Contoh dan Penjelasan
BAHASA SINGKONG
Bekerja dengan Database Relasional

Dilengkapi Pengantar:
Database Relasional
Interaksi Manusia dan Komputer

```
load_module("db_util")

reset()
var t = component("table", "A,B", true)
add(t)

var d = db_connect_embed("test")
if (d != null) {
    db_create_table_embed(d, "test",
        [["a", "integer."], ["b", "varchar."]]
    )

    db_insert(d, "test",
        {"a": random(0,100), "b": "hello"}
    )
    db_update(d, "test",
        [["b = ", "hello", ""]],
        {"b": "Hello World"}
    )

    var r = db_select_all(d, "test")
    if (!empty(r)) {
        config(t, "contents", r[0])
    }
}

show()
```

Dr. Noprianto
Dr. Maria Seraphina Astriani
Dr. Fredy Purnomo

Penerbit PT. Stabil Standar Sinergi

Contoh dan Penjelasan Bahasa Singkong: Bekerja dengan Database Relasional

Penulis: Dr. Noprianto, Dr. Maria Seraphina Astriani, Dr. Fredy Purnomo

ISBN: 978-602-52770-5-4

Penerbit:

PT. Stabil Standar Sinergi

Alamat	Puri Indah Financial Tower Lantai 6, Unit 0612 Jl. Puri Lingkar Dalam Blok T8, Puri Indah, Kembangan, Jakarta Barat 11610
Website	www.singkong.dev
Email	info@singkong.dev

Cover buku:

- Masakan: singkong goreng oleh Meike Thedy, S.Kom.
- Desain cover oleh Noprianto, menggunakan GIMP. Filter yang digunakan adalah: Spiral. Font yang digunakan adalah Calibri. Masing-masing teks tulisan (kecuali tulisan Penerbit) pada cover dibuat dengan beberapa layer untuk mendapatkan efek outline.
- Cuplikan kode adalah bagian dari contoh dalam buku.

Daftar Isi

Kata Pengantar

Buku ini berisi sejumlah contoh source code dan penjelasan langkah demi langkah yang mudah dipahami untuk membuat program komputer yang terhubung ke sistem database relasional, dengan bahasa pemrograman Singkong.

Contoh-contoh yang dibahas mencakup pengenalan sistem database relasional, koneksi, query, dan topik lanjutan. Anda tidak perlu memahami database untuk dapat mengikuti pembahasan dalam buku ini. Akan tetapi, sebagai buku lanjutan, pemahaman akan bahasa Singkong dan bekerja dengan GUI akan membantu.

Melengkapi contoh program, buku ini juga membahas pengantar untuk interaksi manusia dan komputer.

Jakarta, Desember 2022

Tim penulis

Untuk referensi dan dokumentasi lengkap, serta contoh-contoh penggunaan bahasa Singkong, Anda mungkin ingin membaca buku-buku gratis berikut:

- Mengenal dan Menggunakan Bahasa Pemrograman Singkong (ISBN: 978-602-52770-1-6, Dr. Noprianto).
- Contoh dan Penjelasan Bahasa Singkong: Dasar-dasar Aplikasi GUI (ISBN: 978-602-52770-3-0, Dr. Noprianto, Dr. Karto Iskandar, Benfano Soewito, Ph.D.).
- Contoh dan Penjelasan Bahasa Singkong: Mahir Bekerja dengan GUI (ISBN: 978-602-52770-4-7, Dr. Noprianto, Dr. Wartika, Dr. Ford Lumban Gaol).

Semua buku tersebut diterbitkan oleh PT. Stabil Standar Sinergi dan dapat dibaca atau didownload dari: https://singkong.dev

Persiapan

Pertama-tama, siapkanlah sebuah komputer, yang dilengkapi layar, keyboard, dan mouse/trackpad. Walaupun dengan tablet atau ponsel juga memungkinkan secara teknis, akan diperlukan pengaturan/instalasi tambahan yang tidak dibahas dalam buku ini.

Untuk perangkat keras komputernya, dapat menggunakan spesifikasi komputer mulai dari yang terbaru ataupun yang telah dijual sekitar 20 tahun yang lalu. Yang penting, dapat menjalankan salah satu dari daftar sistem operasi berikut.

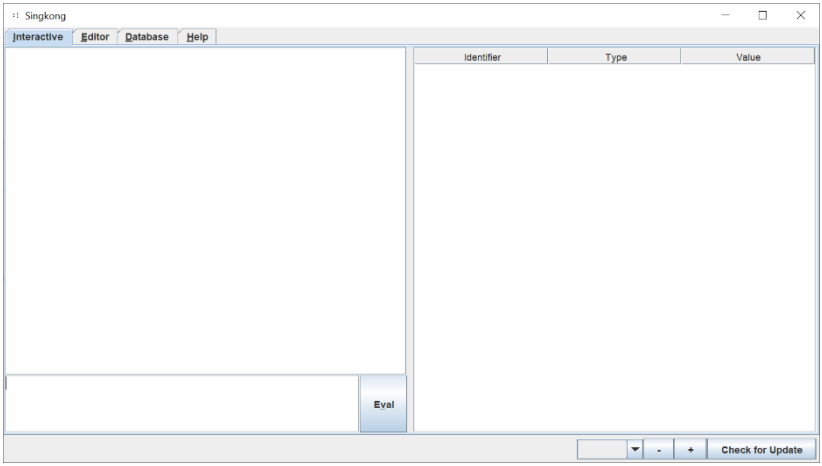
Interpreter Singkong (dan program yang Anda buat nantinya) dapat berjalan pada berbagai sistem operasi berikut:

- macOS (mulai dari Mac OS X 10.4 Tiger)
- Windows (mulai dari Windows 98)
- Linux (mulai yang dirilis sejak awal 2000-an; juga termasuk Raspberry Pi OS dan Debian di Android)
- Chrome OS (sejak tersedia Linux development environment, telah diuji pada versi 101 di chromebook)
- Solaris (telah diuji pada versi 11.4)
- FreeBSD (telah diuji pada versi 13.0 dan 12.1)
- OpenBSD (telah diuji pada versi 7.0 dan 6.6)
- NetBSD (telah diuji pada versi 9.2 dan 9.0)

Setelah komputer siap, lakukanlah instalasi Java, apabila belum terinstal sebelumnya. Secara teknis, Anda hanya membutuhkan Java Runtime Environment, versi 5.0 atau lebih baru. Versi 5.0 dirilis pada tahun 2004 (sekitar 18 tahun lalu pada saat buku ini ditulis). Gunakan versi yang masih didukung secara teknis, apabila memungkinkan. (Kenapa perlu menginstalasi Java? Karena interpreter Singkong ditulis dengan bahasa Java dan bahasa Singkong itu sendiri.)

Sebagai langkah terakhir, downloadlah interpreter Singkong, yang akan selalu didistribusikan sebagai file jar tunggal (Singkong.jar). Downloadlah selalu dari https://nopri.github.io/Singkong.jar. Pada saat buku ini ditulis, ukurannya hanya 4,3 MB dan berisikan semua yang diperlukan untuk mengikuti semua contoh dalam buku ini.

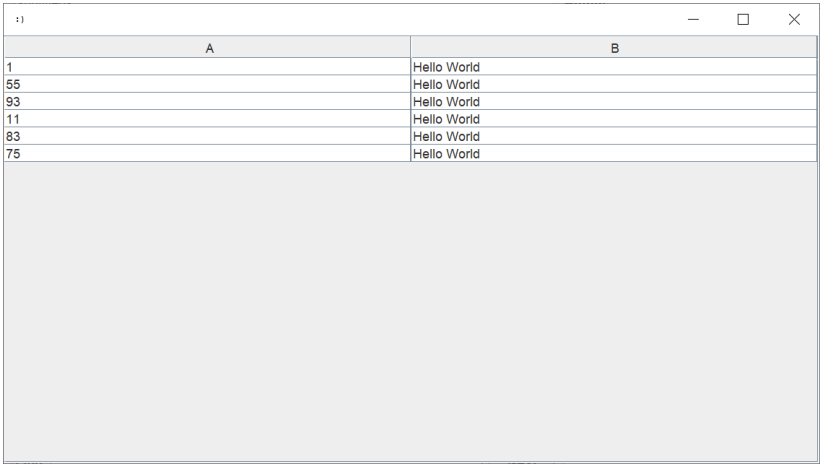
Apabila Singkong.jar dapat dijalankan, maka persiapan sudah selesai (silahkan langsung melanjutkan ke contoh pertama). Perhatikanlah bahwa kita akan aktif pada tab Interactive untuk menguji kode program secara langsung dan tab Editor (sebagian besar contoh) untuk mengetikkan, menyimpan/membuka, dan menjalankan kode program yang lebih panjang.



Apabila langkah detil instalasi Java dan menjalankan Singkong.jar diperlukan, bacalah juga halaman 5 pada buku gratis: *Contoh dan Penjelasan Bahasa Singkong: Dasar-dasar aplikasi GUI*.

Dan, apabila Anda perlu mendistribusikan program yang Anda buat dengan bahasa Singkong dalam satu file jar yang dapat dijalankan, bacalah juga bab *Distribusi Aplikasi* pada buku gratis: *Mengenal dan Menggunakan Bahasa Pemrograman Singkong*.

Contoh 1: Program GUI dan Database dalam 15 Baris Kode



Ketika membuat program yang bekerja dengan data, besar kemungkinan, kita perlu menyimpan data tersebut ke media yang lebih permanen.

Mari kita lihat contoh kode program berikut:

```
reset()


var n = component("text", "")
config(n, "border", "Nama")
var b = component("button", "Tambah")
var t = component("table", "Nama")


add(t)
add_s([n, b])
```

```
show()


event(b, fn() {
    var c = trim(get(n, "contents"))
    if (!empty(c)) {
        table_add(t, [[c]])
        config(n, "contents", "")
        config(n, "focus", true)
    }
})
```

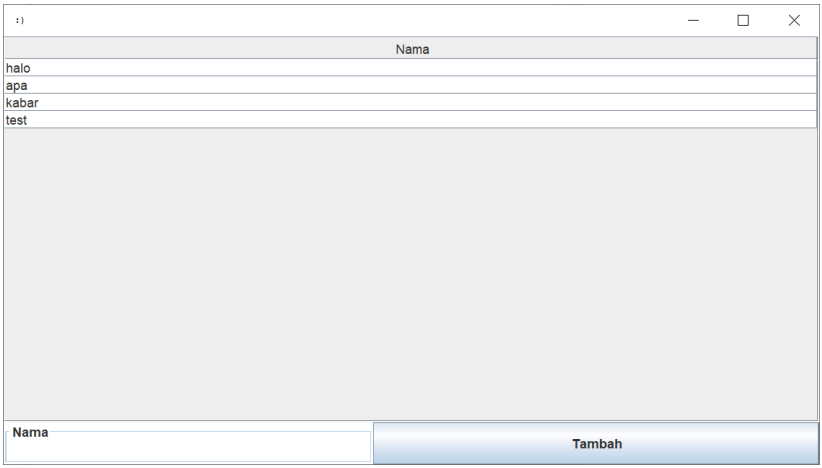
Program tersebut memungkinkan penggunanya untuk menginput nama dan klik button Tambah. Apabila input tidak kosong, maka data yang diinput (diwarnai hijau) akan ditambahkan pada table (diwarnai biru). Dalam bentuk yang sangat sederhana, kita telah membuat sebuah formulir untuk data entri. Tentu tidak sulit, karena kita telah membahas ini di buku-buku sebelumnya.

