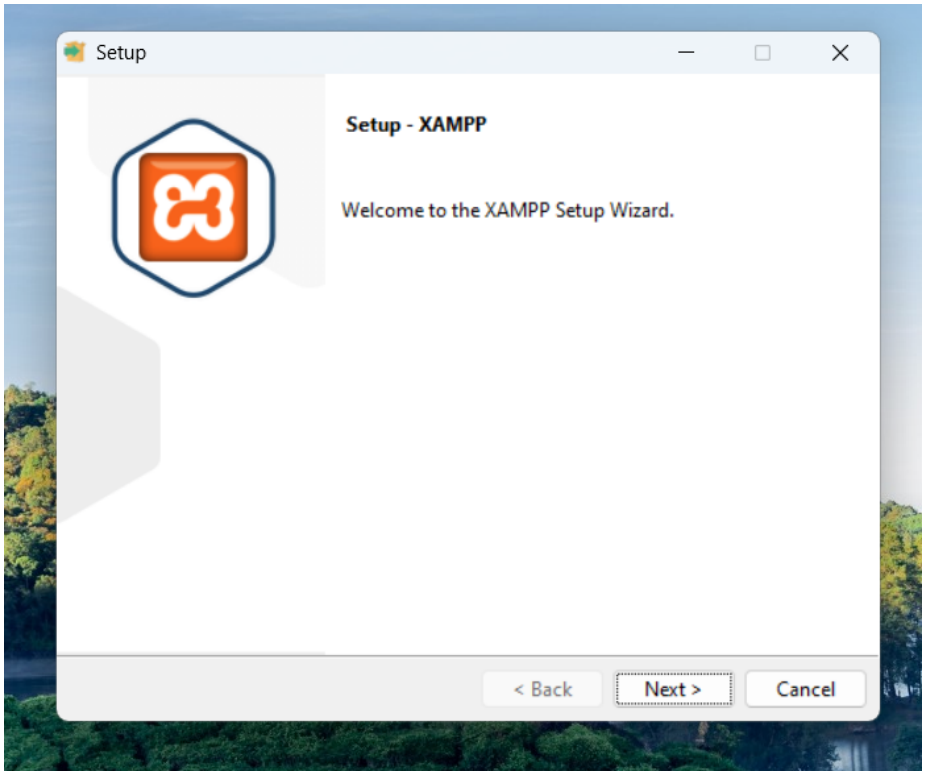

Figure 1.14: Langkah 4: Instalasi XAMPP)

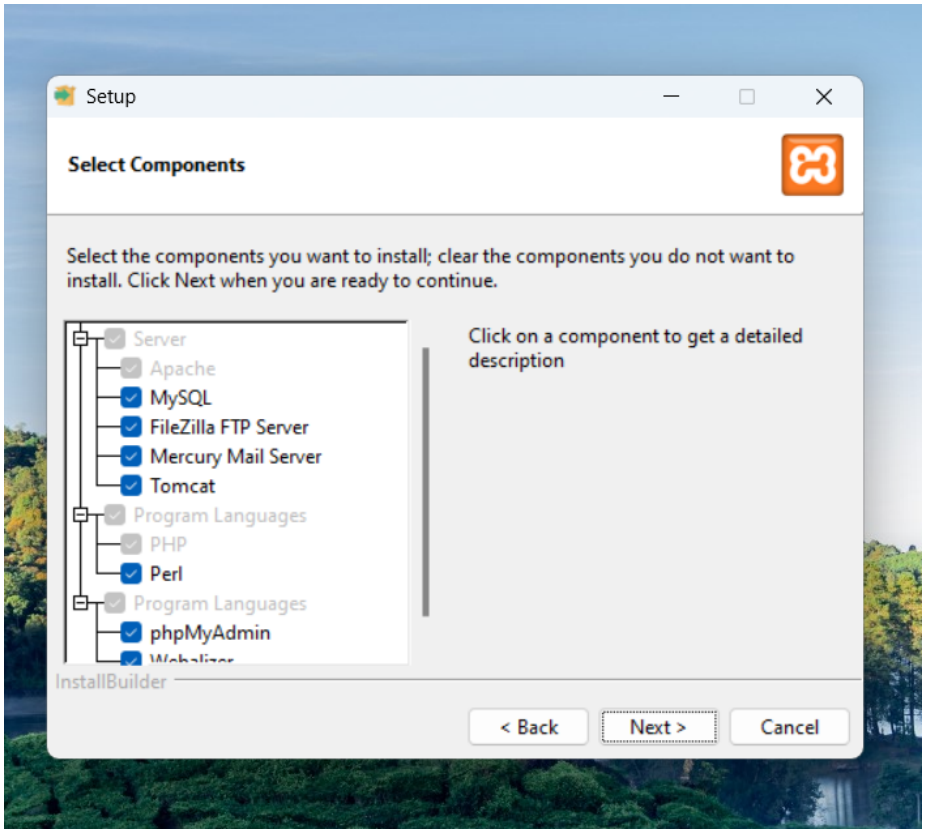

Figure 1.15: Langkah 5, Instalasi XAMPP)

1.4.3 Pilih Folder

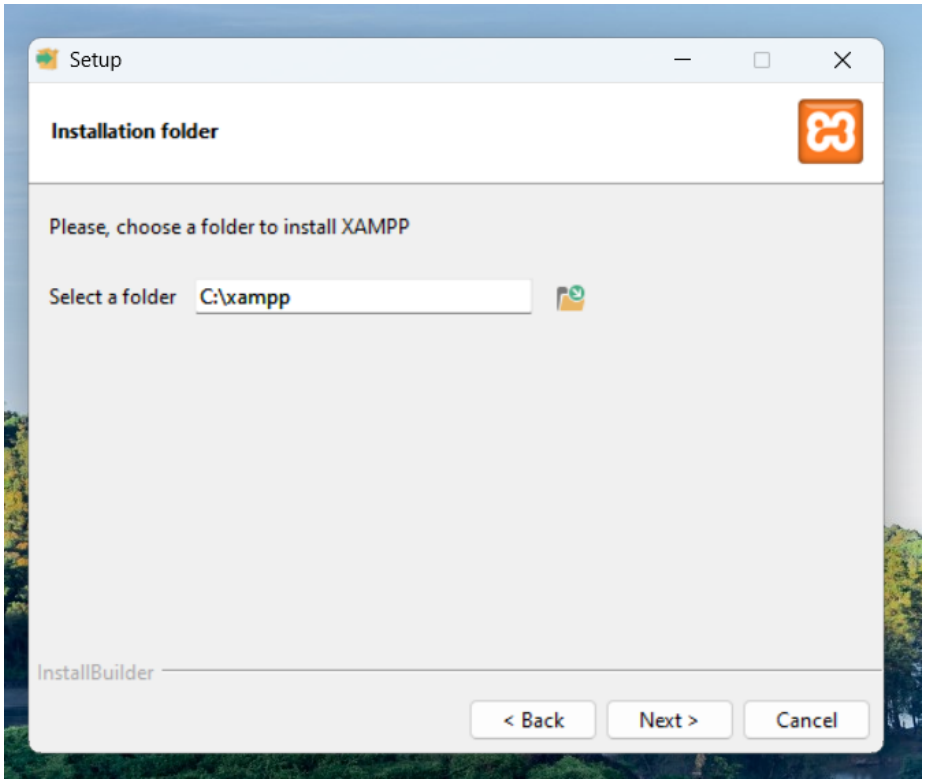

Figure 1.16: Langkah 6, Instalasi XAMPP)

Secara default akan membuat folder baru **C:\xampp**, lalu pilih next.

note: jika anda sudah pernah mendownload aplikasi xampp, perlu di hapus terlebih dahulu file xampp yang lama di file **C:\xampp**

1.4.4 Jalankan proses Instalasi

Tunggu proses instalasi selesai **Biasanya 5-10 menit, tergantung kecepatan komputer anda**.

1.4.5 XAMPP sudah terinstall

Setelah aplikasi terinstall, sudah bisa digunakan.

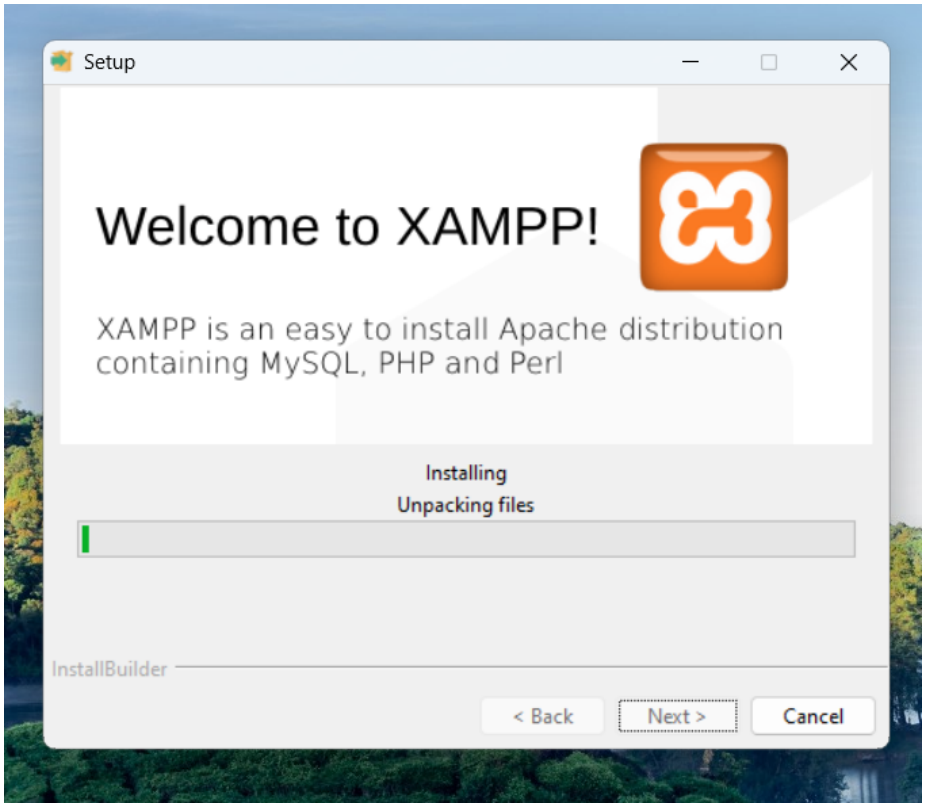

Figure 1.17: Langkah 7, Instalasi XAMPP)

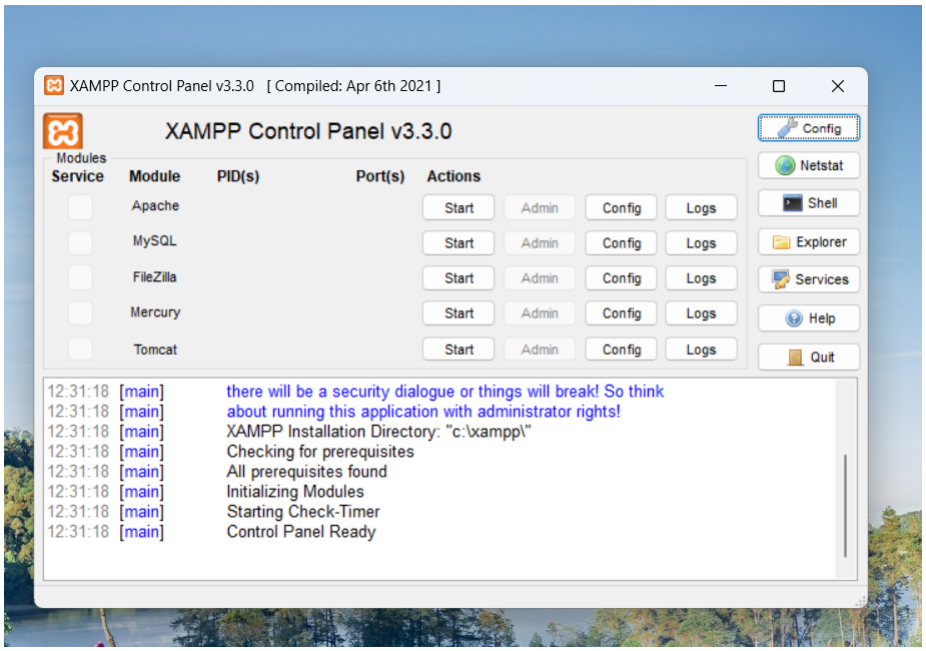

Figure 1.18: Langkah 8, Instalasi XAMPP)

1.4.6 Video Instalasi XAMPP

1.5 Instalasi PostgreSQL

Berikut ini adalah proses langkah demi langkah tentang Cara Menginstal PostgreSQL di Windows:

1.5.1 Buka Browser

Klik https://www.postgresql.org/download and pilih Windows

1.5.2 Cek Option

Klik Download the installer Interactive Installer by EnterpriseDB

1.5.3 Pilih PostgreSQL version

Anda akan diminta untuk memilih versi PostgreSQL dan sistem operasi yang diinginkan. Pilih versi PostgreSQL terbaru dan OS sesuai dengan environment