Yang akan menjadi perhatian kita adalah: daftar nama yang diinput hanya tersimpan memory. Ketika program ditutup, maka apa yang diisikan tersebut pun ikut hilang. Kita pun sudah punya solusinya, sebagaimana dibahas di buku Contoh dan Penjelasan Bahasa Singkong: Dasar-dasar Aplikasi GUI. Yaitu, dengan menyimpan ke file, seperti dengan menambah baris kode berikut setelah table_add:

```
        append("data.txt", c + lf())
```

Ketika program ditutup, apa yang diisikan sebelumnya masih tersimpan, dan dengan mudah kita tampilkan. Mari tambahkan baris-baris kode berikut sebelum kita memanggil fungsi show.

```
var c = split(read("data.txt"), lf())

each(c, fn(e, i) {

    table_add(t, [[e]])

})
```



Dengan demikian, setiap kali program dijalankan, isi dari data.txt akan ditampilkan dalam table. Setiap kali nama ditambahkan ke dalam table, nama tersebut juga akan ditambahkan dalam data.txt.

Dalam 20-an baris kode, kita telah menyimpan data ke dalam media yang lebih permanen, yaitu sebuah file pada file system.

Jadi, sampai di sini saja pembahasan kita?

Tentu saja tidak. Kita bahkan belum membahas penggunaan sistem database relasional.

Sekarang, mari kita lihat beberapa poin berikut:

1. Masih di dalam buku Contoh dan Penjelasan Bahasa Singkong: Dasar-dasar Aplikasi GUI, kita membahas bagaimana kita menyimpan ke file teks biasa dan file CSV.
    a. Di dalam contoh sebelumnya, kita hanya menyimpan satu nilai per baris, yang dipisahkan karakter line feed ("turun ke baris berikut"). Pada kenyataannya, kita mungkin perlu menyimpan beberapa nilai (misal: nama, alamat, nomor telepon) per baris.
    b. Sebagaimana kita ingat, sebuah nilai bisa saja mengandung line feed tersebut (misal: input beberapa baris dengan component edit). Ini akan menjadikan definisi nilai per baris menjadi tidak jelas, apabila dengan file teks biasa. Ini adalah salah satu alasan kita di dalam buku tersebut untuk menggunakan format CSV.
    c. Tergantung sistem operasi yang digunakan ataupun kode yang menulis ke file teks, akhir baris mungkin berupa line feed ataupun carriage return ("ke awal baris") diikuti line feed. Karena file teks dapat diedit dengan berbagai editor teks, data yang akan kita baca lagi mungkin disimpan penanda akhir barisnya secara berbeda. Dengan demikian, mungkin diperlukan pemrosesan tambahan.
    d. Kendati kita dapat mengabaikan penanda akhir baris, dengan beberapa nilai dalam satu baris, ada kalanya ada nilai yang tidak diisikan oleh pengguna. Kita pun perlu memikirkan pemisah setiap nilai tersebut apabila dengan file teks biasa.
2. Bahasa Singkong menyediakan modul bawaan untuk baca tulis file CSV (yang juga ditulis sepenuhnya dengan bahasa Singkong). Jadi, kita menggunakan file CSV saja dan pembahasan selesai? Tentu saja juga tidak.

a. Satu file CSV dapat merepresentasikan satu data tabular, dimana setiap ‘baris’ dapat berisikan sejumlah ‘kolom’, yang dipisahkan karakter pemisah tertentu. Apabila jumlah kolom per baris tidak sama, karakter pemisah tetap ditambahkan. Apabila karakter pemisah terkandung dalam nilai, maka nilai akan dikutip (“”).
b. Namun, seringnya, kita tidak hanya bekerja dengan satu data tabular saja. Misal, dalam mengelola produk yang dijual, kita memiliki file produk.csv. Tapi, kita juga bekerja dengan data penjualan, yang disimpan ke penjualan.csv. Dan tentunya, dalam data penjualan, kita perlu merujuk ke data produk.
c. Ketika relasi data menjadi semakin kompleks, kita perlu mengelola relasi tersebut secara manual. Dengan data produk dan penjualan tersebut saja misalnya, kita perlu memiliki daftar produk (misal: ARRAY dari HASH) yang dibaca sebelumnya dari produk.csv. Lalu, setiap kali terdapat penjualan, kita akan tambahkan baris baru ke penjualan.csv yang diantaranya berisikan nomor produk (bukan nama produknya, karena nama bisa saja diubah untuk produk yang sama), pelanggan, kuantitas, dan harganya. Bukankah kita juga perlu mengelola data pelanggan?
d. Bukan saja relasi yang perlu dikelola secara manual. Kita juga perlu tahu kapan baca ulang harus dilakukan pada file produk.csv atau pelanggan.csv, karena baris-baris pada kedua file tersebut bisa ditambahkan sementara penjualan akan dilakukan.
e. Kemudian, kalau dipikir, tipe data untuk tanggal/waktu ataupun jumlah produk yang dijual tentunya akan berbeda. Di Singkong, tanggal/waktu adalah tipe data DATE, dan jumlah produk adalah NUMBER. Tapi, karena

file CSV adalah file teks juga (semua adalah karakter tekstual), pemetaan dari teks ke tipe yang tepat harus dilakukan secara manual. Ini pun harus kita kelola secara manual.

3. Jadi, kita gunakan file dengan format tersendiri yang dapat mengakomodir relasi antar nilai atau setidaknya tipe data? Singkong menyediakan sejumlah fungsi untuk bekerja dengan file system dan file. Tentu juga tidak. Karena, kalau demikian, maka judul buku ini tentunya tidak mengandung Bekerja dengan Database Relasional.
    a. Apabila kita menempuh jalur ini, rasanya akan lebih mudah untuk menggunakan format file lain yang sudah lebih teruji. Tapi, kita mungkin perlu tahu cara untuk membaca tulis format tersebut. Umumnya, ini tidaklah trivial semudah memanggil satu dua fungsi.
    b. Selain baca, tulis, dan mengelola data, kita perlu pikirkan ketika sejumlah program perlu baca tulis file tersebut secara bersamaan. Sistem operasi dan file system akan punya aturan (“batasan”) tersendiri. Untuk urusan ‘tulis bersamaan’, kita mungkin perlu melakukan prosedur tersendiri yang juga rasanya tidak semudah memanggil satu fungsi.
    c. Lalu, ketika bekerja dengan file pada file system, kita akan mengakses file tersebut secara langsung. Konsep otentikasi (“apakah pengguna terbukti sah”) dan otorisasi (“haknya apa saja”) pada program mungkin tidak terlalu efektif apabila file bisa dikopikan begitu saja. Tentunya, kita tidak akan menempuh jalur yang lebih sulit lagi dengan enkripsi/dekripsi.
    d. Pada akhirnya, untuk bekerja dengan nilai tertentu dalam file tersebut, kita harus melakukan langkah-langkah tertentu. Bukan sekedar menggambarkan apa

yang diinginkan. Alih-alih “tolong tambahkan satu baris”, kita perlu (1) siapkan struktur data barisnya (misal bekerja dengan ARRAY) dan (2) tulis ke file dengan memanggil fungsi tertentu.

4. Kita sudah cukup panjang membahas ini. Bahkan, ketika menulispun, rasanya cukup melelahkan (terutama ketika batas waktu rilis buku sudah dekat dan ini baru bab pertama). Jadi, kita ingin:
    a. Mengelola data tabular tanpa repot memikirkan urusan baris dan kolom.
    b. Setiap nilai bisa memiliki tipe tersendiri dan pemetaan otomatis akan dilakukan ke tipe data Singkong ketika baca tulis dilakukan.
    c. Bekerja dengan relasi antar data tabular dengan relatif mudah.
    d. Tidak perlu bekerja dengan format file khusus secara manual. Juga tidak perlu pusing memikirkan urusan baca tulis bersamaan dan aturan dari sistem operasi dan file system. Sebagai bonus, kita mungkin mendapatkan pengelolaan pengguna dan haknya (otentikasi dan otorisasi).
    e. Seolah masih kurang nyaman, kita pun ingin baca tulis dengan cukup memberikan perintah apa yang diinginkan. Langkah-langkahnya? Tidak perlu repot dipikirkan.
    f. Kita mungkin membutuhkan transaksi: tulis beberapa data sekaligus (yang menjadi bagian dari satu transaksi) atau tidak sama sekali.
    g. Apabila suatu hari data yang dikelola menjadi sangat besar dan kompleks, serta perlu selalu tersedia pada berbagai server yang berbeda (andaikata ada server yang

terkendala), maka kita pun mungkin tidak perlu repot memikirkan.

h. Kita cukup menggunakan sistem database relasional yang ada.

Lebih dari empat halaman tanpa gambar untuk membahas hal ini? Contoh sederhana dalam 15 baris kode? Bahkan contoh tanpa database saja lebih dari 15 baris. Apakah buku ini bahkan membosankan di bab pertama? Anda mungkin bertanya-tanya.

Kadang penulis juga bertanya ke diri sendiri. Mungkinkah lebih baik pembahasan seperti ini ditempatkan pada bab tersendiri? Mungkinkah sekalian tidak perlu dibahas? Tapi bukankah empat halaman juga cukup berarti demi menambah isi buku dan semangat untuk menulis :)?

Baiklah. Mari kita lihat contoh yang 15 baris tersebut (*akhirnya!*). Sebelum tentunya kita menutup bab ini dengan beberapa poin pembahasan lagi (*ya, ampun!*).

```
load_module("db_util")

reset()

var t = component("table", "A,B", true)

add(t)


var d = db_connect_embed("test")

if (d != null) {

    db_create_table_embed(d,      "test",      [["a",
"integer."], ["b", "varchar."]])
```

```
    db_insert(d, "test", {"a": random(0,100), "b":
"hello"})

    db_update(d, "test", [["b = ", "hello", ""]],
{"b": "Hello World"})

    var r = db_select_all(d, "test")

    config(t, "contents", r[0])

}


show()
```

Karena pemformatan, tampaknya lebih dari 15 baris. Tapi sesungguhnya hanya 13 baris kode. Dengan asumsi Anda menjalankan kode tersebut di direktori dimana Anda memiliki hak tulis ("buat direktori/file"), maka ketika dijalankan berkali-kali, tampilannya akan mirip seperti gambar layar pada awal bab ini.

Mari kita lihat bersama apa yang dapat kita bahas supaya bab pertama ini cepat selesai. Supaya bab-bab lain juga kebagian tempat.

- Kita tidak akan membahas sisi GUI di contoh ini, sebagaimana telah dibahas dalam dua buku Contoh dan Penjelasan Bahasa Singkong: Dasar-dasar Aplikasi GUI dan Contoh dan Penjelasan Bahasa Singkong: Mahir Bekerja dengan GUI.
- Diwarnai hijau: terdapat berbagai pilihan sistem database relasional. Masing-masing akan membutuhkan driver dan parameter koneksi ke sistemnya. Demi kesederhanaan, contoh ini menggunakan driver bawaan dalam interpreter bahasa Singkong (Singkong.jar), yaitu Apache Derby. Driver bawaan lainnya adalah PostgreSQL. Apabila menggunakan sistem database lain, cara koneksinya dapat berbeda pula.
    - Selain client driver dan embedded driver Apache Derby, Singkong.jar juga menyertakan network

server Apache Derby. Jadi, ketika bekerja dengan Apache Derby, baik server dan client, cukup menggunakan Singkong.jar.

  - Apache Derby, yang akan kita bahas lebih lanjut di bab tersendiri, dapat pula bekerja secara embedded, dimana tidak ada server database yang dijalankan secara eksplisit. Kita akan membahas keduanya dalam buku ini, tapi di bab pertama ini, kita terkoneksi ke database embedded.
  - Dengan fungsi db_connect_embed, kita akan melakukan koneksi ke database Apache Derby secara embedded, dan database secara otomatis akan dibuat apabila belum ditemukan.
  - Fungsi-fungsi koneksi database, termasuk db_connect_embed, akan mengembalikan DATABASE apabila berhasil dan NULL apabila gagal. Oleh karenanya, kita cek apakah variabel d adalah NULL.
  - Apabila berhasil, lihatlah isi direktori aktif (dapat pula dengan kode `dir(cwd())` pada tab Interactive). Anda akan menemukan direktori test, karena kita terkoneksi ke (dan membuat otomatis apabila perlu) database test. Direktori tersebut menyimpan isi dari database kita.

- Diwarnai biru: Satu database bisa berisi banyak tabel. Ketika pertama kali program dijalankan dan database belum ditemukan, maka pada saat koneksi dilakukan dengan db_connect_embed, database akan dibuat secara otomatis. Pada saat itu, tentu saja database yang baru dibuat belum berisikan tabel yang akan kita gunakan. Maka, kita membuatnya terlebih dahulu. Salah satu cara yang bisa dilakukan adalah dengan memanggil fungsi

db_create_table_embed. Pembuatan tabel akan dibahas dalam bab tersendiri. Apabila kita terbiasa bekerja dengan format CSV, maka tabel pada database tentunya juga terdiri baris dan kolom. Hanya saja, setiap kolom secara eksplisit memiliki nama dan dapat ditentukan agar menyimpan data dengan tipe tertentu (seperti bilangan, karakter, dan sebagainya).

  - Ketika program dijalankan di kali berikutnya, kita kembali memanggil fungsi db_create_table_embed ini. Yang artinya, membuat tabel lagi. Padahal, tabel yang dimaksud sudah ada. Maka, operasi pada database akan gagal dan fungsi akan mengembalikan NULL.
  - Apabila Anda terbiasa bekerja dengan sistem database relasional, Anda juga mungkin terbiasa menggunakan statement dalam SQL (Structured Query Language). Sebagai contoh, CREATE TABLE, untuk membuat tabel. Kita tidak menggunakan statement ini secara langsung dalam contoh program ini.
  - Kita membuat sebuah tabel dengan dua kolom: a dan b, dimana a menampung nilai bilangan bulat (integer, termasuk NUMBER di Singkong) dan b menampung nilai karakter (varchar, STRING di Singkong).

- Diwarnai merah: kita sudah memiliki satu tabel, sebagaimana kita sudah memiliki satu file CSV. Kini, kita menambahkan satu baris ke tabel tersebut. Sama seperti menambahkan satu baris baru pada file CSV.
  - Karena setiap kolom memiliki nama dan tipe dalam contoh ini, kita menentukan nilai yang diisikan ke kolom tersebut.

    - Kita menambahkan satu baris dimana kolom a berisi nilai acak antara 0 dan 100, dan kolom b berisikan “hello”. Tipe data di Singkong akan dipetakan ke sistem database berdasarkan kriteria tertentu yang akan kita bahas lebih lanjut.
    - Perhatikanlah bahwa kita tidak menggunakan statement SQL secara langsung, seperti halnya INSERT.
- Diwarnai oranye: dengan mudah, kita bisa mengubah isi dari sebuah tabel berdasarkan kriteria tertentu.
    - Sebagai contoh, kita mengubah isi tabel test, dimana setiap baris dengan kriteria kolom b berisi ‘hello’ akan diganti menjadi ‘Hello World’. Ini artinya, termasuk baris yang kita tambahkan sebelumnya (dimana b berisikan ‘hello’).
    - Tentu saja, kriteria pengubahan bisa melibatkan sejumlah kolom dengan pola atau nilai yang lebih rumit.
    - Kita juga tidak menggunakan statement SQL, yaitu UPDATE, secara langsung.
- Diwarnai ungu: sama seperti pada file CSV, kita bisa mendapatkan isi dari tabel. Dengan fungsi db_select_all, keseluruhan isi tabel akan didapatkan.
    - Kriteria untuk mendapatkan isi tabel bisa rumit, melibatkan sejumlah kolom dengan pola/nilai yang juga rumit, dan dapat pula lintas tabel, sesuai relasi yang kita definisikan.
    - Dalam contoh ini, kembali kita tidak menggunakan statement SQL secara langsung. Bagi Anda yang terbiasa bekerja dengan SQL, tentunya, untuk contoh ini, kita dapat menggunakan statement SELECT.

Lima belas baris kode dan lebih dari tiga halaman penjelasan yang bahkan rasanya kurang begitu tuntas?

Apabila demikian, mari kita tambahkan beberapa catatan:

- Terdapat sejumlah pilihan sistem (software) database relasional. SQL sendiri, sebagai bahasa, memang terstandarisasi (dan ada versinya). Akan tetapi, implementasi sistem database dapat tidak sepenuhnya patuh pada standar versi tertentu. Oleh karena itu, beberapa bagian statement SQL mungkin tidak diimplementasikan, atau terdapat statement, klausa, dan perintah yang spesifik pada sistem database tertentu.
- Dalam contoh kode, kita tidak menggunakan statement SQL secara langsung karena kita menggunakan sejumlah fungsi bantu yang disediakan bersama modul db_util. Oleh karena itu, kita perlu load modulnya terlebih dahulu dengan fungsi load_module.
- Terkait tipe data, sistem database tertentu juga dapat memiliki kebijakan berbeda-beda.
    - Ada yang ketika mendefinisikan tipe data, cukup ketat. Misal tipe varchar maksimum berapa panjang. Ada yang cukup tipe varchar saja tanpa harus menyebutkan panjangnya.
    - Umumnya, ketika ditentukan sebuah kolom hanya dapat menampung tipe tertentu, maka ketika nilai selain tipe tersebut diberikan, operasi akan gagal. Namun, sistem database tertentu dapat mengijinkan.
- Apakah Singkong dapat bekerja dengan sistem database selain Apache Derby dan PostgreSQL? Tentu saja. Singkong ditulis dengan Java (dan Singkong) dan dalam hal bekerja

dengan sistem database relasional, mengandalkan JDBC dan drivernya. Kita akan contohkan di bab lain.

- Kenapa contoh ini menggunakan fungsi-fungsi bantu dan bukan SQL secara langsung? Karena apa yang kita contohkan memang cukup sederhana dan bisa diakomodir oleh fungsi-fungsi tersebut. Untuk statement SQL yang rumit, atau pada sistem database tertentu, pilihannya hanyalah statement SQL langsung.
- Apakah kemampuan SQL diperlukan sebelum membaca bab berikutnya? Tentunya tidak. Kita akan bahas bersama. Tapi, karena buku ini tidak spesifik tentang bahasa pemrograman SQL, pembahasannya akan terbatas.

Apakah bab ini sudah selesai? Sayangnya, belum juga.

Bagaimanakah cara Anda menjalankan Singkong.jar? Ini adalah buku ketiga dari rangkaian Contoh dan Penjelasan Bahasa Singkong. Tentunya, Anda pun telah terbiasa. Apakah tinggal klik? Apabila demikian, mari kita tambahkan catatan lagi (*astaga!*). Opsional, tidak terlalu perlu dikhawatirkan, namun mungkin berguna.

- Ketika bekerja dengan database embedded seperti contoh dalam bab ini, direktori aktif cukup penting. Ketika Singkong.jar dijalankan lewat klik, program pengelolaan file sistem operasi Anda mungkin menentukan di direktori mana Singkong.jar dijalankan. Periksalah apakah Anda memiliki hak tulis pada direktori itu, sebagaimana dibahas sebelumnya.
- Apabila Anda bekerja dengan database embedded namun ingin database dibuat pada home directory, di mana Anda memiliki hak tulis, fungsi bantu db_connect_embed_user mungkin berguna.

- Ketika bekerja dengan sistem database selain Apache Derby dan PostgreSQL, di mana kita perlu menentukan driver sistem database secara eksplisit, Singkong.jar perlu dijalankan dengan cara berbeda. Kita akan membahas ini nanti.

Sampai di sini, bab pertama dari buku ini pun berakhir. Total 15 baris dijanjikan, namun 15 halaman disajikan. Lalu, apakah kalau contoh kode berikutnya 30 baris, akan membutuhkan 30 halaman pembahasan? Pastinya, tidak. Banyak hal mendasar telah kita bahas pada bab ini. Contoh-contoh berikutnya akan lebih praktis.

Halaman ini sengaja dikosongkan

Mengenal Lebih Dekat Database Relasional

Penulis: Dr. Maria Seraphina Astriani

Pada saat menggunakan suatu aplikasi misalkan untuk memesan tiket pesawat, umumnya aplikasi tersebut perlu mengambil data penerbangan serta menyimpan data booking kita pada aplikasi. Untuk memenuhi kebutuhan akan hal tersebut, diperlukan suatu tempat khusus untuk menyimpan seluruh data dan disinilah database berperan penting.

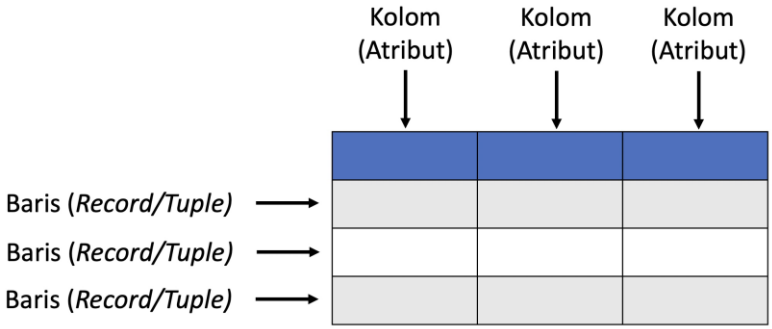
Mungkin kita sudah terbiasa sehari-hari mendengar dan menggunakan istilah database, namun di dalam bahasa Indonesia, database disebut sebagai basis data. Yuk, mari kita mengenal lebih dekat tentang database, terutama tentang database relasional karena jika tak kenal, maka tak sayang ^_^.

Istilah database ternyata sudah ada dari tahun 1960-an bersumber dari US military information system dan istilah relasional (relational) berasal dari paper yang ditulis oleh E.F. Codd pada tahun 1970 dengan judul "A Relational Model of Data for Large Shared Data Banks." Namun paper Peter Chen pada 1976 dengan judul "The Entity Relationship Model-Toward a Unified View of Data" yang memberikan kontribusi yang sangat besar terhadap database relasional yang kita gunakan saat ini. Database relasional dapat dilihat sebagai suatu koleksi data yang terorganisir dan saling berhubungan antara satu dengan yang lainnya.

Database relasional berguna untuk mengatur data sehingga dapat mempermudah untuk memilah dan menemukan informasi yang ingin kita cari. Jika perusahaan menggunakan database relasional pada aplikasinya, mereka menjadi lebih terbantu karena dapat membuat

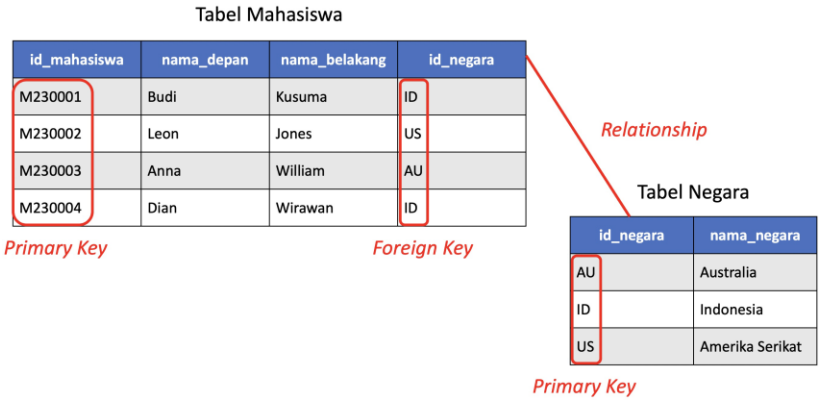
keputusan bisnis lebih efisien dan tentu saja dapat meminimalkan biaya dibanding tidak menggunakan database sama sekali.