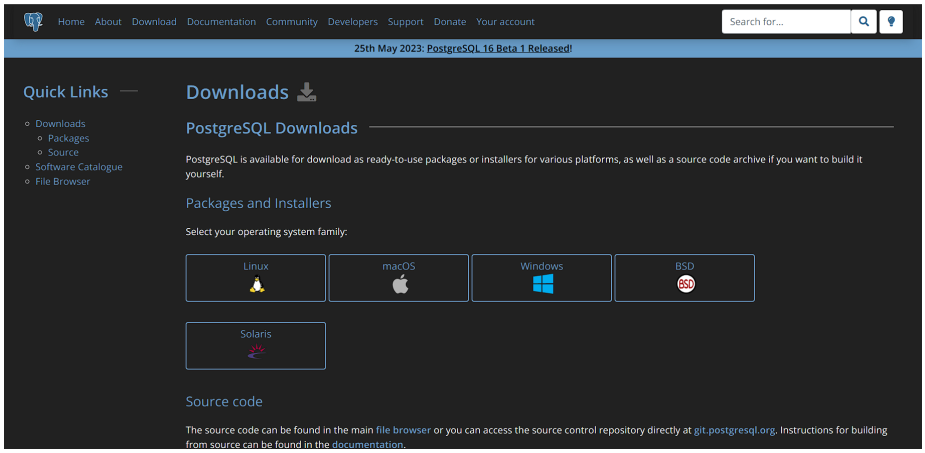

Figure 1.19: Langkah 1, Instalasi PostgreeSQL)

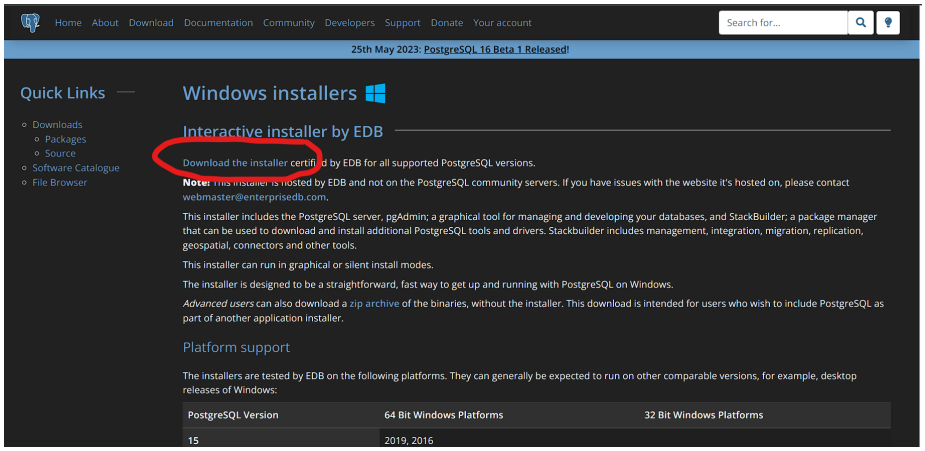

Figure 1.20: Langkah 2, Instalasi PostgreeSQL)

Anda, **klik tombol download.**

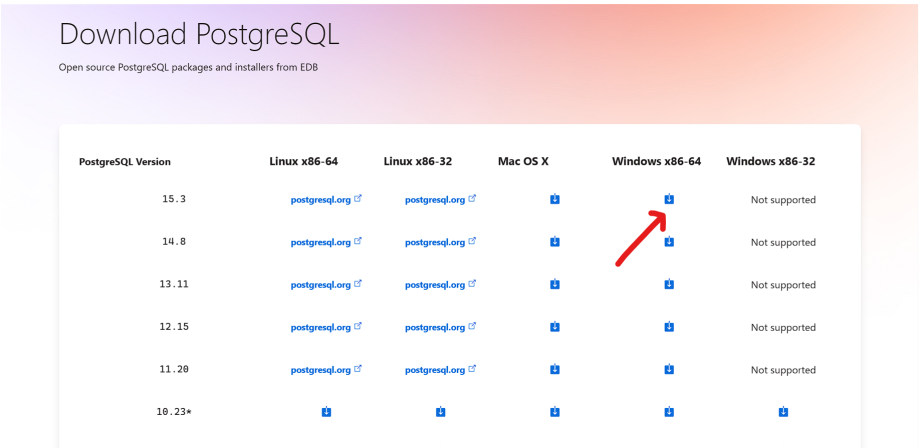

Figure 1.21: Langkah 3, Instalasi PostgreeSQL)

1.5.4 Open exe file

Setelah Anda mengunduh PostgreSQL, buka exe yang telah diunduh dan Klik berikutnya pada layar install welcome screen.

1.5.5 Pilih folder

Ubah direktori Instalasi jika diperlukan, jika tidak, biarkan default, **klik Next.**

1.5.6 Select components

Anda dapat memilih komponen yang ingin Anda instal di sistem Anda. Anda dapat menghapus centang pada Stack Builder (*disarankan ikuti secara default*), **klik Next.**

1.5.7 Check data location

Anda dapat mengubah lokasi data, **Klik Next.**

1.5.8 Masukan Password

Masukkan kata sandi superuser. Catat kata sandi tersebut, **Klik Next.**

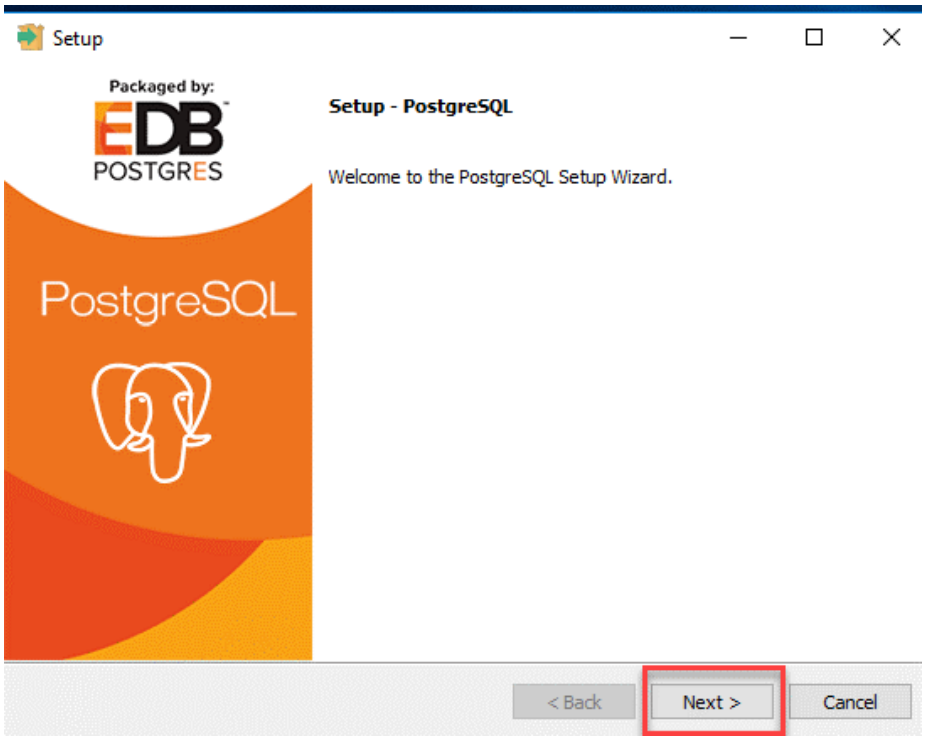

Figure 1.22: Langkah 4, Instalasi PostgreeSQL)

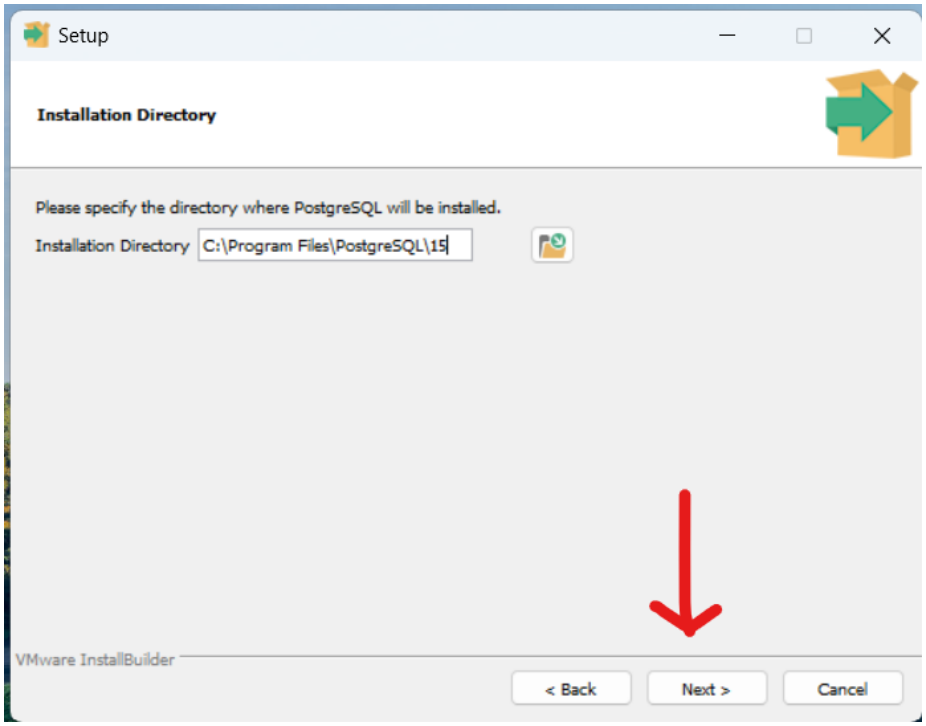

Figure 1.23: Langkah 5, Instalasi PostgreeSQL)

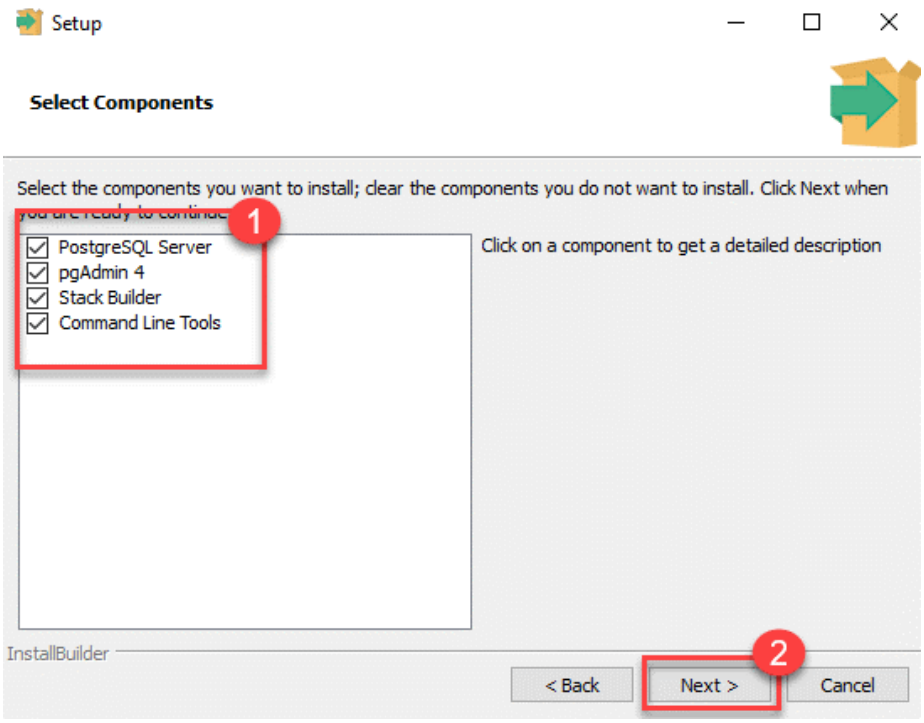

Figure 1.24: Langkah 6, Instalasi PostgreeSQL)

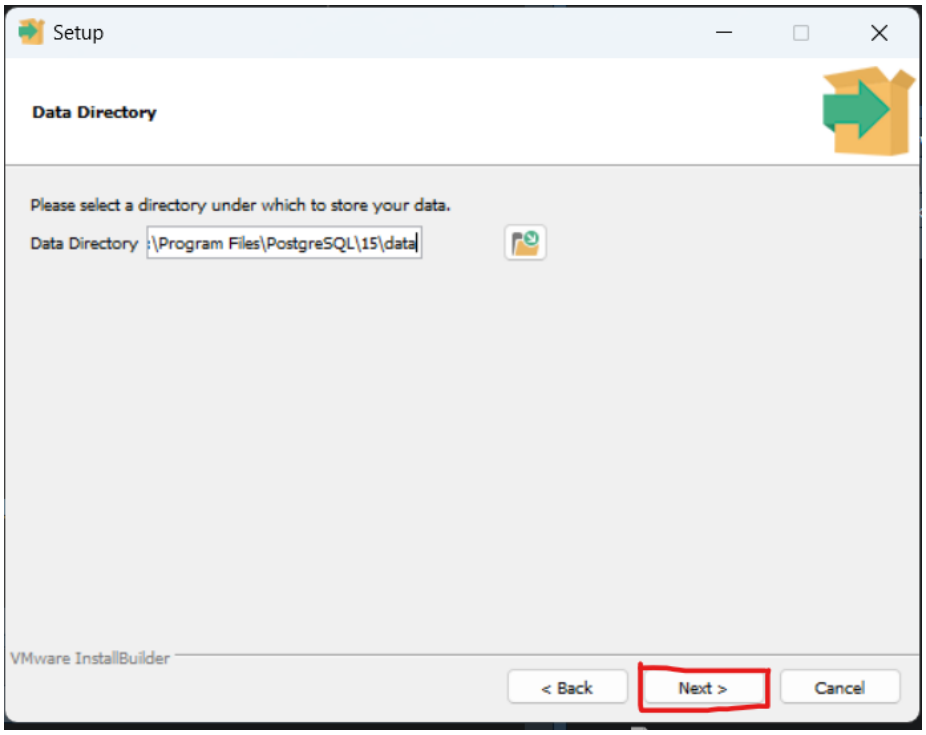

Figure 1.25: Langkah 7, Instalasi PostgreeSQL)

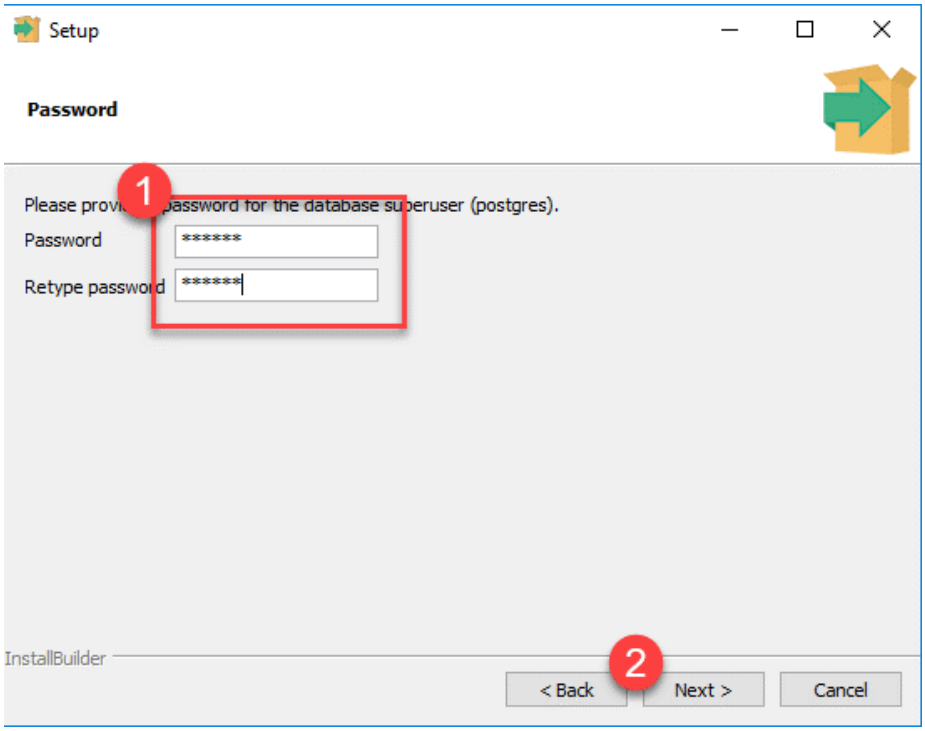

Figure 1.26: Langkah 8, Instalasi PostgreeSQL)

1.5.9 Cek opsi port

Biarkan nomor port menjadi default, **Klik Next.**

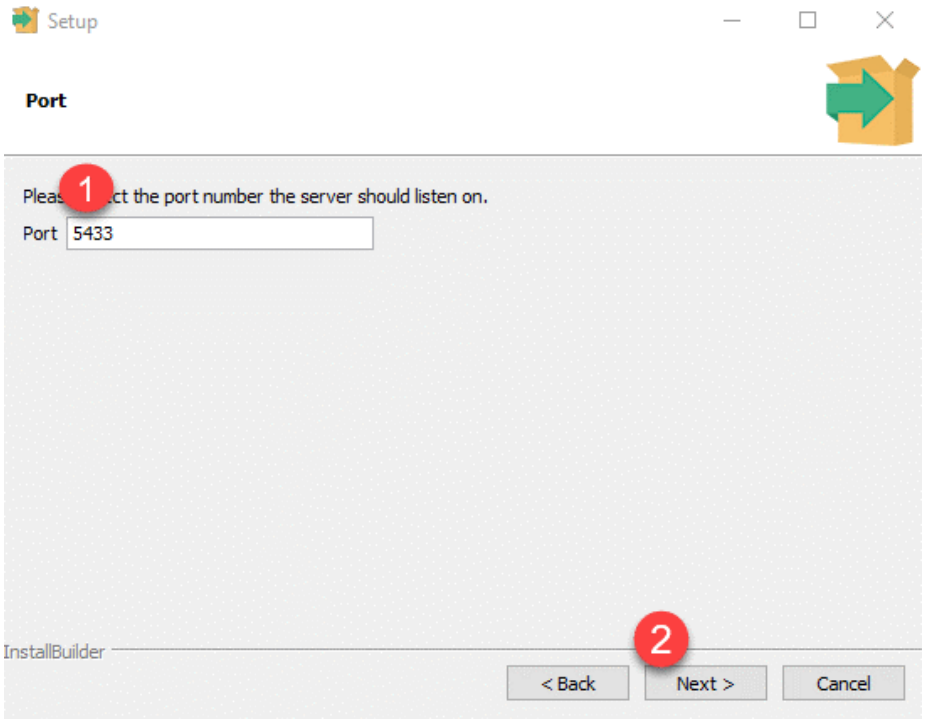

Figure 1.27: Langkah 9, Instalasi PostgreeSQL)

1.5.10 Cek Summary

Periksa pra-penginstalan summary, **Klik Next**

1.5.11 Ready to Install

Klik tombol Next

1.5.12 Check stack builder prompt

Setelah instalasi selesai, Anda akan melihat prompt Stack Builder. Hapus centang pada opsi tersebut. Kita akan menggunakan Stack Builder dalam tutorial selanjutnya, **Klik Finish.**

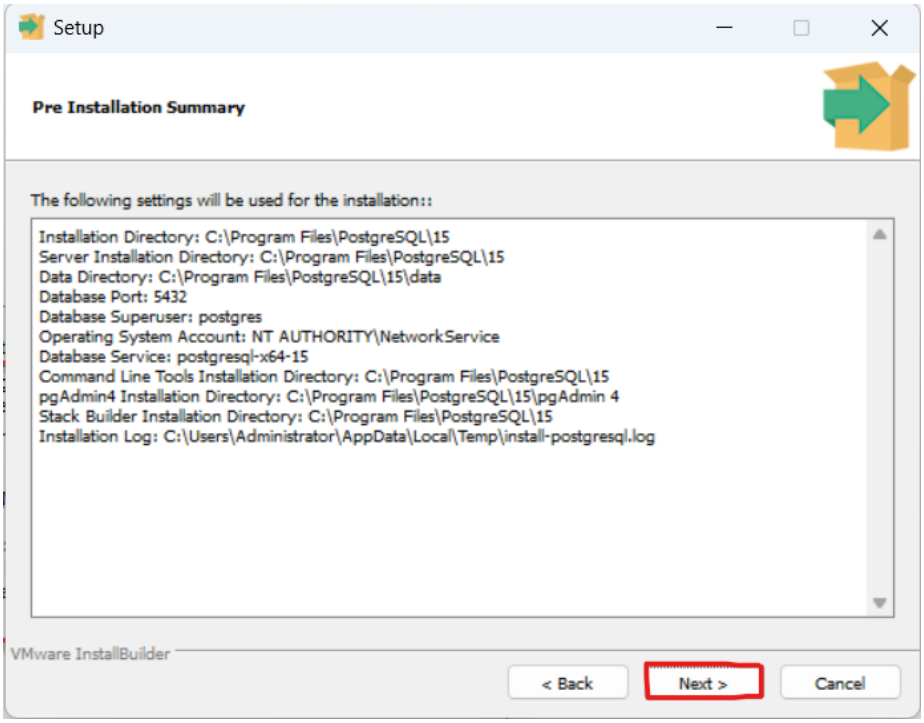

Figure 1.28: Langkah 10, Instalasi PostgreeSQL)

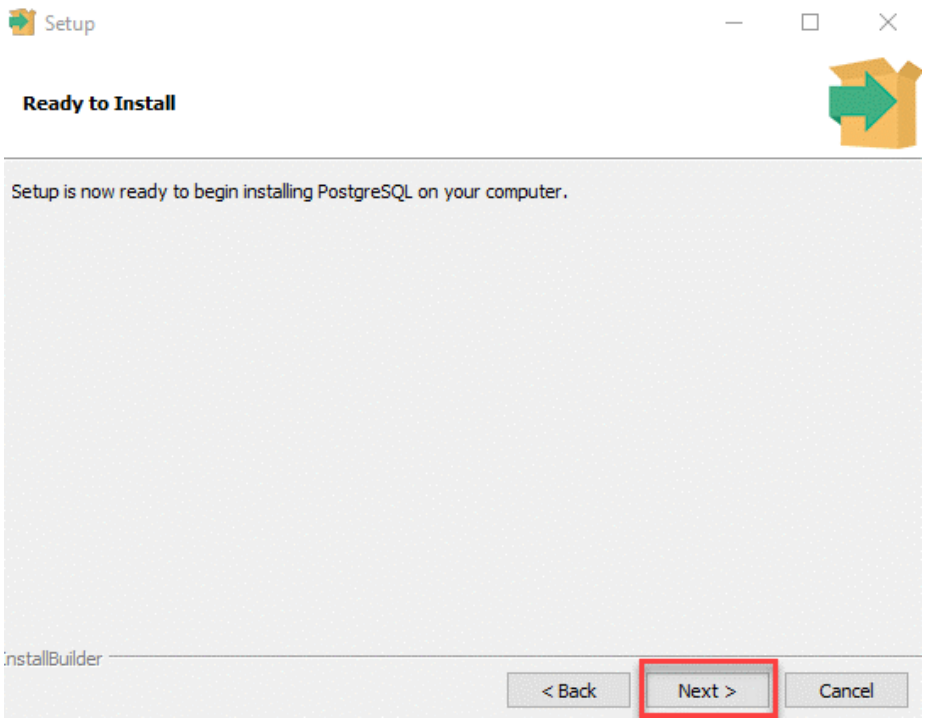

Figure 1.29: Langkah 11, Instalasi PostgreeSQL)

Figure 1.30: Langkah 12, Instalasi PostgreeSQL)

1.5.13 Launch PostgreSQL

Untuk launch PostgreSQL, buka Start Menu dan cari pgAdmin 4

1.5.14 Check pgAdmin

Anda akan melihat beranda pgAdmin

1.5.15 Cari PostgreSQL 15

Klik pada Servers > PostgreSQL 15 di sub sebelah kiri

1.5.16 Enter password

Masukkan kata sandi superuser yang ditetapkan selama instalasi, **Klik OK**

1.5.17 Cek Dashboard

Anda akan melihat Dashboard

1.5.18 Video Instalasi PostgreSQL

1.6 Praktikum

- Tutorial di MySQL (CREATE & Drop Database, Create & Drop Tabel)
- Tutorial di PostgreeeSQL (CREATE & Drop Database, Create & Drop Tabel)

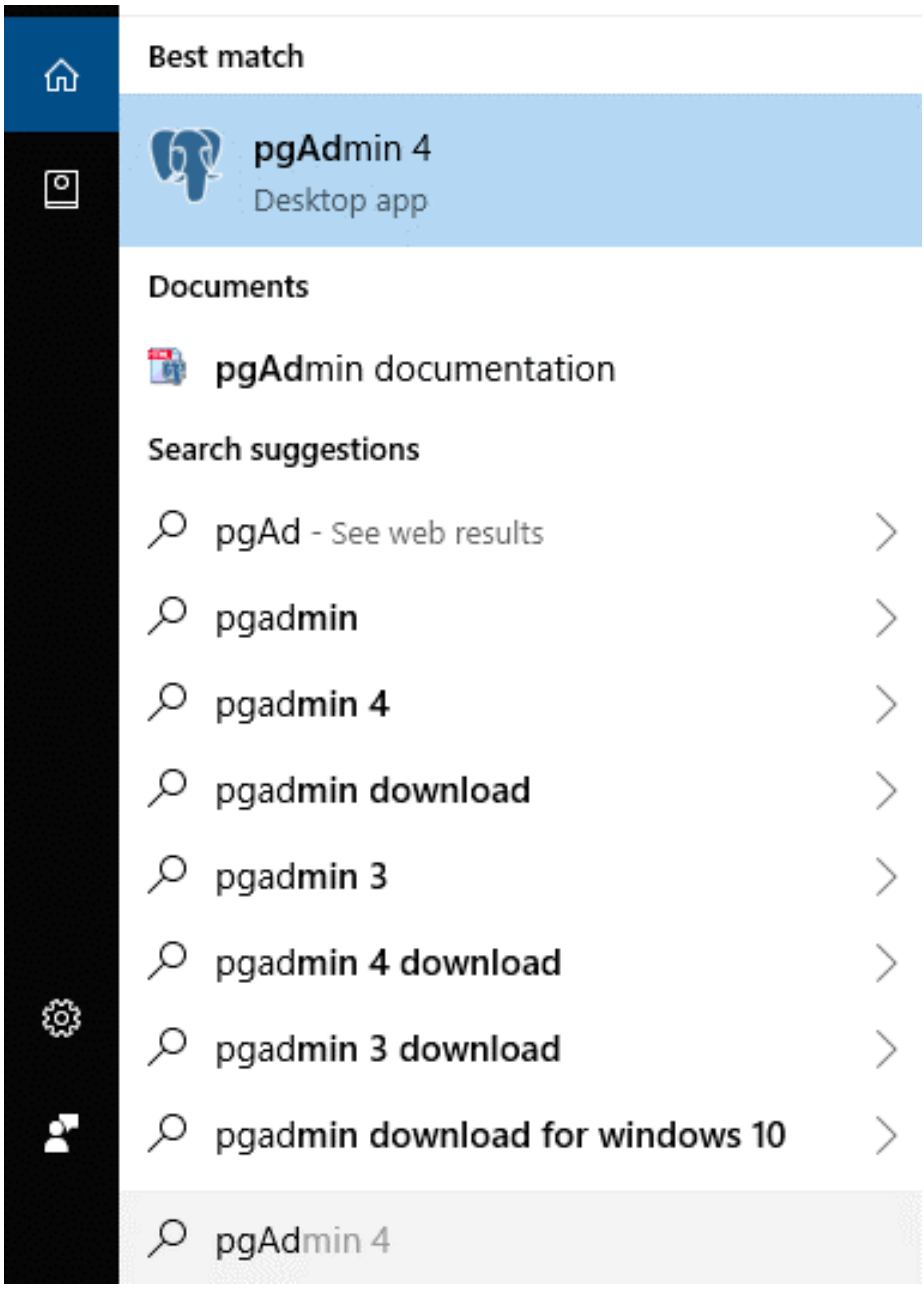

Figure 1.31: Langkah 13, Instalasi PostgreeSQL)