Struktur database relasional terdiri dari table (tabel) dan relationship (relasi/hubungan). Apakah bentuk tabelnya sama dengan tabel yang kita kenal? Ternyata sama! Tabel terdiri dari kolom dan baris. Nah, di dalam database relasional, kolom dikenal sebagai attribute (atribut) dan baris dikenal sebagai record atau tuple, sebagaimana dapat dilihat pada gambar berikut.



Kolom dan Baris Pada Tabel Database Relasional

Biasanya setiap tabel mempunyai primary key (kunci primer), disingkat PK, yang unik (tidak ada duplikatnya) yang mengindentifikasikan suatu baris dalam tabel. Foreign key (kunci asing), disingkat sebagai FK, berguna sekali pada database relasional karena dapat digunakan sebagai penanda untuk mengetahui relasi antara satu tabel dengan yang lainnya. Mari kita lihat gambar di halaman berikut untuk contoh pengimplementasian primary key dan foreign key.



Contoh Primary Key dan Foreign Key Pada Relationship
Tabel Mahasiswa - Tabel Negara

Pada tabel “Mahasiswa” terdapat terdapat primary key pada atribut “id_mahasiswa” dan pada tabel “Negara” atribut “id_negara” merupakan primary key. Jika kita lihat, seluruh data yang ada pada “id_mahasiswa” tidak ada satupun yang sama atau tidak ada duplikatnya, dan umumnya memang “id_mahasiswa” pastilah berbeda antar satu mahasiswa dengan yang lain. “id_mahasiswa” ini menandakan/mewakili suatu data, misalkan M230001 pasti merupakan ID mahasiswa dengan nama Budi Kusuma dan M230003 adalah ID mahasiswa Anna William.

Bagaimana dengan foreign key? Foreign key merupakan suatu yang sangat spesial terutama di dalam database relasional karena dapat sebagai penghubung/penunjuk bahwa data pada suatu tabel berelasi dengan tabel lainnya. Pada table “Mahasiswa”, atribut “id_negara” merupakan foreign key dan terdapat data ID, US, dan AU. Mungkin kita tidak mengetahui apa maksud dari data tersebut. Namun jika dilihat, atribut tersebut berelasi dengan atribut “id_negara” pada tabel “Negara” sehingga kita dapat mengetahui bahwa ID merupakan

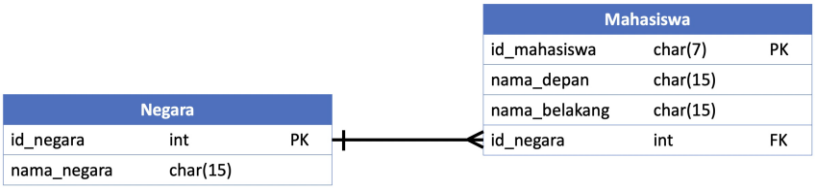
“id_negara” untuk nama negara Indonesia. Jadi jika ingin mengetahui data mahasiswa dengan “id_mahasiswa” M230001, kita dapat mengetahui bahwa nama depannya adalah Budi, nama belakangnya Kusuma, dan dari negara Indonesia.

Oh iya, data pada foreign key boleh tidak unik, tidak seperti pada primary key. Jadi jika kita lihat contoh pada Gambar x.2 di dalam tabel “Mahasiswa” pada atribut “id_negara” terdapat dua ID, hal itu sah-sah saja ^_^.

Pada database relasional umumnya terdapat tiga jenis relasi:

1. One-to-one
2. One-to-many
3. Many-to-many

Relasi ini dapat memberikan gambaran bagaimana hubungan antara satu tabel dengan tabel yang lain. Jika gambar sebelumnya dibuat relasinya, maka dapat digambarkan seperti berikut.



Contoh Relasi One-to-Many

Jika dilihat, relasi tabel “Negara” dan tabel “Mahasiswa” adalah relasi one-to-many (dilambangkan dengan ) karena satu negara bisa terdapat banyak penduduk, dan di dalam contoh ini ada banyak mahasiswa yang berasal dari satu negara.

Saat ini kita sudah lebih mengenal tentang database relasional. Namun, jika ingin mengimplementasikan atau menghubungkan dengan suatu aplikasi, kita membutuhkan Relational Database Management System atau dapat disingkat sebagai RDBMS. Nah, untuk mempermudah membayangkan RDBMS itu seperti apa, kita dapat membayangkan RDBMS merupakan suatu program yang mempermudah kita untuk mengelola (manage) database relasional seperti melakukan create, read, update, dan delete (operasi CRUD) data di dalam database.

Sebagian besar RDBMS menggunakan SQL untuk berinteraksi dengan database. Contoh RDBMS adalah MySQL, PostgreSQL, Microsoft SQL Server, Microsoft Access, MariaDB, Oracle Database, Apache Derby, IBM Db2, Firebird, dan masih banyak lainnya.

Yang membuat semakin asik lagi, kita bisa menggunakan database relasional juga lho pada bahasa Singkong. Derby dan PostgreSQL merupakan sistem database yang built-in datang bersama interpreter Singkong.

Halaman ini sengaja dikosongkan

Interaksi Manusia dan Komputer

Penulis: Dr. Fredy Purnomo

Pada dekade sebelumnya, interaksi manusia dengan komputer hanya sebatas pada layar monitor, mouse, dan keyboard, dimana komputer kebanyakan berupa desktop atau personal computer yang tetap di suatu tempat. Segala interaksi yang terjadi dilakukan di sebuah meja untuk komputer dan kursi untuk pengguna. Namun semua itu berubah ketika muncul era perangkat bergerak (mobility), dengan perangkat yang ada, kita bisa mengakses aplikasi dan data di mana saja dan kapan saja, apalagi didukung dengan teknologi selular yang menjadi media penghubung antar perangkat dan perangkat dengan server. Dengan kemudahan yang ada tersebut, maka pengguna komputer tidak hanya eksklusif bagi mereka yang bekerja di bidang teknologi informasi saja, namun berkembang untuk semua kalangan, dari anak-anak, dewasa dan orang tua. Apalagi, fitur layar sentuh membuat operasi komputer menjadi sangat natural bagi mereka yang awam.

Era dimana kita mengevaluasi aplikasi dari segi tampilan di layar monitor saja sudah lewat. Sekarang, kita ada di era dimana evaluasi menjadi multidimensi pada berbagai macam perangkat baik yang station ataupun yang bergerak, dari ukuran layar 21 inchi sampai 3 inchi. Itupun ditambah dengan tujuan pemakaian, apakah dipakai indoor atau outdoor, untuk anak-anak, dewasa, atau orang tua.

Ilmu interaksi manusia dan komputer (human computer interaction) berkembang untuk mengakomodir hal tersebut. Berbagai cabang ilmu digabungkan dengan tujuan untuk merancang sebuah aplikasi yang bisa mendukung manusia untuk berinteraksi dan berkomunikasi baik dalam pekerjaan maupun kehidupan sehari-hari. Diantaranya,

ilmu ergonomis, psikologi, design, social, dan tentu saja ilmu komputer.

Tujuan utama dalam human computer interaction adalah menghasilkan aplikasi yang efektif, efisien, aman digunakan, dengan utilitas yang bagus, mudah dipelajari, dan mudah diingat. Walau tidak ada hal yang sempurna, namun proses dalam perancangan aplikasi dilakukan secara iteratif untuk menghasilkan rancangan aplikasi yang memuaskan semua pihak.

Ada empat tahapan penting dalam membuat design aplikasi, yaitu penggalian kebutuhan, perancangan alternatif, prototyping, dan evaluasi. Proses penggalian kebutuhan melibatkan para pengguna yang nantinya akan berinteraksi dengan aplikasi tersebut. Dibentuk kelompok kecil untuk mendengarkan harapan dari pimpinan dan brainstorming dengan para user untuk mendapatkan gambaran apa yang akan dibuat. Designer harus bisa membedakan mana yang merupakan kebutuhan pokok dan mana yang merupakan keinginan tambahan dari user.

Proses perancangan alternatif adalah proses dimana interpretasi rancangan dibuat dengan beberapa model pilihan. User akan dilibatkan kembali untuk menilai dan memberikan masukan. Disini kebutuhan user dikofirmasi ulang.

Proses prototyping dilakukan ketika design sudah ada bayangan lebih jelas. Metode prototyping dilakukan dengan mengajukan tampilan aplikasi persis seperti hasil jadinya, namun tentu saja tidak ada fungsi yang berjalan. Prototype seperti sebuah mainan mobil yang mereprentasikan bentuk jadi dari mobil yang sebenarnya. User kembali dilibatkan untuk melakukan testing untuk prototype yang ada, baik secara individu atau berkelompok. Biasanya beberapa skenario simulasi dipakai untuk menguji prototype yang ada.

Proses evaluasi melibatkan user dan designer untuk menentukan hasil dari evaluasi prototype sebelumnya. Saling berdiskusi fitur mana saja yang akan dipakai dan fitur apa yang yang seharusnya ada, termasuk juga dari segi tampilan, pemilihan warna, pemilihan dialog, ukuran serta peletakan tombol.

Dalam pengembangan design yang baik, metafora menduduki peranan yang penting. Design yang baik adalah design yang mampu menerjemahkan kondisi di dunia nyata ke dalam tampilan aplikasi. Misalnya ketika orang ingin membuat aplikasi untuk pengaturan switch lampu di rumah, maka metafora saklar harus menjadi acuan. Kebiasaan orang menekan tombol switch on atau off yang sehari-hari dilakukan, harus mampu diterjemahkan dalam tampilan aplikasi. Ketika orang diminta untuk membuat aplikasi social media, tentu saja harus ada metafora berdiskusi sekelompok orang. Model diskusi ini yang kemudian diwujudkan dalam bentuk tampilan dan interaksi aplikasi. Penggunaan metafora yang tepat akan mengurangi retensi atau penolakan user dalam menggunakan aplikasi. User akan merasakan hal yang sama dengan cara manual maupun menggunakan aplikasi, bahkan lebih menyenangkan. Itu yang diharapkan.

Halaman ini sengaja dikosongkan

Contoh 2: Driver dan Koneksi Database

Pada contoh sebelumnya, kita melakukan koneksi ke database Apache Derby Embedded. Tidak membutuhkan server database yang dijalankan secara eksplisit. Di contoh ini, mari kita bahas lebih lanjut perihal koneksi database, baik untuk yang dibundel bersama Singkong.jar ataupun sistem database lainnya.

Dengan modul db_util dan database yang dibundel

Modul db_util menyediakan empat fungsi untuk koneksi database:

db_connect
db_connect_embed
db_connect_embed_
db_connect_embed_user

Dapat dilihat, terdapat 3 varian untuk bekerja dengan database Derby Embedded. Perbedaan antara db_connect_embed_ dan db_connect_embed hanyalah pada pembuatan database secara otomatis apabila belum ditemukan. Sementara, sebagaimana telah dibahas sebelumnya, db_connect_embed_user akan melakukan koneksi database pada home directory (dan membuatnya apabila belum ada).

Bagaimana dengan db_connect? Fungsi ini dapat digunakan untuk melakukan koneksi database, akan tetapi, hanya untuk database yang dibundel dan didukung secara resmi. Bagaimana dengan koneksi ke sistem database lainnya? Kita akan menggunakan fungsi database secara langsung.