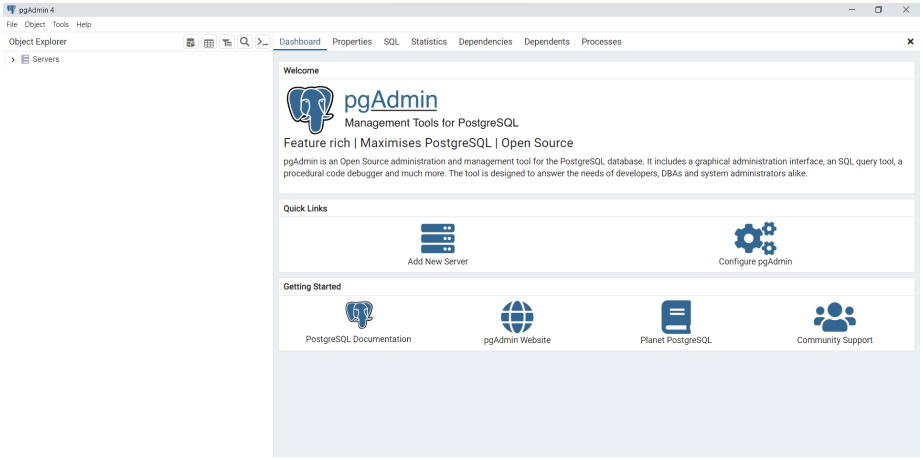

Figure 1.32: Langkah 14, Instalasi PostgreeSQL)

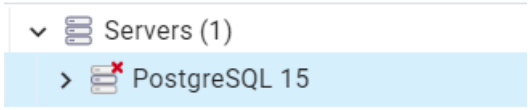

Figure 1.33: Langkah 15, Instalasi PostgreeSQL)

Figure 1.34: Langkah 16, Instalasi PostgreeSQL)

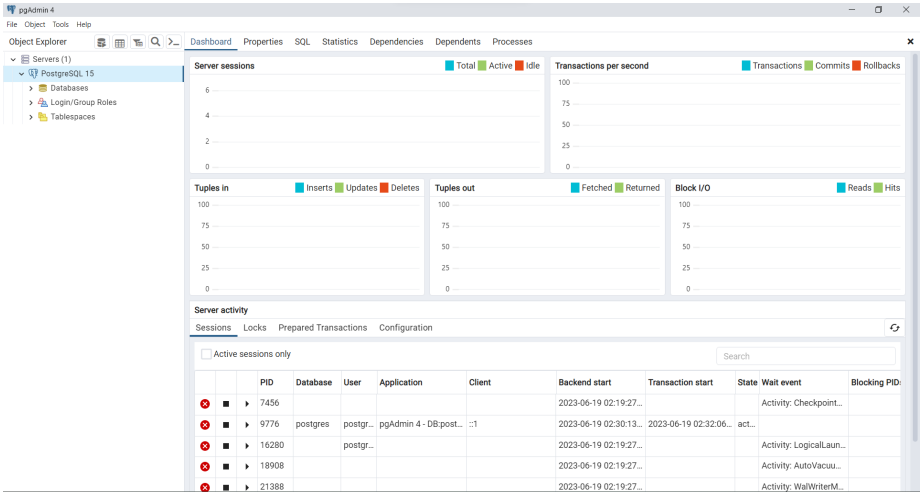

Figure 1.35: Langkah 17, Instalasi PostgreeSQL)

Chapter 2

Connecting R to SQL

2.1 Introduction

A database is a structured set of data. Terminology is a little bit different when working with a database management system compared to working with data in R.

- **field:** variable or quantity
- **record:** collection of fields
- **table:** collection of records with all the same fields
- **database:** collection of tables

The relationship between R terminology and database terminology is explained below.

R terminology	Database terminology
column	field
row	record
data frame	table
types of columns	table schema
collection of data	frames database

2.2 Connecting R to SQL

Connecting R to SQL databases allows you to leverage the power of R for data analysis while directly interacting with and querying data stored in relational

databases. This connection enables you to retrieve, manipulate, and analyze data using SQL queries within your R environment.

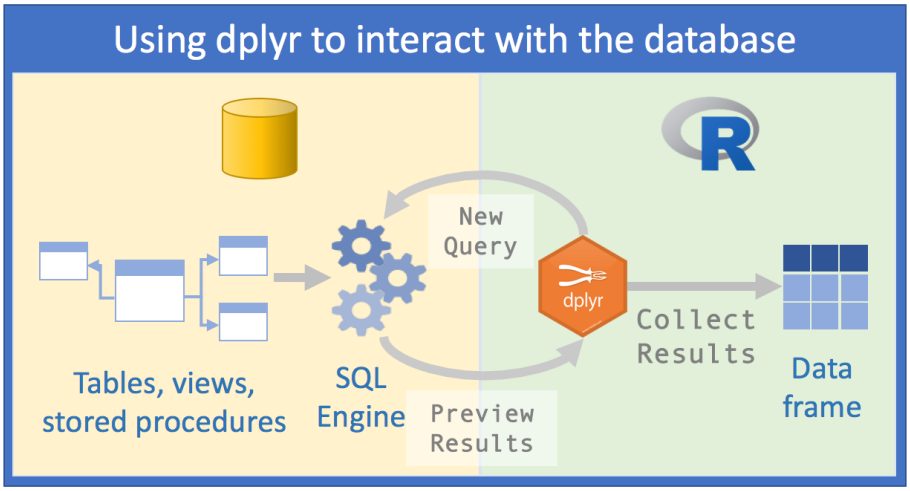

Figure 2.1: Connecting R to SQL [https://rviews.rstudio.com](https://rviews.rstudio.com/2017/05/17/d using-r/)

Here's a step-by-step introduction to connecting R to an SQL database:

2.2.1 Install Required Packages

First, you need to install R packages that facilitate database connections and SQL interactions. The DBI package provides a common interface for database connections, and you'll also need a database-specific package like RMySQL for MySQL, RPostgreSQL for PostgreSQL, or RODBC for ODBC connections.

You can install these packages using the following commands:

```
install.packages(c(
                   "RMariaDB",
                   "RMySQL",
                   "RPostgres",
                   "RSQLite",
                   )
                 )
```

2.2.2 Load Packages

Then, load all these requirement packages:

```
library("RMariaDB")                    # Database Interface and 'MariaDB' Driver
library("RMySQL")                      # Database Interface and 'RMySQL' Driver
library("RPostgres")                   # Database Interface and 'RPostgres' Driver
library("RSQLite")                     # Database Interface and 'RSQLite' Driver
```

2.2.3 Establish a Connection

There are many ways to connect your database with R. This article shows you three of the most common ways:

MariaDB

You'll need to establish a connection to the MySQL server first. Use the `dbConnect()` function to create a connection. Provide the necessary connection information, such as the host, user, and password. For example:

```
MariaDB<- dbConnect(RMariaDB::MariaDB(),
                    user='root',
                    password='',
                    host='localhost')
MariaDB
```

To get a list of all databases on the MySQL server, you can use the dbListTables() function. It provides a list of tables in the currently selected database. To see all databases, you can run an SQL query using the dbGetQuery() function. Here's an example:

```
dbGetQuery(MariaDB, "SHOW DATABASES")          # daftar semua database
```

To create a database, you can use the dbExecute() function to run a SQL query that creates the database. To drop (delete) a database, you can use the dbExecute() function with an SQL query that deletes the database. Be careful when dropping a database, as all data in the database will be permanently deleted. Here's an example:

```
dbExecute(MariaDB,"CREATE DATABASE new_MariaDB") # create a database
dbExecute(MariaDB,"DROP DATABASE new_MariaDB")    # Drop a Database
```

Or, you can create/drop a database by using IF Exist Statement,

```
dbExecute(MySQL,"CREATE DATABASE IF NOT EXISTS new_MariaDB") # create a database
dbExecute(MariaDB1,"DROP DATABASE IF EXISTS new_MariaDB")    # Drop a Database
```

To list the tables in the selected database, you can use the dbListTables() function:

```
dbListTables(MariaDB)                          # table list on your database
```

MySQL

We can perform similar database operations using the MySQL package in R as you would with the MariaDB package. Both packages provide database connectivity and SQL query execution capabilities, and their usage is quite similar.

Here's how you can establish a database connection and list tables in a specific database using the MySQL package:

```
MySQL <- dbConnect(MySQL(),
                   user='root',
                   password='',
                   dbname='coba_coba',
                   host='localhost')
dbListTables(MySQL)                              # table list on your database
dbExecute(MySQL,"CREATE DATABASE new_MySQL")     # Create a new Database
dbExecute(MySQL,"DROP DATABASE new_MySQL")       # Drop a Database
```

Postgres

Certainly! You can use similar approaches to connect to and interact with a MySQL database using the RMariaDB package in R as you would with packages like RPostgreSQL for PostgreSQL. Both packages provide similar functionalities for connecting to and querying databases, but the specific functions and connection parameters may differ.

Here's a comparison between connecting to a MySQL database using RMariaDB and connecting to a PostgreSQL database using RPostgreSQL:

```
postgres <- dbConnect(RPostgres::Postgres(),
                   user='postgres',
                   password='123',
                   dbname='postgres',
                   host='localhost')
```

```
dbListTables(postgres)                          # table list on your database
dbExecute(postgres,"CREATE DATABASE new_PG")    # Create a new Database
dbExecute(postgres,"DROP DATABASE coba_coba")   # Drop a Database
```

SQLite

Last but not least, you can perform similar database operations in RMariaDB as you would in packages like SQLite or any other database package in R. The general concepts and functions for database operations are quite similar across different database management systems. Here is the RSQLite example;

```
RSQLite <- dbConnect(RSQLite::SQLite(), "MySQLite.sqlite")
dbListTables(RSQLite)                           # table list on your database
```

- *Notes:* RSQLite will store the database you created in your current working directory.

2.3 Import Data

This section can be ignored if the data (table) that you need is already registered in your database. If not, then it is necessary to import data set according to your available files, download it below:

- Customers.csv
- Categories.csv
- Employees.csv
- OrderDetails.csv

- Orders.csv
- Products.csv
- Shippers.csv
- Suppliers.csv
- RawDatabase.xlsx

2.3.1 CSV Files

When you're working with files in R, such as reading data from a CSV file or saving plots as image files, R needs to know the location of these files. By setting the working directory, you provide a starting point for R to look for and save files.

In R, the setwd() function is used to set the working directory for your R session. The working directory is the folder on your computer where R will look for files and where it will save files unless you specify a different location. Here's why the setwd() function is important and when you might use it:

```
# Set the working directory
setwd("/path/to/your/folder")
```

```
# Now you can read CSV files without specifying the full path
data <- read.csv("file.csv")
```

Replace "/path/to/your/folder" with the actual path to the folder containing your CSV files. Then, you can run the following code!.

```
Customers   <-read.csv("data/Customers.csv")
Categories  <-read.csv("data/Categories.csv")
Employees   <-read.csv("data/Employees.csv")
OrderDetails<-read.csv("data/OrderDetails.csv")
Orders      <-read.csv("data/Orders.csv")
Products    <-read.csv("data/Products.csv")
Shippers    <-read.csv("data/Shippers.csv")
Suppliers   <-read.csv("data/Suppliers.csv")
```

2.3.2 XLSX Files

```
library("readxl")
Customers   <-read_excel("data/RawDatabase.xlsx",sheet=1)
Categories  <-read_excel("data/RawDatabase.xlsx",sheet=2)
Employees   <-read_excel("data/RawDatabase.xlsx",sheet=3)
OrderDetails<-read_excel("data/RawDatabase.xlsx",sheet=4)
Orders      <-read_excel("data/RawDatabase.xlsx",sheet=5)
Products    <-read_excel("data/RawDatabase.xlsx",sheet=6)
Shippers    <-read_excel("data/RawDatabase.xlsx",sheet=7)
Suppliers   <-read_excel("data/RawDatabase.xlsx",sheet=8)
```

2.4 Write Dataframe to Database

The key here is the `dbWriteTable` function which allows us to write an R data frame directly to a database table. The data frame's column names will be used as the database table's fields. In the following example I use `RMariaDB` connection, you can apply another driver as you like.

```
new_con <- dbConnect(MariaDB(),
                     user='root',
                     password='',
                     dbname='new_MariaDB',
                     host='localhost')

dbWriteTable(new_con, "Customers", Customers, append=T)
dbWriteTable(new_con, "Categories", Categories, append=T)
dbWriteTable(new_con, "Employees", Employees, append=T)
dbWriteTable(new_con, "OrderDetails", OrderDetails, append=T)
dbWriteTable(new_con, "Orders", Orders, append=T)
dbWriteTable(new_con, "Products", Products, append=T)
dbWriteTable(new_con, "Shippers", Shippers, append=T)
dbWriteTable(new_con, "Suppliers", Suppliers, append=T)
```

Note: Some important things that must be considered when storing table data are as follows:

- Data Structure adjustments
- Changes Data type (especially, Date and Time)

In this case, we have a problem with the data table `Employees` and `Orders`. When you consider these Table (Employees and Orders), you will find there is no date are written correctly in the database. In order to handle this problem, just type the following code in your R console:

```
dbRemoveTable(new_con, "Orders")
```

```
Orders["OrderDate"] <-as.Date(Orders$OrderDate, format = "%Y-%m-%d")
```

```
dbWriteTable(new_con, "Orders", Orders, append=T)
```

Your Exercise: Do the same thing to update data table `Employees`

2.5 Basic SQL in R

2.5.1 SELECT

The SELECT statement is used to select data from a database.

```
library(DT)
df1<-dbGetQuery(new_con,'SELECT city
                         FROM Customers')
datatable(df1)
```

2.5.2 WHERE

The WHERE clause is used to filter records, extract only those records that fulfill a specified condition.

```
df2<-dbGetQuery(new_con,"SELECT *
                         FROM Customers
                         WHERE Country='Germany'")
datatable(df2)
```

2.5.3 INSERT INTO

If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. The INSERT INTO syntax would be as follows:

```
dbExecute(new_con,"INSERT INTO Customers(CustomerName,ContactName,Address,City,PostalC
                   VALUES('Bakti','Siregar','Jl.Bahagia Selalu','Tangerang','081369','
```

2.5.4 DELETE

The DELETE statement is used to delete existing records in a table.

```
dbExecute(new_con,"DELETE FROM Customers
                   WHERE CustomerName ='Bakti' ")
```

2.5.5 UPDATE

The UPDATE statement is used to modify the existing records in a table.

```
dbExecute(new_con,"UPDATE Customers
                   SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'
                   WHERE CustomerID = 1")
```

2.5.6 Disconnect Database

If you are done with the query process and you don't want to use it anymore, you should disconnect the connection from your database.

```
dbDisconnect(new_con)                                    # disconnect from your database
```

2.6 Your Job

1. Write Dataframe to your Database directory, using the following Engine:

- RMySQL
- RPostgres
- RSQLite

2. Create a 'Basic Queries' tutorial using all engine that you already use at task no 1, (Such as: `SELECT`, `WHERE`, `INSERT INTO`, `DELETE`, and `UPDATE`)!

Chapter 3

Fundamental SQL in R

In the previous section, we learned how to connect R to a Database System (SQL) Such as RMariaDB, RMySQL, and RSQLite. In this section, we continue to cover all that you have to know about fundamental operations in SQL (Here, focus on RMySQL).

3.1 Connecting R to MySQL

Connecting R to MySQL is made very easy with the `RMySQL` package. To connect to a MySQL database simply install the package and load the library.

```
library(RMySQL)
MySQL <- dbConnect(MySQL(),
                   user='root',
                   password='',
                   dbname='mysql',
                   host='localhost')
dbListTables(MySQL)                    # a list of the tables in our connection
```

Note: Open and your XAMPP, click start on Apache and MySQL. Then, make sure you have the admin privilege before creating any database.

3.2 Create DB

If you want to create a new database, then the CREATE DATABASE statement would be as shown below:

```
dbExecute(MySQL, "CREATE DATABASE factory_db")
```

The result show us 1, means that you have succeeded to create a database.

3.3 Drop DB

If you want to delete an existing database, then the DROP DATABASE statement would be as shown below:

```
dbExecute(MySQL, "DROP DATABASE factory_db")
```

The result show us 0, means that you have succeeded to remove (Drop) a database.

3.4 Create Table

Once you have a database, you can continue to create table as shown below:

```
dbExecute(MySQL, "CREATE TABLE Persons(
                  PersonID int,
                  LastName varchar(255),
                  FirstName varchar(255),
                  Address varchar(255),
                  City varchar(255))")
```

3.4.1 Insert Value

If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. The INSERT INTO syntax would be as follows:

```
dbExecute(MySQL,"INSERT INTO Persons(PersonID,LastName,FirstName, Address,City)
                  VALUES(1,'Siregar','Bakti', 'Jl.Bahagia','Tangerang')")
```

3.4.2 Truncate Table

The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself.

```
dbExecute(MySQL, "TRUNCATE TABLE Persons")
```

3.4.3 Drop Table

The DROP TABLE statement is used to drop an existing table in a database.

```
dbExecute(MySQL, "DROP TABLE Persons")
```

3.4.4 Write Table

The key here is the `dbWriteTable` function which allows us to write an R data frame directly to a database table. The data frame's column names will be used as the database table's fields.

```
Orders      <-read.csv("data/Orders.csv")
dbWriteTable(MySQL, "Orders", Orders, append=T)
```

3.4.5 Alter Table

The ALTER TABLE statement is used to add, delete, or modify columns in an existing table. The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

3.4.6 Add Column

To add a column in a table, use the following syntax:

```
dbExecute(MySQL, "ALTER TABLE Orders
                  ADD Email varchar(255)")
```

3.4.7 Drop Column

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

```
dbSendQuery(MySQL, "ALTER TABLE Orders
                    DROP COLUMN Email")
```

3.4.8 Modify Column

```
dbSendQuery(MySQL," ALTER TABLE Orders
                  MODIFY COLUMN OrderDate date")
```

3.5 Constraints

SQL constraints are used to specify rules for the data in a table. Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table. The following constraints are commonly used in SQL:

- *NOT NULL:* Ensures that a column cannot have a NULL value
- *UNIQUE:* Ensures that all values in a column are different
- *PRIMARY KEY:* A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- *FOREIGN KEY:* Uniquely identifies a row/record in another table
- *CHECK:* Ensures that all values in a column satisfies a specific condition
- *DEFAULT:* Sets a default value for a column when no value is specified
- *INDEX:* Used to create and retrieve data from the database very quickly

3.5.1 Not Null

The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values when the "Persons_NotNull" table is created:

```
dbSendQuery(MySQL,"CREATE TABLE Person_NotNull (
                   ID int NOT NULL,
                   LastName varchar(255) NOT NULL,
                   FirstName varchar(255) NOT NULL,
                   Age int)")
```

3.5.2 Unique

The following SQL creates a UNIQUE constraint on the "ID" column when the "Persons" table is created:

```
dbSendQuery(MySQL,"CREATE TABLE Persons_Unique (ID int NOT NULL UNIQUE,
                   LastName varchar(255) NOT NULL,
                   FirstName varchar(255) NOT NULL,
                   Age int)")
```

To create a UNIQUE constraint on the "ID" column when the table is already created, use the following SQL:

```
dbSendQuery(MySQL,"ALTER TABLE Persons_Unique
                   ADD UNIQUE (ID)")
```

To define a UNIQUE constraint on multiple columns, use the following SQL syntax:

```
dbSendQuery(MySQL,"ALTER TABLE Persons_Unique
                  ADD CONSTRAINT UNIQUE (ID,LastName)"
```

To drop a UNIQUE constraint, use the following SQL:

```
dbSendQuery(MySQL,"ALTER TABLE Persons_Unique
                  DROP INDEX ID")
```

3.5.3 Primary Key

The PRIMARY KEY constraint uniquely identifies each record in a table. Primary keys must contain UNIQUE values, and cannot contain NULL values. A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).

```
dbSendQuery(MySQL,"CREATE TABLE Persons_PK (ID int NOT NULL PRIMARY KEY,
                   LastName varchar(255) NOT NULL,
                   FirstName varchar(255),
                   Age int)")
```

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

```
dbSendQuery(MySQL,"CREATE TABLE Persons_PK (ID int NOT NULL,
                  LastName varchar(255) NOT NULL,
                  FirstName varchar(255),
                  Age int,
                  CONSTRAINT Persons_PK PRIMARY KEY (ID,LastName))")
```

To create a PRIMARY KEY constraint on the "ID" column when the table is already created, use the following SQL:

```
dbSendQuery(MySQL, "ALTER TABLE Persons_PK
                    ADD PRIMARY KEY (ID)")
```

3.5.4 Foreign Key

A FOREIGN KEY is a key used to link two tables together. A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table. The table containing the foreign key is called the child table, and the table containing the candidate key is called the referenced or parent table.

Look at the following two tables:

- "Persons" table:

PersonID	LastName	FirstName	Age
1	Xi	Bakti	28
2	Li	Chong	23
3	Gou	Mei	20

- "Orders" table:

OrderID	OrderNumber	PersonID
1	77895	3
2	44678	3

Notice that the "PersonID" column in the "Orders" table points to the "PersonID" column in the "Persons" table.

- The "PersonID" column in the "Persons" table is the PRIMARY KEY in the "Persons" table.
- The "PersonID" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.
- The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.
- The FOREIGN KEY constraint also prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

```
dbSendQuery(MySQL,
"CREATE TABLE Orders (OrderID int NOT NULL,
                      OrderNumber int NOT NULL,
                      PersonID int,
                      CONSTRAINT FOREIGN KEY (PersonID))")
```

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

```
dbSendQuery(MySQL,
"CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    FOREIGN KEY (PersonID) REFERENCES Persons_pk (PersonID))")
```

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

```
dbSendQuery(MySQL,
"ALTER TABLE Orders
ADD CONSTRAINT FK_Person Order
FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)")
```

3.5.5 Check

The CHECK constraint is used to limit the value range that can be placed in a column. If you define a CHECK constraint on a single column it allows only certain values for this column. If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row. The following SQL creates a CHECK constraint on the "Age" column when the "Persons" table is created. The CHECK constraint ensures that the age of a person must be 18, or older:

```
dbSendQuery(MySQL,
"CREATE TABLE Persons (ID int NOT NULL,
                       LastName varchar(255) NOT NULL,
                       FirstName varchar(255),
                       Age int,
                       CHECK (Age>=18))")
```

To allow naming of a CHECK constraint, and for defining a CHECK constraint on multiple columns, use the following SQL syntax:

```
dbSendQuery(MySQL,
"CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255),
    CONSTRAINT CHK_Person CHECK (Age>=18 AND City='Sandnes'))")
```

To create a CHECK constraint on the "Age" column when the table is already created, use the following SQL:

```
dbSendQuery(MySQL, "ALTER TABLE Persons
                        ADD CHECK (Age>=18)")
```

To allow naming of a CHECK constraint, and for defining a CHECK constraint on multiple columns, use the following SQL syntax:

```
dbSendQuery(MySQL, "ALTER TABLE Persons
                    ADD CONSTRAINT CHK_PersonAge
                    CHECK (Age>=18 AND City='Sandnes')")
```

3.5.6 Default

The DEFAULT constraint is used to provide a default value for a column. The default value will be added to all new records IF no other value is specified. The following SQL sets a DEFAULT value for the "City" column when the "Persons" table is created:

```
dbSendQuery(MySQL,
"CREATE TABLE Persons_default (ID int NOT NULL,
                               LastName varchar(255) NOT NULL,
                               FirstName varchar(255),
                               Age int,
                               City varchar(255) DEFAULT 'Sandnes')")
```

To create a DEFAULT constraint on the "City" column when the table is already created, use the following SQL:

```
dbSendQuery(MySQL,"ALTER TABLE Persons
                        ALTER City SET DEFAULT 'Sandnes'")
```

3.5.7 Index

The CREATE INDEX statement is used to create indexes in tables. Indexes are used to retrieve data from the database more quickly than otherwise. The users cannot see the indexes, they are just used to speed up searches/queries. Creates an index on a table. Duplicate values are allowed:

```
dbSendQuery(MySQL, "CREATE INDEX idx_pname
                    ON Persons (LastName, FirstName)")
```

Note: Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

3.5.8 Auto Increment

Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table. Often this is the primary key field that we would like to be created automatically every time a new record is inserted. The following SQL statement defines the "Personid" column to be an auto-increment primary key field in the "Persons" table:

```
dbSendQuery(MySQL,
"CREATE TABLE Persons_ai (
    Personid int NOT NULL AUTO_INCREMENT,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    PRIMARY KEY (Personid))")
```

3.6 Previewing `.sql` in R

When you open a new `.sql` file in RStudio, it automatically populates the file with the following code:

```
library(RSQLite)
library(dplyr)
library(dbplyr)
```

```
conn <- src_memdb() # create a SQLite database in memory
copy_to(conn,
        storms,     # this is a dataset built into dplyr
        overwrite = TRUE)
tbl(conn, sql("SELECT * FROM storms LIMIT 5"))
```

You need to create a `.sql` file with the following code:

```
-- !preview conn=src_memdb()$con

SELECT * FROM storms LIMIT 5
```

Then, you will the resul like this:

```
library(knitr)
include_graphics("./images/Bab3/sql-file-preview.png")
```

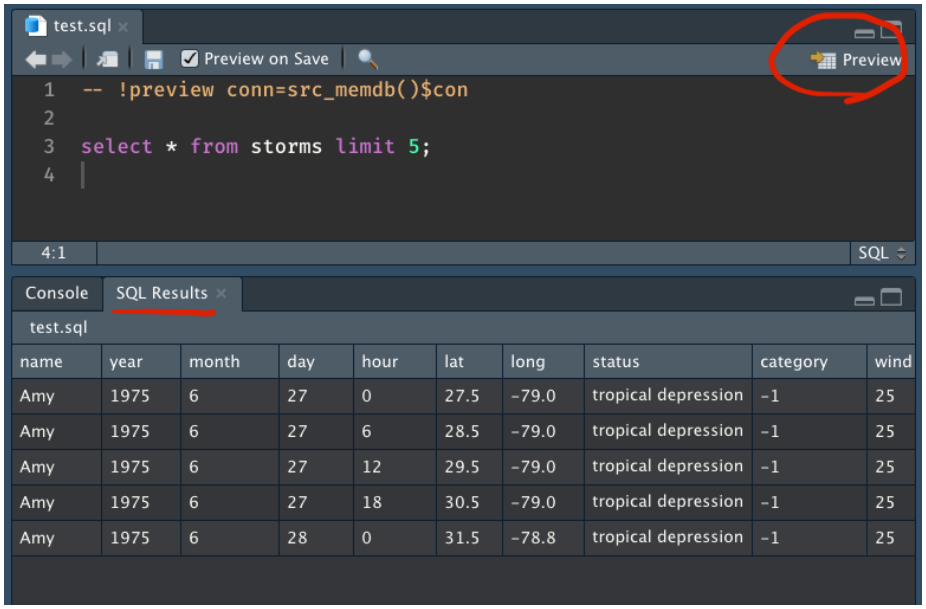

Figure 3.1: Previewing ‘.sql‘ in R

3.7 SQL chunks in RMarkdown

I generally prefer to show RMarkdown output in the console 1 (and it looks like I’m not the only one). This means that when I run code in an .Rmd file, it feels

more or less the same as when I run an .R file: the plots show up in the plots pane, code is run in the console, and so on. While you can use SQL chunks with this setting, there is NO chunk preview option. You must trust your queries and knit the file to make sure everything runs. You get the syntax highlighting razzle-dazzle but alas– no preview.