Mari kita lihat penggunaan db_connect terlebih dahulu. Untuk itu, kita akan terkoneksi masing-masing ke:

- PostgreSQL (localhost, port: 5432, nama database: test, user: test, password: test).
- Network server Apache Derby (localhost, port: 1527, nama database: test, user: test, password: test).
- Apache Derby secara Embedded, dengan nama database: test, di direktori aktif.

Catatan: Cara menjalankan network server Apache Derby dan instalasi/konfigurasi PostgreSQL (Windows, macOS, Linux) akan dibahas secara singkat pada akhir bab ini.

Ketikkanlah dua baris kode berikut pada tab Interactive. Pastikanlah modul db_util telah diload dengan load_module("db_util").

```
>  var d_pgsql = db_connect("postgresql",
"//localhost/test", "test", "test")

>  d_pgsql

DATABASE (URL=jdbc:postgresql://localhost/test,
user=test, driver=org.postgresql.Driver)
```

Bisa kita lihat, variabel d_pgsql adalah merupakan suatu DATABASE dan bukannya NULL, sehingga koneksi database berhasil.

Catatan: Dengan fungsi db_connect ke PostgreSQL, untuk argumen pertama, kita bisa melewatkan: “postgresql”, “pgsql”, ataupun “postgres”, dalam berbagai variasi huruf besar/kecil.

Apakah lebih nyaman menyebutkan nama host dan port secara eksplisit? Lewatkanlah pada argumen kedua, seperti contoh berikut:

```
>  var d_pgsql = db_connect("postgresql",
"//localhost:5432/test", "test", "test")
```

```
>  d_pgsql

DATABASE
(URL=jdbc:postgresql://localhost:5432/test,
user=test, driver=org.postgresql.Driver)
```

Mari kita lanjutkan ke network server Apache Derby. Ketikkanlah dua baris berikut, juga pada tab Interactive:

```
>  var d_derby = db_connect("derby",
"//localhost:1527/test", "test", "test")

>  type(d_derby)

"NULL"
```

Kode tersebut mengasumsikan database test belum ditemukan pada direktori aktif server. Bisa kita lihat, koneksi gagal karena d_derby adalah NULL.

Mari kita buat pada saat koneksi.

```
>  var d_derby = db_connect("derby",
"//localhost:1527/test;create=true", "test",
"test")

>  d_derby

DATABASE
(URL=jdbc:derby://localhost:1527/test;create=true,
user=test,
driver=org.apache.derby.jdbc.ClientDriver)
```

Bisa kita lihat bersama, dengan menambahkan ;create=true, database dibuat dan dapat dilihat pada direktori aktif server.

Bagaimana dengan Apache Derby secara embedded dengan fungsi db_connect? Mari kita lihat contoh kodenya, dengan asumsi, database test telah ditemukan pada direktori aktif, dari bab pertama.

```
>  var d_embed = db_connect("embedded", "test", "",
"")
```

```
>  d_embed
```

```
DATABASE (URL=jdbc:derby:test, user=,
driver=org.apache.derby.jdbc.EmbeddedDriver)
```

Tentu saja, dalam hal ini, d_embed akan serupa apabila kita menggunakan fungsi db_connect_embed_ dengan false dilewatkan sebagai argumen kedua:

```
>  var d_embed = db_connect_embed_("test", false)
```

```
>  d_embed
```

```
DATABASE (URL=jdbc:derby:test, user=,
driver=org.apache.derby.jdbc.EmbeddedDriver)
```

Perhatikanlah apabila argumen kedua adalah true (database dibuat):

```
>  var d_embed = db_connect_embed_("test", true)
```

```
>  d_embed
```

```
DATABASE (URL=jdbc:derby:test;create=true, user=,
driver=org.apache.derby.jdbc.EmbeddedDriver)
```

Bisa kita lihat, ;create=true akan ditambahkan. Ini sama seperti pada bab pertama ketika kita menggunakan db_connect_embed:

```
>  var d_embed = db_connect_embed("test")
```

```
>  d_embed
```

```
DATABASE (URL=jdbc:derby:test;create=true, user=,
driver=org.apache.derby.jdbc.EmbeddedDriver)
```

Catatan: Ketika menggunakan Apache Derby secara embedded, hanya satu Java virtual machine yang dapat terkoneksi ke database embedded tersebut. Koneksi lain dari Java virtual machine lain ke database yang sama akan gagal.

Sebagai contoh: jalankanlah Singkong.jar dua kali, masing-masing dari direktori aktif yang sama. Pada tab Interactive pertama kita melakukan dua kali koneksi dan berhasil:

```
>  load_module("db_util")
>  var d = db_connect_embed("test")
>  d
DATABASE (URL=jdbc:derby:test;create=true, user=,
driver=org.apache.derby.jdbc.EmbeddedDriver)

>  var d = db_connect_embed("test")
>  d
DATABASE (URL=jdbc:derby:test;create=true, user=,
driver=org.apache.derby.jdbc.EmbeddedDriver)
```

Pada tab Interactive kedua (tentunya merupakan Java virtual machine yang lain), mari kita lakukan koneksi ke database yang sama:

```
>  load_module("db_util")
>  var d = db_connect_embed("test")
>  type(d)
"NULL"
```

Bisa kita lihat, cara koneksi sama persis dan yang kedua akan gagal. Apabila dibutuhkan sejumlah koneksi dari Java virtual machine berbeda, kita akan menggunakan network server Apache Derby.

Tanpa modul db_util dan database yang dibundel

Dengan empat fungsi koneksi database yang disediakan oleh modul db_util, koneksi database dapat dilakukan dengan relatif lebih sederhana. Cukup menyebutkan nama sistem database, URL, user, dan password. Akan tetapi, sebagaimana disebutkan sebelumnya, fungsi-fungsi bantu tersebut hanya berlaku pada database yang dibundel dan didukung secara resmi.

Apabila Anda juga perlu bekerja dengan sistem database lain, terbiasa menggunakan fungsi database mungkin akan membantu karena jadinya cukup mengingat satu fungsi saja. Lagi pula, keempat fungsi bantu tersebut juga memanggil fungsi database yang sama.

Fungsi database membutuhkan empat argumen:

- STRING: driver database, nama class Java dalam class path.
- STRING: URL database, diberikan lengkap dengan prefix jdbc.
- STRING: user
- STRING: password

Bisa dilihat, perbedaan dengan fungsi bantu hanyalah pada argumen pertama dan kedua. Untuk database yang dibundel, argumen pertama adalah salah satu dari berikut, dituliskan persis:

- Network server Apache Derby: “org.apache.derby.jdbc.ClientDriver”
- Apache Derby Embedded: "org.apache.derby.jdbc.EmbeddedDriver"
- PostgreSQL: “org.postgresql.Driver”

Sementara, untuk argumen kedua, prefix berikut, juga dituliskan persis, perlu ditambahkan, apabila dibandingkan ketika memanggil empat fungsi bantu sebelumnya:

- Apache Derby: “jdbc:derby:”
- PostgreSQL: "jdbc:postgresql:"

Catatan: Perhatikanlah bahwa nama class driver dan prefix jdbc haruslah dituliskan persis, secara case-sensitive.

Berikut adalah contoh koneksi ke database PostgreSQL (localhost, port: 5432, nama database: test, user: test, password: test):

```
>  var d_pgsql = database("org.postgresql.Driver",
"jdbc:postgresql://localhost:5432/test", "test",
"test")

>  d_pgsql

DATABASE
(URL=jdbc:postgresql://localhost:5432/test,
user=test, driver=org.postgresql.Driver)
```

Berikut adalah contoh koneksi ke network server Apache Derby (localhost, port: 1527, nama database: test, user: test, password: test):

```
>  var d_derby =
database("org.apache.derby.jdbc.ClientDriver",
"jdbc:derby://localhost:1527/test", "test", "test")

>  d_derby

DATABASE (URL=jdbc:derby://localhost:1527/test,
user=test,
driver=org.apache.derby.jdbc.ClientDriver)
```

Dan, berikut adalah contoh koneksi Apache Derby Embedded (nama database: test, di direktori aktif):

```
>  var d_embed =
database("org.apache.derby.jdbc.EmbeddedDriver",
"jdbc:derby:test", "", "")

>  d_embed

DATABASE (URL=jdbc:derby:test, user=,
driver=org.apache.derby.jdbc.EmbeddedDriver)
```

Catatan: Pada contoh dan pembahasan berikut, kita mungkin akan bekerja dengan command line interface. Di Windows, kita akan menggunakan cmd.exe (Command Prompt). Di macOS, kita dapat menggunakan program Terminal bawaan. Dan, di Linux, kita dapat menggunakan berbagai terminal emulator yang datang bersama distribusi Linux yang digunakan.

Di Windows 10, untuk menjalankan cmd.exe dengan direktori aktif sesuai direktori aktif File Explorer: tahanlah tombol Ctrl dan tekanlah tombol L pada keyboard (atau klik pada location di sisi atas window), kemudian ketikkanlah cmd.exe dan tekan Enter. Window command prompt akan ditampilkan dengan direktori aktif yang sesuai.

MySQL dan sistem database Lain

Apabila diperhatikan, dan mungkin Anda juga telah menduga, untuk sistem database lain, perbedaannya hanyalah pada nama class driver dan URL ketika memanggil fungsi database. Informasi tersebut tentunya disediakan oleh sistem database yang akan digunakan.

Sebagai contoh, kita akan menggunakan sistem database MySQL, dan nama class driver serta URL nya adalah:

- Nama class driver: “com.mysql.cj.jdbc.Driver”. Kita akan membutuhkan file class drivernya, yang dalam contoh ini didistribusikan dalam satu file: mysql-connector-j-8.0.32.jar.
- Untuk localhost, port: 3306, nama database: test, user: test, password: test, URL yang digunakan adalah: “jdbc:mysql://localhost:3306/test”.

Catatan: Untuk contoh pembahasan ini, diasumsikan server MySQL telah terinstal dan dijalankan, serta user dan database telah tersedia. Pengujian dilakukan pada MySQL server 8.0.32. Driver JDBC didistribusikan dalam file mysql-connector-j-8.0.32.jar, yang terdapat dalam arsip mysql-connector-j-8.0.32.zip. Semua file yang dibutuhkan didownload dari mysql.com.

Yang menjadi catatan penting di sini adalah di mana mencari class Java tersebut. Apabila interpreter Singkong dijalankan dengan klik pada Singkong.jar atau `java -jar Singkong.jar`, maka tentunya, apabila tidak dibundel bersama Singkong.jar, class dimaksud tidak akan ditemukan. Pada contoh berikut, tentunya koneksi database akan gagal dan variabel d_mysql akan bernilai NULL:

```
>  var d_mysql =
database("com.mysql.cj.jdbc.Driver",
"jdbc:mysql://localhost:3306/test", "test", "test")

>  type(d_mysql)

"NULL"
```

Dengan file Singkong.jar dan mysql-connector-j-8.0.32.jar berada di dalam direktori aktif, kita akan jalankan ulang interpreter bahasa Singkong. Berikanlah sebaris perintah berikut lewat command line:

```
java -cp Singkong.jar:mysql-connector-j-8.0.32.jar
com.noprianto.singkong.Singkong
```

Perintah tersebut dimaksudkan untuk mencari class-class yang dibutuhkan ke dalam file-file Singkong.jar dan mysql-connector-j-8.0.32.jar, dan memanggil fungsi main dari class com.noprianto.singkong.Singkong.