It is in this very specific case where inline mode wins big time. SQL previews magically become an option, allowing you to interact with your beautifully colored SQL code.

Chapter 4

Database Normalization in SQL

Normalization is a database design technique that reduces data redundancy and eliminates undesirable characteristics like Insertion, Update and Deletion Anomalies. Normalization rules divides larger tables into smaller tables and links them using relationships. The purpose of Normalisation in SQL is to eliminate redundant (repetitive) data and ensure data is stored logically.

The inventor of the relational model Edgar Codd proposed the theory of normalization of data with the introduction of the First Normal Form, and he continued to extend theory with Second and Third Normal Form. Later he joined Raymond F. Boyce to develop the theory of Boyce-Codd Normal Form.

Here is a list of Normal Forms in SQL:

- 1NF (First Normal Form)
- 2NF (Second Normal Form)
- 3NF (Third Normal Form)
- BCNF (Boyce-Codd Normal Form)
- 4NF (Fourth Normal Form)
- 5NF (Fifth Normal Form)
- 6NF (Sixth Normal Form)

The Theory of Data Normalization in MySQL server is still being developed further. For example, there are discussions even on 6th Normal Form. However, in most practical applications, normalization achieves its best in 3rd Normal Form.

4.1 The Process of Normalization

The process of normalization involves breaking down a large table into smaller, related tables and defining relationships between them. This helps in achieving the following benefits:

- **Elimination of Data Redundancy:** Redundant data can lead to inconsistencies and increased storage requirements. Normalization ensures that each piece of data is stored in only one place, reducing redundancy and promoting consistency.
- **Data Integrity:** Normalization minimizes the chances of inconsistencies and anomalies that may occur when data is duplicated or updated in one place but not in another.
- **Efficient Data Updates:** Since data is stored in smaller, more specific tables, updates are more efficient and require fewer changes.
- **Simpler Queries:** Normalized data allows for more straightforward and efficient querying due to the structured relationships between tables.

The process of database normalization is typically divided into several "normal forms" (often referred to as 1NF, 2NF, 3NF, BCNF, etc.), each with its own set of rules and requirements. These normal forms build on each other, with higher normal forms addressing more complex issues of redundancy and dependency. Here's a brief overview of some common normal forms:

1. **First Normal Form (1NF):**

- Eliminate duplicate columns.
- Create separate tables for related data.
- Define a primary key for each table.

2. **Second Normal Form (2NF):**

- Meet 1NF requirements.
- Remove partial dependencies (attributes dependent on only part of the primary key) by creating
- separate tables.

3. **Third Normal Form (3NF):**

- Meet 2NF requirements.
- Remove transitive dependencies (attributes dependent on non-key attributes) by creating separate tables.

4. **Boyce-Codd Normal Form (BCNF):**

- Meet 3NF requirements.
- Remove overlapping candidate keys by creating separate tables.

Higher normal forms exist beyond these, such as **Fourth Normal Form (4NF)** and **Fifth Normal Form (5NF)**, but they are less commonly encountered and may be more relevant in specific cases of complex data modeling. While normalization offers significant benefits, it's important to strike a balance between normalization and performance. Over-normalization can lead to complex query logic and decreased query performance. Therefore, designing a database often involves considering the nature of the data and the queries that will be performed on it.

4.2 Simple Database Normalization

Let's go through a simple example of database normalization using a hypothetical scenario of an online bookstore. We'll start with an unnormalized table and then progressively normalize it through different normal forms.

Scenario: Consider an unnormalized table that stores information about books, authors, and their publishers.

Unnormalized Table (1NF):

Book ID	Title	Author	Author Birth	Publisher	Year
1	Algorithm	John Smith	1980-05-15	ABC Pub	2000
2	Data Science	Jane Doe	1975-10-20	XYZ Books	2015
3	Database System	John Smith	1980-05-15	ABC Pub	2012

In this unnormalized table, we have duplicate author and publisher information, leading to redundancy. John Smith's information is repeated, and if any of his details change, we need to update multiple rows.

First Normal Form (1NF):

To achieve 1NF, we break the table into smaller tables and remove duplicate data. We create separate tables for authors and publishers.

Authors Table:

Author ID	Author	Author Birth
1	John Smith	1980-05-15
2	Jane Doe	1975-10-20

Publishers Table:

Publisher ID	Publisher
1	ABC Pub
2	XYZ Books

Books Table (1NF):

Book ID	Title	Author ID	Publisher ID	Year
1	Algorithm	1	1	2000
2	Data Science	2	2	2015
3	Database System	1	1	2012

We've eliminated redundancy by referencing author and publisher IDs in the books table.

Second Normal Form (2NF):

To achieve 2NF, we identify partial dependencies and create a separate table for author information.

Authors Table (2NF):

Author ID	Author	Author Birth
1	John Smith	1980-05-15
2	Jane Doe	1975-10-20

Books Table (2NF):

Book ID	Title	Author ID	Publisher ID	Year
1	Algorithm	1	1	2000
2	Data Science	2	2	2015
3	Database System	1	1	2012

No changes are required in the Books table for 2NF since there were no partial dependencies.

Third Normal Form (3NF):

To achieve 3NF, we identify transitive dependencies and create a separate table for publisher information.

Publishers Table (3NF):

Table 4.9: First Normal Form (1NF)

BookID	Title	Author	Genre	Publisher	PublicationYear
1	Book A	Author X	Fiction	dsciencelabs	2021
2	Book B	Author Y	Mystery	Matana	2022
3	Book B	Author X	Romance	dsciencelabs	2023

Publisher ID	Publisher
1	ABC Pub
2	XYZ Books

Books Table (3NF):

Book ID	Title	Author ID	Publisher ID	Year
1	Algorithm	1	1	2000
2	Data Science	2	2	2015
3	Database System	1	1	2012

No changes are required in the Books table for 3NF since there were no transitive dependencies. The result is a normalized database structure that eliminates redundancy and ensures data integrity.

Please note that the above example is simplified for demonstration purposes. In real-world scenarios, databases can have more complex structures and relationships, which may require deeper levels of normalization to achieve higher normal forms like BCNF or 4NF.

4.3 Your Job

Consider a hypothetical database for an online bookstore. We'll start with a denormalized table and then go through the normalization process step by step.

Suppose we have a single table called Books with the following columns:

Your job is the following statements:

1. Display Database Normalization Process
2. Create Database to your PC After Normalization Process using R and SQL

Chapter 5

Join Table in SQL

A SQL join is a Structured Query Language (SQL) instruction to combine data from two sets of data (i.e. two tables). Before we dive into the details of a SQL join, let's briefly discuss what SQL is, and why someone would want to perform a SQL join.

SQL is a special-purpose programming language designed for managing information in a relational database management system (RDBMS). The word relational here is key; it specifies that the database management system is organized in such a way that there are clear relations defined between different sets of data. Typically, you need to extract, transform, and load data into your RDBMS before you're able to manage it using SQL, which you can accomplish by using a tool like Stitch.

5.1 Relational Database

Imagine you're running a store and would like to record information about your customers and their orders. By using a relational database, you can save this information as two tables that represent two distinct entities: customers and orders .

5.1.1 Table Customers

```
library(DT)
costomers<-read.csv("data/customers.csv")
```

```
datatable(head(costomers, 5),
          caption = htmltools::tags$caption(
            style = 'caption-side: bottom; text-align: center;',
            htmltools::em('Table 1: customers.')),
          options = list(dom = 't'))
```

Table 1, informs about each customer is stored in its own row, with columns specifying different bits of information, including their first name, last name, and email address. Additionally, we associate a unique customer number, or primary key, with each customer record.

5.1.2 Table Orders

```
orders<-read.csv("data/orders.csv")
datatable(head(orders,5),
          caption = htmltools::tags$caption(
            style = 'caption-side: bottom; text-align: center;',
            htmltools::em('Table 2: orders.')),
          options = list(dom = 't'))
```

Again, Table 2 are contains information about a specific order. Each order has its own unique identification key `order_id` for this table – assigned to it as well.

5.2 Relational Model

You've probably noticed that these two examples share similar information. You can see these simple relations diagrammed below:

Note that the orders table contains two keys: one for the order and one for the customer who placed that order. In scenarios when there are multiple keys in a table, the key that refers to the entity being described in that table is called the primary key *(PK)* and other key is called a foreign key *(FK)*.

In our example, `order_id` is a primary key in the orders table, while `customer_id` is both a primary key in the customers table and a foreign key in the orders table. Primary and foreign keys are essential to describing relations between the tables, and in performing SQL joins.

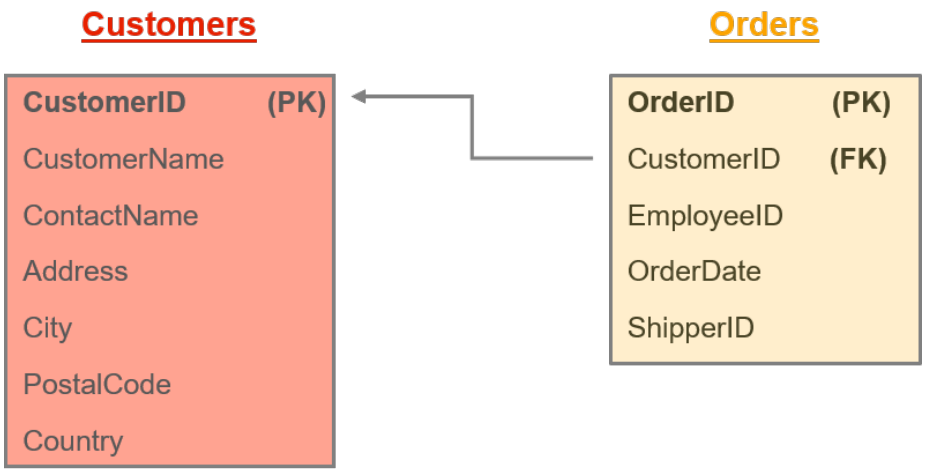

Figure 5.1: Relational Table

5.3 Factory Database

To make you more convenient about all the data tables that we will use in this section. Here, I summarize the following SQL relational for database `factory_db`:

Note: Don't forget to consider the data structure of your database (all table)

5.4 Basic SQL Join Types

There are four basic types of SQL joins: inner, left, right, and full. The easiest and most intuitive way to explain the difference between these four types is by using a Venn diagram, which shows all possible logical relations between data sets.

Again, it's important to stress that before you can begin using any join type, you'll need to extract the data and load it into an RDBMS like Amazon Redshift, where you can query tables from multiple sources. You build that process manually, or you can use an ETL service like Stitch, which automates that process for you.

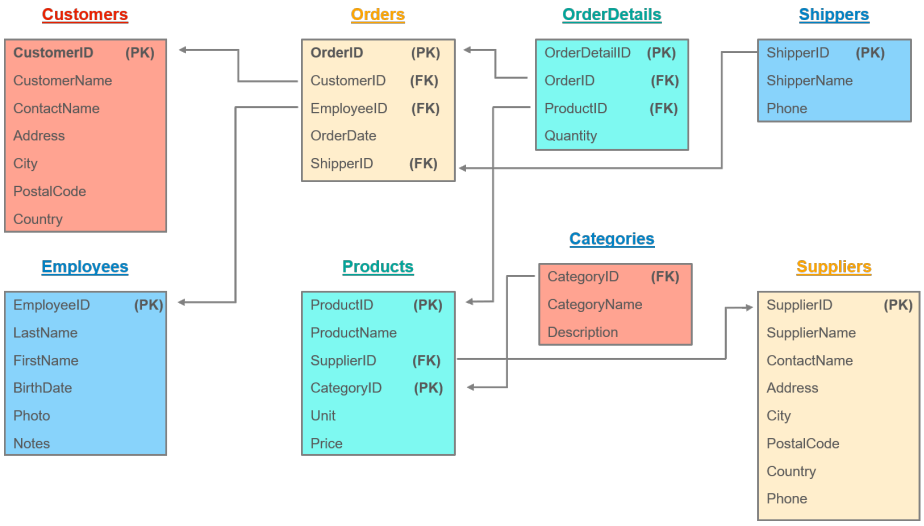

Figure 5.2: Relational Table of Factory Database

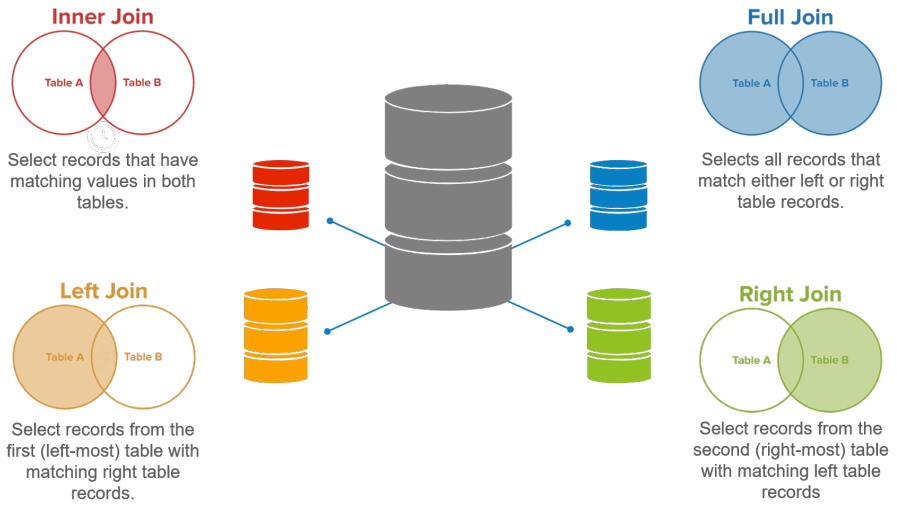

Figure 5.3: Basic Join Table

5.5 Connect to MySQL

Reading data from `MySQL` into R workspace, it requires two R libraries, `RMySQL` and `DBI`. The connection data should not be embedded in analysis code. Separate the connection code in another script. The script should set up the connection and save it into the workspace.

The saved connection is accessible by its name in the analysis code. In the dbConnect function, you need to replace dbname, username, pwd, dbserver and port with the actual values of your remote database.

```
# set up the connection and save it into the workspace
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
library(RMySQL)
library(DBI)
bakti <- dbConnect(RMySQL::MySQL(),
                   dbname='factory_db',
                   username='root',
                   password='',
                   host='localhost',
                   port=3306)
knitr::opts_chunk$set(connection = "bakti")   # set up the connection
```

After set up the connection and save it into the workspace. Next, we can run SQL in a code chunk of type sql. By setting the connection in the code chuck and adding the option output.var, the resulting table from the SQL is written into a variable in R.

```
'''{sql connection=bakti, output.var="report_model_by_make"}
      Your SQL code Here
'''
```

5.6 Inner Join

Let's say we wanted to get a list of those customers who placed an order and the details of the order they placed. This would be a perfect fit for an inner join, since an inner join returns records at the intersection of the two tables.

```
SELECT OrderID, CustomerName
  FROM Orders O
    INNER JOIN Customers C
      ON O.CustomerID = C.CustomerID
```

```
library(DT)
datatable(Inner1,
          caption = htmltools::tags$caption(
            style = 'caption-side: bottom; text-align: center;',
            htmltools::em('Table 3: SQL Inner Join Two Tables.')))
```

The following SQL statement selects all orders with customer and shipper information:

```
SELECT *
  FROM ((Orders O
    INNER JOIN Customers C
      ON O.CustomerID = C.CustomerID)
    INNER JOIN Shippers S
      ON O.ShipperID = S.ShipperID)
```

```
datatable(Inner2,
          caption = htmltools::tags$caption(
            style = 'caption-side: bottom; text-align: center;',
            htmltools::em('Table 4: SQL Inner Join Three Tables.')),
          extensions = 'FixedColumns',
          options = list(scrollX = TRUE, fixedColumns = TRUE)
          )
```

5.7 Left Join

If we wanted to simply append information about orders to our customers table, regardless of whether a customer placed an order or not, we would use a left join. A left join returns all records from table A and any matching records from table B. The result is NULL from the right side, if there is no match.

```
SELECT CustomerName, OrderID
  FROM Customers C
    LEFT JOIN Orders O
      ON C.CustomerID = O.CustomerID
        ORDER BY C.CustomerName
```

```
datatable(Left,
          caption = htmltools::tags$caption(
            style = 'caption-side: bottom; text-align: center;',
            htmltools::em('Table 5: SQL Left Join Two Tables.')))
```

5.8 Right Join

The following SQL statement will return all employees, and any orders they might have placed. The result is NULL from the left side, when there is no match.

```
SELECT OrderID, LastName, FirstName
  FROM Orders O
    RIGHT JOIN Employees E
      ON O.EmployeeID = E.EmployeeID
        ORDER BY O.OrderID
```

```
datatable(Right,
          caption = htmltools::tags$caption(
            style = 'caption-side: bottom; text-align: center;',
            htmltools::em('Table 6: SQL Right Join Two Tables.')))
```

5.9 Full Join

The FULL OUTER JOIN keyword returns all records when there is a match in left (table1) or right (table2) table records. FULL OUTER JOIN, FULL JOIN, and JOIN (MariaDB) are the same. The following SQL statement selects all customers, and all orders:

```
SELECT CustomerName, OrderID
  FROM Customers C
    JOIN Orders O
      ON C.CustomerID=O.CustomerID
        ORDER BY C.CustomerName
```

```
datatable(Full,
          caption = htmltools::tags$caption(
            style = 'caption-side: bottom; text-align: center;',
            htmltools::em('Table 7: SQL Full Join Two Tables.')))
```

5.10 Self JOIN

A self JOIN is a regular join, but the table is joined with itself. The following SQL statement matches customers that are from the same city:

```
SELECT A.CustomerName AS CustomerName1,
       B.CustomerName AS CustomerName2,
       A.City
  FROM Customers A,
       Customers B
    WHERE A.CustomerID <> B.CustomerID
      AND A.City = B.City
        ORDER BY A.City
```

```
datatable(Self,
          caption = htmltools::tags$caption(
            style = 'caption-side: bottom; text-align: center;',
            htmltools::em('Table 8: SQL Self Join Two Tables.')))
```

After finishing the work with the database, close the connection.

```
DBI::dbDisconnect(bakti)
```

5.11 Your Job

1. Apply Left join and Right join to returns all records from table Orders and any matching records from table Suppliers.
2. Choose the correct JOIN clause to select all records from the two tables (Orders and Suppliers) where there is a match in both tables.
3. Choose the correct JOIN clause to select all the records from the Suppliers table plus all the matches in the Orders table.

Chapter 6

Simple Query

The real power of a relational database lies in its ability to quickly retrieve and analyze your data by running a query. Queries allow you to pull information from one or more tables based on a set of search conditions you define. In this section, you will learn how to create a simple one-table query.

First, we need to connect to our database. Please type the following code in your R console:

```
# set up the connection and save it into the workspace
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
library(RMySQL)
library(DBI)
bakti <- dbConnect(RMySQL::MySQL(),
                   dbname='factory_db',
                   username='root',
                   password='',
                   host='localhost',
                   port=3306)
knitr::opts_chunk$set(connection = "bakti") # to set up the connection in your Rmarkdown chunk
```

6.1 SELECT

The SQL SELECT statement is used to fetch the data from a database table which returns this data in the form of a result table. These result tables are called result-sets. The basic syntax of the SELECT statement is as follows:

```
SELECT column1, column2, columnN
  FROM table_name;
```

Assume, column1, column2... are the fields of a table whose values you want to fetch. If you want to fetch some of the fields available in the field, then you can use the following syntax.

```
SELECT CustomerName, Address, City, Country
  FROM CUSTOMERS;
```

If you want to fetch all the fields of the CUSTOMERS table, then you should use the following query.

```
SELECT *
  FROM CUSTOMERS;
```

6.2 DISTINCT

The SQL DISTINCT keyword is used in conjunction with the SELECT statement to eliminate all the duplicate records and fetching only unique records. There may be a situation when you have multiple duplicate records in a table. While fetching such records, it makes more sense to fetch only those unique records instead of fetching duplicate records.

The basic syntax of DISTINCT keyword to eliminate the duplicate records is as follows:

```
SELECT DISTINCT column_name
  FROM table_name
```

Now, let us use the DISTINCT keyword with the above SELECT query and then see the result.

```
SELECT DISTINCT Country
  FROM customers;
```

6.3 WHERE

The WHERE clause is used to filter records. The WHERE clause is used to extract only those records that fulfill a specified condition.

```
SELECT column1, column2, ...
  FROM table_name
    WHERE [condition];
```

Note: The WHERE clause is not only used in SELECT statement, it is also used in UPDATE, DELETE statement, etc.!

The following SQL statement selects all the customers from the country "Mexico", in the "Customers" table:

```
SELECT *
  FROM Customers
    WHERE Country='Mexico';
```

The following operators can be used in the WHERE clause, please try it by your self!

Operator	Description
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<>	Not equal. Note: In some versions of SQL this operator may be written as !=
IS NULL or IS NOT NUL	A field with a NULL value is a field with no value.
BETWEEN	Between a certain range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

6.4 BETWEEN

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates. The BETWEEN operator is inclusive: begin and end values are included.

```
SELECT column_name(s)
  FROM table_name
    WHERE column_name
      BETWEEN value1 AND value2;
```

The following SQL statement selects all products with a price BETWEEN 10 and 20:

```
SELECT *
```

```
  FROM Products
    WHERE Price
      BETWEEN 10 AND 20;
```

The following SQL statement selects all orders with an OrderDate BETWEEN '01-July-1996' and '31-July-1996':

```
SELECT *
  FROM Orders
    WHERE OrderDate
      BETWEEN '1996-07-01' AND '1996-07-31';
```

6.5 IN

The IN operator allows you to specify multiple values in a WHERE clause. The IN operator is a shorthand for multiple OR conditions.

```
SELECT column_name(s)
  FROM table_name
    WHERE column_name
      IN (SELECT STATEMENT);
```

The following SQL statement selects all customers that are located in "Germany", "France" or "UK":

```
SELECT *
  FROM Customers
    WHERE Country
      IN ('Germany', 'France', 'UK');
```

The following SQL statement selects all customers that are from the same countries as the suppliers:

```
SELECT *
  FROM Customers
    WHERE Country
      IN (SELECT Country FROM Suppliers);
```

6.6 LIKE

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column. There are two wildcards often used in conjunction with the LIKE operator:

```
SELECT column1, column2, ...
  FROM table_name
    WHERE columnN LIKE pattern;
```

- % : The percent sign represents zero, one, or multiple characters
- _ : The underscore represents a single character

The following SQL statement selects all customers with a CustomerName starting with "a":

```
SELECT *
  FROM Customers
    WHERE CustomerName
      LIKE 'a%';
```

Here are some examples showing different LIKE operators with '%' and '_' wildcards:

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that start with "a"
WHERE CustomerName LIKE '%a'	Finds any values that end with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%'	Finds any values that start with "a" and are at least 2 characters in length
WHERE CustomerName LIKE 'a___%'	Finds any values that start with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	

6.7 AND, OR and NOT

The WHERE clause can be combined with AND, OR, and NOT operators. The AND and OR operators are used to filter records based on more than one condition:

- The AND operator displays a record if all the conditions separated by AND are TRUE.
- The OR operator displays a record if any of the conditions separated by OR is TRUE.
- The NOT operator displays a record if the condition(s) is NOT TRUE.

```
SELECT column1, column2, ...
  FROM table_name
    WHERE condition1 AND condition2 OR condition3 NOT condition4;
```

The following SQL statement selects all fields from "Customers" where country is "Germany" AND city must be "Berlin" OR "München" (use parenthesis to form complex expressions):

```
SELECT * FROM Customers
WHERE Country='Germany' AND (City='Berlin' OR City='München');
```

Let see one more example, the following SQL statement selects all fields from "Customers" where country is NOT "Germany" and NOT "USA":

```
SELECT * FROM Customers
WHERE Country='Germany' AND (City='Berlin' OR City='München');
```

6.8 ORDER BY

The SQL ORDER BY clause is used to sort the data in ascending or descending order, based on one or more columns. Some databases sort the query results in an ascending order by default. The basic syntax of the ORDER BY clause is as follows:

```
SELECT column-list
  FROM table_name
    [WHERE condition]
      [ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

- By default ORDER BY sorts the data in ascending order.
- We can use the keyword DESC to sort the data in descending order and the keyword ASC to sort in ascending order.

You can use more than one column in the ORDER BY clause. Make sure whatever column you are using to sort that column should be in the column-list.The following code block has an example, which would sort the result in an ascending order by the City and the Country:

```
SELECT * FROM Customers
WHERE Country='Germany' AND (City='Berlin' OR City='München')
  ORDER BY Country, City;
```

6.9 LIMIT

If there are a large number of tuples satisfying the query conditions, it might be resourceful to view only a handful of them at a time.

- The LIMIT clause is used to set an upper limit on the number of tuples returned by SQL.
- It is important to note that this clause is not supported by all SQL versions.
- The LIMIT clause can also be specified using the SQL 2008 OFFSET/FETCH FIRST clauses.
- The limit/offset expressions must be a non-negative integer.