Hasilnya mungkin akan tampak serupa dengan ketika Singkong.jar di klik atau dijalankan dengan `java -jar Singkong.jar`. Bedanya, class com.mysql.cj.jdbc.Driver dapat ditemukan. Oleh karena itu, koneksi database berikut akan berhasil:

```
>  var d_mysql =
database("com.mysql.cj.jdbc.Driver",
"jdbc:mysql://localhost:3306/test", "test", "test")

>  type(d_mysql)

"DATABASE"

>  d_mysql

DATABASE (URL=jdbc:mysql://localhost:3306/test,
user=test, driver=com.mysql.cj.jdbc.Driver)
```

Apabila menggunakan sistem database lain, cobalah juga untuk melakukan koneksi. Caranya akan serupa, dan oleh karenanya, tidak dibahas dalam buku ini.

Catatan: Berikutnya, kita akan mencontohkan cara menjalankan network server Apache Derby. Tidak ada instalasi tambahan, karena hanya Singkong.jar yang diperlukan (untuk versi yang dibundel). Hanya saja, Anda perlu aktif di direktori tertentu, dimana Anda memiliki hak untuk membuat file dan direktori.

Contoh caranya akan fokus pada development dan bukan production. Dengan demikian, sebagai contoh, aspek keamanan tidaklah menjadi fokus kita.

Catatan tambahan: Tidak menggunakan network server Apache Derby? Apabila Anda menggunakan Apache Derby secara embedded, tentunya dapat langsung melanjutkan ke bab berikutnya.

Catatan tambahan lagi: Menggunakan PostgreSQL dan membutuhkan contoh instalasi PostgreSQL pada beberapa sistem operasi? Kita akan membahasnya setelah ini.

Menjalankan network server Apache Derby

Pastikanlah Anda mengetahui path ke Singkong.jar. Anda dapat mengopikan Singkong.jar ke direktori aktif apabila diinginkan. Dan, sekali lagi, pastikanlah Anda dapat membuat file atau direktori di direktori aktif tersebut.

Apabila hanya sekedar ingin mencoba menjalankan network server Apache Derby (bukan untuk production) tanpa otentikasi dan security manager, satu baris perintah berikut sudah cukup, dengan Singkong.jar di direktori aktif:

```
java -cp Singkong.jar
org.apache.derby.drda.NetworkServerControl start -
noSecurityManager
```

Jalankanlah interpreter Singkong lain dan cobalah untuk melakukan koneksi sambil membuat database, seperti contoh berikut:

```
>  load_module("db_util")

>  var d = db_connect("derby",
"//localhost:1527/test2;create=true", "test",
"test")

>  d
```

```
DATABASE
(URL=jdbc:derby://localhost:1527/test2;create=true,
user=test,
driver=org.apache.derby.jdbc.ClientDriver)
```

Bisa kita lihat, koneksi dan pembuatan database berhasil. Lihatlah juga isi direktori aktif dimana network server Apache Derby dijalankan, dimana akan ditemukan direktori database dengan nama test2.

Untuk contoh penggunaan security manager (dapat disesuaikan nantinya) dan otentikasi builtin (user: test, password: test), mari kita buat dua file: derby.policy dan derby.properties di direktori aktif.

Berikut adalah isi file derby.policy (dituliskan baris per baris, dimana setiap baris dalam blok grant diawali dengan permission dan diakhiri dengan titik koma):

```
grant {

    permission java.io.FilePermission
"${user.dir}${/}-", "read, write, delete";


    permission java.lang.RuntimePermission
"getFileStoreAttributes";

    permission java.lang.RuntimePermission
"createClassLoader";

    permission java.lang.RuntimePermission
"accessUserInformation";


    permission java.util.PropertyPermission
"derby.__serverStartedFromCmdLine", "read, write";
```

```
    permission java.util.PropertyPermission
"user.dir", "read";


    permission java.net.SocketPermission
"127.0.0.1:1527", "accept, connect,
listen,resolve";

    permission java.net.SocketPermission
"127.0.0.1", "accept, resolve";


};
```

Dan, berikut adalah isi file derby.properties:

```
derby.connection.requireAuthentication=true

derby.authentication.provider=BUILTIN

derby.user.test=test
```

Berikanlah perintah berikut untuk menjalankan network server Apache Derby dengan derby.policy, derby.properties, dan Singkong.jar di direktori aktif:

```
java -cp Singkong.jar -Djava.security.manager -
Djava.security.policy=derby.policy
org.apache.derby.drda.NetworkServerControl start
```

Koneksi ke network server Apache Derby kemudian dapat dilakukan seperti contoh sebelumnya.

Catatan: Setelah ini, kita akan mencontohkan instalasi PostgreSQL di Windows, macOS, dan Linux. Apabila Anda telah melakukan instalasi PostgreSQL atau menggunakan sistem database lain, silahkan untuk melanjutkan ke bab berikutnya.

Catatan tambahan: Instalasi PostgreSQL yang dicontohkan akan fokus pada development dan bukan production. Dengan demikian, sebagai contoh, aspek keamanan tidaklah menjadi fokus kita. Dalam lingkungan production misalnya, kita mungkin tidak mengatur password untuk user postgres.

Instalasi PostgreSQL di Windows

Melalui halaman download dari postgresql.org, kita dapat menggunakan distribusi binary PostgreSQL, tersedia dalam berbagai versi, dilengkapi installer langkah demi langkah yang mudah, serta termasuk client PostgreSQL grafikal yang nyaman.

Akan tetapi, diantaranya untuk buku ini, penulis juga telah menyediakan distribusi PostgreSQL untuk Windows, yang tidak membutuhkan instalasi di sistem, dapat dijalankan dari direktori manapun, oleh user biasa (tidak membutuhkan hak administrator). Ukurannya pun hanya sekitar 11 MB, dalam format zip, yang dapat didownload dari https://nopri.github.io. Berikut adalah link langsungnya: https://nopri.github.io/compiled/postgresql.zip

Sebagai catatan, apabila Anda menggunakan perkedel (https://nopri.github.io/perkedel.exe; bundel compiler, interpreter, dan sistem database untuk Windows), maka PostgreSQL yang sama telah termasuk di dalamnya.

Pada saat buku ini ditulis, versi PostgreSQL dari penulis adalah 9.6.24. Tentunya bukan versi terbaru, namun dapat digunakan untuk contoh-contoh dalam buku ini.

Sistem operasi yang digunakan dalam pembahasan instalasi ini adalah Windows 10. Sesuaikanlah apabila diperlukan. Setelah postgresql.zip didownload, dengan File Explorer, aktiflah pada direktori yang berisi file tersebut. Klik kananlah pada file, kemudian pilih Extract All…. Pada dialog yang tampil, kliklah tombol Extract setelah memilih direktori tujuan. Direktori manapun, selama Anda memiliki hak tulis.

Setelah proses extract selesai, kita akan ke direktori tujuan. Hasil extract adalah sebuah direktori dengan nama postgresql. Jalankanlah cmd.exe (Command Prompt) dan aktiflah di direktori tujuan yang berisi direktori postgresql tersebut (cara cepatnya telah dicontohkan pada catatan ketika bekerja dengan sistem database lain).

Selanjutnya, kita akan bekerja dengan command line dan memberikan beberapa perintah. Termasuk untuk inisialisasi database, menjalankan server PostgreSQL, melakukan pembuatan user, dan pembuatan database untuk user yang baru dibuat. Pada akhirnya, setelah selesai digunakan, server databasenya dapat dihentikan.

Pada window command prompt, ketikkanlah perintah berikut untuk masuk ke direktori postgresql, yang merupakan direktori hasil extract:

```
cd postgresql
```

Inisialisasi cluster database, yang cukup dilakukan sekali, dapat dilakukan dengan perintah berikut:

```
bin\pg_ctl.exe initdb -D data -o "-U postgres -W -A
md5"
```

Kita akan diminta untuk memasukkan password untuk superuser postgres sebagaimana argumen perintah initdb tersebut. Pastikanlah password ini diingat.

Setelah perintah ini selesai, direktori data (ditentukan pada -D sebelumnya) akan ditemukan pada direktori aktif, dan merupakan lokasi untuk cluster database yang dibuat.

Berikutnya, kita akan menjalankan server PostgreSQL. Server database perlu dijalankan agar koneksi database yang dicontohkan dalam buku ini dapat dilakukan. Tentunya, server hanya perlu dijalankan apabila sebelumnya belum berjalan. Kita akan memberikan perintah berikut:

```
bin\pg_ctl.exe -D data start
```

Apabila tidak ada pesan kesalahan, maka server PostgreSQL telah berjalan. Tekanlah Enter untuk kembali ke prompt.

Kemudian, kita akan membuat user test, dimana user ini diberikan hak untuk membuat database. User ini juga diberikan password, dan kita akan isikan test. Perintahnya adalah sebagai berikut (selain password user test, akan diminta juga password superuser postgres yang diisikan pada saat inisialisasi cluster database):

```
bin\createuser.exe -h localhost -U postgres -P -d
test
```

Dengan user test telah dibuat, kita akan membuat database test sebagai user tersebut, karena memiliki hak yang diperlukan (-d). Pada perintah berikut, akan diminta password user test:

```
bin\createdb.exe -h localhost -U test test
```

Pada tahapan ini, kita sudah dapat melakukan koneksi database, seperti yang dibahas dalam bab ini. Akan tetapi, Anda mungkin ingin menggunakan client PostgreSQL psql untuk mencoba melakukan koneksi (sebagai user test, ke database test), seperti contoh perintah berikut:

```
bin\psql.exe -h localhost -U test test
```

Saat ini, belum terdapat tabel. Anda dapat melihatnya dengan memberikan perintah \d (atau \dt), diikuti penekanan Enter. Untuk keluar dari psql, berikanlah perintah \q, diikuti Enter.

Server PostgreSQL tentunya perlu tetap berjalan selama program terhubung dan bekerja dengan sistem database. Apabila server database ingin dihentikan, perintah berikut dapat diberikan:

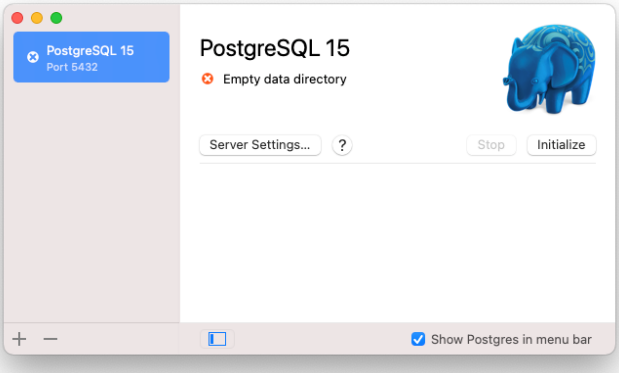
```
bin\pg_ctl.exe -D data stop
```

Instalasi PostgreSQL di macOS

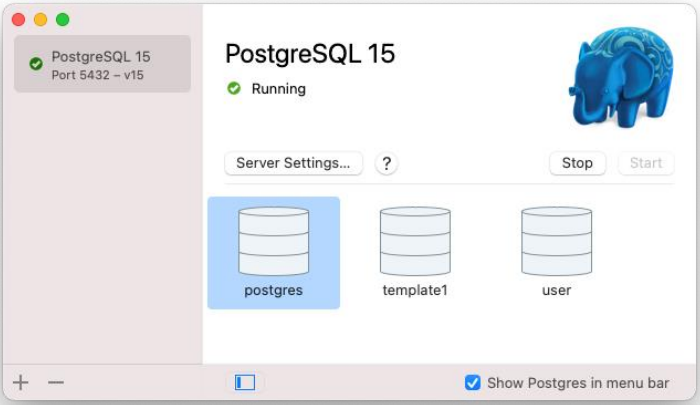
Untuk PostgreSQL di macOS, kita akan menggunakan Postgres.app, yang bisa didownload dari postgresapp.com. Pada saat buku ini ditulis, versi terbaru adalah 2.5.12 yang datang bersama PostgreSQL versi 15.1. Ukuran download adalah 93 MB. Kliklah file dmg hasil download. Setelah itu, draglah Postgres ke Applications, seperti contoh berikut:

Jalankanlah Postgres dari Launcher. Konfirmasi bahwa Postgres adalah aplikasi yang didownload dari Internet akan ditampilkan.

Kliklah Open. Setelah itu, Postgres.app akan dijalankan, seperti contoh layar berikut:



Kliklah tombol Initialize. Setelah selesai, tampilannya akan serupa dengan layar berikut:



Pastikanlah server telah berjalan. Klik Start untuk menjalankan server dan Stop untuk menghentikan server, apabila tidak lagi diperlukan. Klik ganda pada icon database dapat dilakukan untuk akses ke database. Pastikanlah Postgres.app memiliki hak untuk otomatisasi Terminal. Jalankanlah Terminal dan konfirmasi hak akses ini.

Pada instalasi yang penulis gunakan, program yang diperlukan untuk bekerja dengan sistem database tersimpan pada /Applications/Postgres.app/Contents/Versions/15/bin, dan oleh karenanya, penulis menyebutkan path lengkap ke program, seperti contoh berikut ketika membuat user test, dengan hak membuat database, dan password test:

```
/Applications/Postgres.app/Contents/Versions/15/bin
/createuser -d -P test
```

Demikian juga ketika membuat database test sebagai user test:

```
/Applications/Postgres.app/Contents/Versions/15/bin
/createdb -h localhost -U test test
```

Selain melakukan koneksi seperti yang dibahas dalam bab ini, kita juga dapat menggunakan client PostgreSQL psql, seperti perintah berikut:

```
/Applications/Postgres.app/Contents/Versions/15/bin
/psql -h localhost -U test test
```

Tentunya, saat ini, belum terdapat tabel. Anda dapat melihatnya dengan memberikan perintah \d (atau \dt), diikuti penekanan Enter. Untuk keluar dari psql, berikanlah perintah \q, diikuti Enter.

Instalasi PostgreSQL di Linux

Distribusi Linux yang digunakan dalam contoh ini adalah Ubuntu Linux versi 22.04. Cara instalasi PostgreSQL untuk versi-versi lain dari

Ubuntu ataupun Debian GNU/Linux akan serupa. PostgreSQL sendiri umum disertakan dalam berbagai distribusi Linux dan cara instalasinya akan serupa dengan instalasi paket lain yang disertakan resmi dalam distribusi-distribusi Linux tersebut.

Kita akan menggunakan command line interface. Untuk memulai, jalankanlah program terminal emulator pilihan Anda, dan berikanlah perintah berikut. Sebagai catatan, user sistem yang login perlu dapat menjalankan sudo dan mungkin diminta memasukkan password user tersebut.

```
sudo apt-get update
```

Tunggulah sampai selesai dan lanjutkan dengan perintah untuk instalasi PostgreSQL:

```
sudo apt-get install postgresql
```

Pada sistem yang penulis gunakan, ukuran download PostgreSQL 14.6 (server dan client) serta paket tambahannya adalah sekitar 40 MB dan setelah itu, diperlukan sekitar 160 MB untuk instalasi. Konfirmasilah instalasi dan tunggulah sampai selesai.

Setelah itu, jalankanlah perintah berikut sebagai user sistem postgres dengan sudo:

```
sudo -i -u postgres
```

Kemudian, kita jalankan psql untuk terhubung ke server PostgreSQL berjalan, dengan tujuan untuk mengatur password user database postgres:

Berikanlah perintah berikut, diikuti penekanan tombol Enter:

```
\password
```

Kemudian, isikanlah password dan konfirmasinya. Berikanlah perintah berikut diikuti Enter untuk keluar dari psql:

```
\q
```

Kemudian, kembalilah ke user sistem semula dengan perintah berikut:

```
exit
```

Kita akan membuat user database test, yang memiliki hak untuk membuat database, dengan password test. Perintah berikut akan meminta password dari user database postgres dan menentukan password untuk user database test.

```
createuser -h localhost -U postgres -P -d test
```

Setelah berhasil, sebagai user database test, kita akan membuat database test:

```
createdb -h localhost -U test test
```

Terakhir, kita mungkin ingin melakukan koneksi sebagai user database test, ke database test, dengan psql:

```
psql -h localhost -U test test
```

Saat ini, belum terdapat tabel. Anda dapat melihatnya dengan memberikan perintah \d (atau \dt), diikuti penekanan Enter. Untuk keluar dari psql, sama seperti sebelumnya, berikanlah perintah \q.

Catatan: Pembahasan-pembahasan berikutnya mengasumsikan modul db_util selalu telah di-load. Selain itu, juga diasumsikan koneksi database telah dilakukan sebelumnya, dengan nama variabel mencerminkan nama sistem database. Untuk Apache Derby, koneksi embedded akan digunakan, kecuali disebutkan berbeda.

Halaman ini sengaja dikosongkan

Contoh 3: Pembuatan Tabel dan Definisi Kolom

Tipe data SQL	**Tipe data Singkong**
CLOB	STRING atau Representasi STRING (apabila error)
Date Timestamp	DATE
Integer Bigint Decimal	NUMBER
Serial Bigserial Numeric	NUMBER
Null	NULL
Boolean Bool	BOOLEAN
(tipe lain)	STRING atau Representasi STRING

Mari kita awali bab ini dengan sebuah tabel. Bisa kita lihat bersama:

- Terdapat tipe data. Sebagaimana dibahas sebelumnya, perbedaan dengan file CSV diantaranya adalah bahwa tabel dalam sistem database relasional pada umumnya mengenal konsep tipe data. Dengan demikian, apabila kita memiliki kolom tanggal_lahir, bertipe date, maka diharapkan akan berisi nilai dengan tipe tersebut. Bahwa apakah hanya nilai dengan tipe yang kompatibel, yang dapat ditambahkan ke dalam tabel, adalah cerita berbeda (walau untuk buku ini, kita asumsikan demikian).

- Terdapat pemetaan antara tipe data SQL dan tipe data bahasa pemrograman Singkong. Dan, seperti dapat dilihat dalam tabel tersebut, beberapa sub tipe berbeda dapat dipetakan ke satu tipe di Singkong. Sebagai contoh, date dan timestamp dipetakan ke DATE. Ketika operasi database dilakukan:
    - Untuk query yang mendapatkan data dari database: interpreter Singkong akan mendapatkan tipe SQL dari nilai yang dikembalikan, dan memetakan ke tipe Singkong yang sesuai. Apabila tidak ditemukan yang sesuai, maka representasi STRING dikembalikan. Demikian juga apabila terjadi kesalahan dalam pemetaan tipe CLOB.
    - Untuk query dimana nilai dari Singkong dikirimkan, maka aturan pemetaan yang sama akan diterapkan.

Mengawali bab dengan tabel dan poin pembahasan memang dapat menambah jumlah halaman. Namun, tentunya bukan itu yang penulis butuhkan (kita masih di bab-bab awal, tapi masih banyak yang perlu dibahas, sementara jumlah halaman terbatas). Ketika membahas pembuatan tabel, kita perlu menentukan tipe data. Dan karena tipe dalam bahasa pemrograman dan tipe SQL mungkin berbeda (iya, untuk Singkong), kita perlu memahami pemetaan ini.

Setelah memahami hal tersebut, kita mendapatkan bonus. Bonus berupa kabar buruk, tapi lebih baik dikemukakan saja di depan. Dari tabel tersebut, kita tahu bahwa tipe selain yang didukung, akan dipetakan sebagai STRING (berdasarkan representasi nilai). Dengan demikian, terdapat batasan untuk tipe data yang dapat digunakan, ketika bekerja dengan bahasa Singkong (walau, mungkin ada jalan keluar, sebagaimana catatan di halaman berikut).

Catatan: Di Singkong, kita tidak dapat membuat tipe data sendiri. Tapi, kita memiliki tipe HASH, dimana key dan valuenya bisa berupa tipe apa saja.

Selain itu, kode program Singkong bisa memanggil method dalam class Java. Kita bisa lakukan apapun yang tidak bisa dilakukan di Singkong, di dalam method tersebut. Tipe kembalian dapat berupa salah satu dari tipe Singkong berikut:

- STRING
- ARRAY (dari STRING)
- ARRAY (dari ARRAY (dari STRING))
- NULL (error)

Dengan kombinasi antara nilai kembalian method dan fungsi eval di Singkong, kita mungkin bisa bekerja dengan tipe data SQL yang tidak didukung secara langsung di Singkong.

Bonus kabar buruk dan catatan yang mungkin tidak diperlukan? Sungguh merepotkan, Anda mungkin berpendapat. Tapi, penulis berjanji, setelah ini, pembahasan akan lebih mudah.

Pembuatan tabel dengan statement SQL

Pada dasarnya, pembuatan tabel dapat dilakukan dengan memberikan statement CREATE TABLE. Terdapat standar (sebagaimana halnya SQL, sejak SQL-86), namun sistem database dapat memiliki fitur spesifik.

Kita akan mencontohkan cara sulit terlebih dahulu, untuk Apache Derby dan PostgreSQL. Pastikanlah koneksi database telah berhasil dilakukan.

Tabel yang akan kita buat sederhana saja. Kita akan berikan nama contact, dengan dua kolom:

1. Nama kolom: contact_id, dengan tipe bilangan bulat. Nilainya akan ditambahkan otomatis setiap kali penambahan baris dilakukan.
2. Nama kolom: name, dengan tipe adalah deretan karakter dengan maksimum panjang 128.

Apache Derby

Pembuatan tabel tersebut pada Apache Derby akan menggunakan statement SQL berikut:

```
create table contact(contact_id integer not null generated always as identity (start with 1, increment by 1), name varchar(128))
```

Berikut adalah contoh kodenya pada Apache Derby embedded, terkoneksi ke database test, di direktori aktif:

```
>  load_module("db_util")

>  var d_embed = db_connect("embedded", "test", "", "")

>  d_embed

DATABASE (URL=jdbc:derby:test, user=, driver=org.apache.derby.jdbc.EmbeddedDriver)

>  var sql = "create table contact(contact_id integer not null generated always as identity (start with 1, increment by 1), name varchar(128))"

>  var r = db_query_simple(d_embed, sql)

>  r

[0]
```

Bisa kita lihat, variabel r adalah sebuah ARRAY, dengan nilai [0]. Ini artinya, bukan NULL, yang artinya, statement SQL tersebut berhasil dijalankan.

Apabila statement SQL tersebut dijalankan sekali lagi, mari kita lihat isi variabel r berikutnya:

```
>  var r = db_query_simple(d_embed, sql)

>  type(r)

"NULL"
```

Bisa kita lihat, pembuatan tabel yang sudah ada sebelumnya akan gagal.

PostgreSQL

Pembuatan tabel sebelumnya pada PostgreSQL dapat menggunakan statement SQL berikut:

```
create table contact(contact_id serial, name varchar(128))
```

Berikut adalah contoh kodenya, terkoneksi ke database test di localhost:

```
>  var d_pgsql = db_connect("postgresql", "//localhost/test", "test", "test")

>  d_pgsql

DATABASE (URL=jdbc:postgresql://localhost/test, user=test, driver=org.postgresql.Driver)

>  var sql = "create table contact(contact_id serial, name varchar(128))"
```

```
>  var r = db_query_simple(d_pgsql, sql)
>  r
[0]
```

Variabel r, sebuah ARRAY, dengan contoh isi [0], menandakan statement SQL tersebut berhasil dijalankan.