```
SELECT column-list
  FROM table_name
    [WHERE condition]
      [ORDER BY column1, column2, .. columnN] [ASC | DESC]
        LIMIT rows_to_skip, next_rows_to_skip;
```

The following illustrates the LIMIT clauses to collect TOP 3 rows:

```
SELECT * FROM Customers
WHERE Country='Germany' AND (City='Berlin' OR City='München')
  ORDER BY Country, City
      LIMIT 3;
```

Next, the following illustrates the LIMIT clauses to collect TOP 5 rows after TOP 3 rows:

```
SELECT CustomerName, Address, City, Country
  FROM customers
    ORDER BY City, Country DESC
      LIMIT 3, 5;
```

6.10 MIN and MAX

The MIN() function returns the smallest value of the selected column. The MAX() function returns the largest value of the selected column.

```
SELECT MIN/MAX(column_name)
  FROM table_name
    WHERE condition;
```

The following SQL statement finds the price of the cheapest product:

```
SELECT MIN(Price) AS SmallestPrice
  FROM Products;
```

The following SQL statement finds the price of the most expensive product:

```
SELECT MAX(Price) AS LargestPrice
  FROM Products;
```

6.11 COUNT, SUM, and AVG

The COUNT() function returns the number of rows that matches a specified criterion. The AVG() function returns the average value of a numeric column. The SUM() function returns the total sum of a numeric column.

```
SELECT COUNT/SUM/AVG(column_name)
FROM table_name
WHERE condition;
```

The following SQL statement finds the average price of all products:

```
SELECT AVG(Price)
  FROM Products;
```

Note: Please try other functions, to get more convenient with SQL!

6.12 HAVING

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

```
SELECT column_name(s)
  FROM table_name
    WHERE condition
      GROUP BY column_name(s)
        HAVING condition
          ORDER BY column_name(s);
```

The following SQL statement lists the number of customers in each country, sorted high to low (Only include countries with more than 5 customers):

```
SELECT COUNT(CustomerID), Country
  FROM Customers
    GROUP BY Country
      HAVING COUNT(CustomerID) > 5
        ORDER BY COUNT(CustomerID) DESC;
```

6.13 CASE

The CASE statement goes through conditions and returns a value when the first condition is met (like an IF-THEN-ELSE statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the ELSE clause.

If there is no ELSE part and no conditions are true, it returns NULL.

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    WHEN conditionN THEN resultN
    ELSE result
END;
```

The following SQL goes through conditions and returns a value when the first condition is met:

```
SELECT OrderID, Quantity,
CASE
    WHEN Quantity > 30 THEN 'The quantity is greater than 30'
```

```
    WHEN Quantity = 30 THEN 'The quantity is 30'
    ELSE 'The quantity is under 30'
END AS QuantityText
FROM OrderDetails;
```

The following SQL will order the customers by City. However, if City is NULL, then order by Country:

```
SELECT CustomerName, City, Country
FROM Customers
ORDER BY
(CASE
    WHEN City IS NULL THEN Country
    ELSE City
END);
```

6.14 Your Job

- Select Some attributes of suppliers in alphabetical order!
- Some attributes of suppliers in reverse alphabetical order!
- Some attributes of suppliers ordered by country, then by city!
- All atributes of suppliers and reverse alphabetical ordered by country, then by city!
- All orders, sorted by total amount, the largest first!
- Get all but the 10 most expensive products sorted by price!
- Get the 10th to 15th most expensive products sorted by price!
- List all supplier countries in alphabetical order!
- Find the cheapest product and Expensive Orders!
- Find the number of Supplier USA!
- Compute the total Quantity of orderitem!
- Compute the average UnitPrice of all product!
- Get all information about customer named Thomas Hardy!
- List all customers from Spain or France!
- List all customers that are not from the USA!
- List all orders that not between $50 and $15000!
- List all products between $10 and $20
- List all products not between $10 and $100 sorted by price!
- Get the list of orders and amount sold between 1996 Jan 01 and 1996 Des 31!
- List all suppliers from the USA, UK, OR Japan!
- List all products that are not exactly $10, $20, $30, $40, or $50!
- List all customers that are from the same countries as the suppliers!
- List all products that start with ‘Cha’ or ‘Chan’ and have one more character!

- List all suppliers that do have a fax number!
- List all customer with average orders between $1000 and $1200 !
- List total customers in each country.
- Display results with easy to understand column headers.
- Measure the average order of product names from each country and order it from max to min.
- Compare the average order of product names from each country in the year 1996 vs 1997 order it from max to min.

Chapter 7

Join Table Queries

In the previous section, you learned how to create a simple query with one table. Most queries you design in Access will likely use multiple tables, allowing you to answer more complex questions. In this lesson, you'll learn how to design and create a multi-table query.

Queries can be difficult to understand and build if you don't have a good idea of what you're trying to find and how to find it. A one-table query can be simple enough to make up as you go along, but to build anything more powerful you'll need to plan the query in advance.

7.1 Planning a Query in SQL

Planning a query in SQL involves several important steps to ensure that you retrieve the desired data efficiently and accurately. Here's a step-by-step guide to planning and executing a successful SQL query:

- **Understand Requirements**

  Clearly define the purpose of your query. Understand what specific data you need and what conditions or criteria need to be met. If you're unsure, discuss the requirements with stakeholders or team members.

- **Select the Right Table(s)**

  Identify the table(s) that contain the relevant data you need. Make sure you understand the table structure, column names, and relationships between tables (if applicable).

- **Choose Columns**

  Determine the columns you need in the query result. Select only the columns that are necessary to fulfill the query requirements. This reduces the amount of data retrieved and improves performance.

- **Apply Filters (WHERE Clause)**

  Use the WHERE clause to filter the rows that meet specific criteria. This helps narrow down the dataset and retrieves only the relevant records. Be cautious not to use overly complex conditions that might slow down the query.

- **Sort Results (ORDER BY Clause, if needed)**

  If you need the results in a specific order, use the ORDER BY clause to sort the output based on one or more columns. Sorting can impact performance, so use it judiciously.

- **Aggregate Data (GROUP BY and HAVING Clauses, if needed)**

  If you need to perform aggregate calculations (e.g., SUM, AVG, COUNT), use the GROUP BY clause to group data based on certain columns. You can also use the HAVING clause to filter groups based on aggregate conditions.

- **Join Tables (if needed)**

  If your query requires data from multiple tables, use the appropriate join operations (INNER JOIN, LEFT JOIN, etc.) to combine data based on related columns. Make sure you understand the relationships and select the appropriate join type.

- **Optimize Performance**

  Consider the performance implications of your query. Avoid using unnecessary subqueries or functions that could slow down execution. Use indexes on columns that are frequently used for filtering or joining.

- **Test the Query**

Before executing the query in a production environment, test it in a safe environment (e.g., a development or testing database). Verify that the query returns the expected results and that the performance is acceptable.

- **Backup Data (if applicable)**

  If your query involves updating or deleting data, create a backup of the relevant tables before making any changes. This helps prevent accidental data loss.

- **Execute and Review**

  Once you're confident in your query, execute it in the production environment if necessary. Review the results to ensure they match your expectations.

- **Monitor and Optimize**

  After executing the query, monitor its performance in the production environment. Use tools like query execution plans to identify bottlenecks and optimize as needed.

- **Document the Query**

  Document the query, including its purpose, the tables involved, the filters applied, and any other relevant details. This documentation can be helpful for future reference and troubleshooting.

By following these steps, you can plan and execute SQL queries effectively, ensuring that you retrieve accurate results in an efficient manner.

7.2 UNION

The UNION operator is used to combine the result-set of two or more SELECT statements.

- Each SELECT statement within UNION must have the same number of columns
- The columns must also have similar data types
- The columns in each SELECT statement must also be in the same order

```
SELECT column_name(s)
  FROM table1
    UNION
      SELECT column_name(s)
        FROM table2;
```

Before we begin, first we need to connect to our database. Please type the following code in your R console:

```
# set up the connection and save it into the workspace
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
library(RMySQL)
library(DBI)
bakti <- dbConnect(RMySQL::MySQL(),
                   dbname='factory_db',
                   username='root',
                   password='',
                   host='localhost',
                   port=3306)
knitr::opts_chunk$set(connection = "bakti") # to set up the connection in your Rmarkdo
```

The following SQL statement returns the cities (only distinct values) from both the "Customers" and the "Suppliers" table:

```
SELECT City
  FROM Customers
    UNION
      SELECT City
        FROM Suppliers
          ORDER BY City;
```

Note: If some customers or suppliers have the same city, each city will only be listed once, because UNION selects only distinct values. Use UNION ALL to also select duplicate values!

7.2.1 UNION ALL

The following SQL statement returns the cities (duplicate values also) from both the "Customers" and the "Suppliers" table:

```
SELECT City
  FROM Customers
    UNION ALL
      SELECT City
        FROM Suppliers
          ORDER BY City;
```

7.2.2 UNION With WHERE

The following SQL statement returns the German cities (only distinct values) from both the "Customers" and the "Suppliers" table:

```
SELECT City, Country
  FROM Customers
    WHERE Country='Germany'
      UNION
        SELECT City, Country
          FROM Suppliers
            WHERE Country='Germany'
              ORDER BY City;
```

7.2.3 UNION ALL With WHERE

The following SQL statement returns the German cities (duplicate values also) from both the "Customers" and the "Suppliers" table:

```
SELECT City, Country
  FROM Customers
    WHERE Country='Germany'
      UNION ALL
        SELECT City, Country
          FROM Suppliers
            WHERE Country='Germany'
              ORDER BY City;
```

7.3 EXISTS

The EXISTS operator is used to test for the existence of any record in a subquery. The EXISTS operator returns true if the subquery returns one or more records.

```
SELECT column_name(s)
  FROM table_name
    WHERE EXISTS
      (SELECT column_name FROM table_name WHERE condition);
```

The following SQL statement returns TRUE and lists the suppliers with a product price over $50

```
SELECT SupplierName
  FROM Suppliers
    WHERE EXISTS (SELECT ProductName
                    FROM Products
                      WHERE Products.SupplierID = Suppliers.supplierID
                        AND Price > 50);
```

7.4 ANY and ALL

The ANY and ALL operators are used with a WHERE or HAVING clause. The ANY operator returns TRUE if any of the subquery values meet the condition.

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ANY
(SELECT column_name FROM table_name WHERE condition);
```

The following SQL statement returns TRUE and lists the product names if it finds ANY records in the OrderDetails table that quantity = 10:

```
SELECT *
  FROM Products
    WHERE ProductID = ANY (SELECT ProductID
                             FROM OrderDetails
                               WHERE Quantity = 10);
```

The ALL operator returns TRUE if all of the subquery values meet the condition. The following SQL statement returns TRUE and lists the product names if ALL the records in the OrderDetails table has quantity = 11.

```
SELECT ProductName
  FROM Products
    WHERE ProductID = ALL (SELECT ProductID
                             FROM OrderDetails
                               WHERE Quantity = 11);
```

7.5 GROUP BY

The GROUP BY statement groups rows that have the same values into summary rows, like “find the number of customers in each country”. The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

```
SELECT column_name(s)
  FROM table_name
    WHERE condition
      GROUP BY column_name(s)
        ORDER BY column_name(s);
```

The following SQL statement lists the number of orders sent by each shipper:

```
SELECT Shippers.ShipperName, COUNT(Orders.OrderID) AS NumberOfOrders
  FROM Orders
    LEFT JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID
      GROUP BY ShipperName;
```

7.6 HAVING

The following SQL statement lists the employees that have registered more than 10 orders:

```
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders
  FROM (Orders
    INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID)
      GROUP BY LastName
        HAVING COUNT(Orders.OrderID) > 10;
```

The following SQL statement lists if the employees "Davolio" or "Fuller" have registered more than 25 orders:

```
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders
  FROM Orders
    INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
      WHERE LastName = 'Davolio' OR LastName = 'Fuller'
        GROUP BY LastName
          HAVING COUNT(Orders.OrderID) > 25;
```

Note: All the functions that we can use in simple queries, it can definitely use them in Multi-table query.

7.7 Your Job

- List all orders with customer information!
- List all orders with product names, quantities, and prices!

- This will list all customers, whether they placed any order or not!
- List customers that have not placed orders!
- List all contacts, i.e., suppliers and customers!
- List products with order quantities greater than 80!
- Which products were sold by the unit (i.e. quantity =1)?
- List customers who placed orders that are larger than the average of each customer order!
- Find best selling products based on quantity!
- Find best selling products based on revenue!
- Find best selling products based on revenue for each country!
- Find suppliers with a product price less than $50!
- Find top 10 best employees based on their sales quantity!
- Find top 10 best supplier countries based on quantity!
- Find top 10 best customer countries based on quantity!
- Find top 10 best selling products based on quantity in every year!

Chapter 8

Introduction to Flexdashboard

Work-in-Progress [https://pkgs.rstudio.com/flexdashboard/articles/flexdashboard.html]

Chapter 9

Flexdasboard with SQLite

Work-in-Progress [https://pkgs.rstudio.com/flexdashboard/articles/flexdashboard.html]

Chapter 10

Shiny Dashboard

Work-in-Progress [https://rstudio.github.io/shinydashboard/index.html]

10.1 Basic Shiny Dashboard

10.2 Shiny Dashboard Plus

Chapter 11

Shiny Dashboard with SQL

Work-in-Progress [https://rstudio.github.io/shinydashboard/index.html]

11.1 Basic Shiny Dashboard

11.2 Shiny Dashboard Plus

Chapter 12

Data Analytics Dashboard

Work-in-Progress

Chapter 13

Referensi