Mari kita jalankan statement SQL tersebut sekali lagi, dan variabel r akan bertipe NULL:

```
>  var r = db_query_simple(d_pgsql, sql)
>  type(r)
"NULL"
```

Bisa kita lihat, perintah SQL yang digunakan untuk kedua contoh tersebut berbeda. Pada sistem database lain, perintahnya juga mungkin akan berbeda.

Catatan: kita menggunakan fungsi bantu dari modul db_util seperti `db_query_simple` untuk menjalankan statement SQL. Fungsi ini pada akhirnya akan memanggil fungsi bawaan `query`, yang akan kita bahas pada bab berikutnya.

Pembuatan tabel dengan fungsi dari modul db_util

Modul db_util menyediakan sejumlah fungsi berikut, untuk pembuatan tabel:

db_create_table
db_create_table_

db_create_table_derby
db_create_table_derby_
db_create_table_embed
db_create_table_embed_
db_create_table_postgresql
db_create_table_postgresql_

Semua fungsi tersebut akan memanggil db_create_table_ (kecuali fungsi itu sendiri). Beberapa variasi disediakan untuk mempermudah. Pada dasarnya, ketika bekerja dengan Apache Derby atau PostgreSQL, kita cukup memanggil salah satu dari:

db_create_table_derby
db_create_table_embed
db_create_table_postgresql

Apa yang membuat fungsi-fungsi tersebut lebih mudah? Berbeda dengan perintah SQL langsung, fungsi-fungsi tersebut menyediakan abstraksi untuk tipe data tertentu, sebagaimana dirinci pada tabel berikut:

Tipe	**Apache Derby**	**PostgreSQL**
id	integer not null generated always as identity (start with 1, increment by 1)	serial
varchar	varchar (32672)	varchar
varchar.	varchar (32672) default ''	varchar default ''
text	clob	text
text.	clob default ''	text default ''
decimal	decimal(19,4)	decimal(19,4)
decimal.	decimal(19,4) default 0	decimal(19,4) default 0

integer	integer	integer
integer.	integer default 0	integer default 0
boolean	boolean	boolean
boolean.	boolean default false	boolean default false
date	date	date
date.	date default current_date	date default current_date
timestamp	timestamp	timestamp
timestamp.	timestamp default current_timestamp	timestamp default current_timestamp

Bisa kita lihat bersama, tipe yang diabstraksikan tersebut menyediakan nilai default, untuk yang namanya diakhiri dengan karakter titik. Selain, tersedia juga id, yang merupakan bilangan bulat yang nilainya akan ditambahkan otomatis ketika baris baru ditambahkan.

Dengan adanya abstraksi tersebut, kita bisa menggunakan definisi kolom yang sama untuk Apache Derby dan PostgreSQL. Sebagai contoh, untuk tabel book dengan kolom-kolom berikut:

- book_id: bilangan bulat yang nilainya akan ditambahkan otomatis setiap kali penambahan baris dilakukan.
- title: deretan karakter.
- note: deretan karakter dengan nilai default adalah ‘’.
- published: tanggal.

Mari kita sediakan definisi kolom tersebut dalam sebuah ARRAY (dari ARRAY):

```
>  var cols = [["book_id", "id"], ["title",
"varchar"], ["note", "varchar."], ["published",
"date"]]
```

Untuk Apache Derby, mari kita contohkan pada Apache Derby embedded, melanjutkan contoh sebelumnya:

```
>  var r = db_create_table_embed(d_embed, "book", cols)

>  r

[[0], "create table book (book_id integer not null generated always as identity (start with 1, increment by 1),title varchar (32672),note varchar (32672) default '',published date)"]
```

Nilai kembalian fungsi, selain contoh [0] yang menandakan statement SQL berhasil, juga dilengkapi statement lengkap yang dijalankan.

Variabel cols yang sama bisa digunakan untuk PostgreSQL:

```
>  var r = db_create_table_postgresql(d_pgsql, "book", cols)

>  r

[[0], "create table book (book_id serial,title varchar,note varchar default '',published date)"]
```

Bisa kita lihat, selama abstraksi tipe data yang tersedia cukup memenuhi kebutuhan, pembuatan tabel dapat dilakukan dengan lebih mudah, karena tidak perlu memberikan statement SQL secara langsung.

Tentu saja, constraint untuk tabel ataupun kolom, serta definisi lain tidak tersedia dengan penggunaan fungsi-fungsi bantu tersebut. Dengan SQL secara langsung, kita dapat melakukan pembuatan tabel sekompleks apapun yang disediakan oleh sistem database yang digunakan.

Catatan: bagaimana kalau kita hanya ingin mendapatkan statement SQL pembuatan tabel (tanpa menjalankannya), memanfaatkan abstraksi tipe yang disediakan? Kita bisa gunakan nama fungsi yang berakhiran _, yaitu:
db_create_table_derby_
db_create_table_embed_
db_create_table_postgresql_

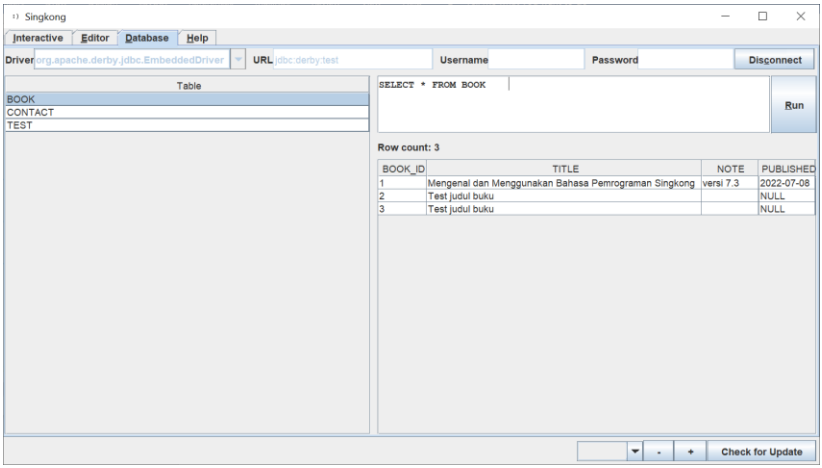
Contoh:

```
> var s = db_create_table_embed_(d_embed, "book", cols)
> s
["create table book (book_id integer not null generated always as identity (start with 1, increment by 1),title varchar (32672),note varchar (32672) default '',published date)", []]

> s[0]
"create table book (book_id integer not null generated always as identity (start with 1, increment by 1),title varchar (32672),note varchar (32672) default '',published date)"
```

Catatan: Setidaknya sampai Singkong versi 7.3, fungsi bantu `db_create_table_postgresql` tidak menambahkan constraint UNIQUE ataupun PRIMARY KEY untuk mencegah nilai duplikat, pada kolom "id".

Sementara, "id" bagi db_create_table_embed atau db_create_table_derby adalah `integer not null generated always as identity (start with 1, increment by 1)`. Pada Apache Derby, `generated always` adalah identify, akan unik, dan kita tidak bisa mengisikan nilainya.

Contoh 4: Menambahkan Baris ke Tabel



Untuk menambahkan baris ke tabel, kita menggunakan statement INSERT. Di Singkong, kita memiliki dua cara: memberikan statement SQL tersebut secara lengkap, atau dengan fungsi bantu yang disediakan oleh modul db_util. Untuk cara pertama, kita dapat menggunakan fungsi bawaan query, atau dengan fungsi bantu dari modul db_util. *(Sepertinya, kita mengawali bab ini dengan kalimat-kalimat yang rumit.)*

Mari kita gunakan fungsi bantu, dengan cara sesederhana mungkin, untuk menambahkan satu baris ke tabel book yang dibuat sebelumnya. Pastikanlah telah terkoneksi ke database. Contoh kode berikut dapat digunakan untuk Apache Derby ataupun PostgreSQL.

```
> var data = {"title": "Mengenal dan Menggunakan Bahasa Pemrograman Singkong", "note": "versi 7.3", "published": @20220708}
```

Kita menyediakan satu HASH, dengan key adalah nama kolom dalam STRING (dalam hal ini: “title”, “note”, dan “published”) dan value adalah nilai yang ingin ditambahkan, sesuai tipenya. Perhatikanlah

bahwa kolom published didefinisikan sebagai tipe date SQL, dan kita berikan nilai berupa DATE di Singkong (@20220708 dalam hal ini). Pemetaan antara tipe di Singkong dan di SQL akan dilakukan otomatis.

Apabila menggunakan Apache Derby (dalam contoh ini, versi embedded):

```
>  var r_derby = db_insert(d_embed, "book", data)
>  r_derby
[1]
```

Apabila menggunakan PostgreSQL:

```
>  var r_pgsql = db_insert(d_pgsql, "book", data)
>  r_pgsql
[1]
```

Kita memanggil fungsi db_insert dengan argumen DATABASE, STRING (nama tabel), dan HASH (baris yang ingin ditambahkan).

Fungsi akan mengembalikan jumlah baris (NUMBER) yang terefek oleh statement SQL yang dijalankan, dalam sebuah ARRAY. Apabila gagal dikerjakan, maka NULL akan dikembalikan.

Lihatlah, walaupun kita mengawali bab ini dengan kalimat rumit, menambahkan satu baris ke sebuah tabel hanya membutuhkan 1 sampai 2 baris.

Sekarang, mari kita lihat statement apa yang dijalankan. Dengan variabel data yang sama, tapi fungsi yang digunakan adalah db_insert_.

```
> db_insert_(d_embed, "book", data)

["insert into book (title,note,published) values (?,?,?)", ["Mengenal dan Menggunakan Bahasa Pemrograman Singkong", "versi 7.3", 2022-07-08 00:00:00]]
```

Fungsi tersebut mengembalikan sebuah ARRAY yang terdiri dari 2 elemen:

- Yang pertama adalah perintah SQL. Perhatikanlah bahwa kita tidak menggabungkan STRING dengan operator + misalnya. Kita ingin menambahkan baris ke tabel book, pada kolom `(title,note,published)`, namun dengan nilai `(?,?,?)`. Ini dituliskan apa adanya, dengan masing-masing ? akan digantikan dengan nilai aktual (yang mana, bisa didapatkan dari input oleh user, dan tidak sepenuhnya akan valid). Dengan demikian, kita tidak perlu memusingkan apakah akan berupa STRING, NUMBER, DATE, ataupun tipe lainnya.
- Yang kedua adalah sebuah ARRAY, dimana elemennya harus sesuai dengan jumlah kolom dan ? sebelumnya. Apabila terdapat tiga kolom (title, note, dan published) yang direpresentasikan dengan (?, ?, dan ?), maka dalam hal ini, kita lewatkan ARRAY `["Mengenal dan Menggunakan Bahasa Pemrograman Singkong", "versi 7.3", 2022-07-08 00:00:00]`. Elemen yang terakhir adalah representasi dari @20220708 dalam HASH data.

Kenapa ini kita bahas? Bukankah contoh pemanggilan db_insert sebelumnya dapat menutup bab ini dengan baik dan sempurna? Kenapa kita kembali membahas hal yang rumit? Jawabannya adalah

karena kita ingin menggunakan cara pertama yang disebutkan di awal bab.

Cara pertama, yaitu dengan statement SQL, dapat menggunakan fungsi bantu seperti:

db_query_single
db_query_simple
db_run_query

Fungsi db_query_simple telah kita contohkan pada pembuatan tabel, dan cocok digunakan apabila kita, sebagai contoh, ingin menjalankan statement SQL sederhana, tanpa input dari user dan parameter.

Karena terdapat parameter seperti kolom title, note, dan published yang diwakili oleh parameter ?, ?, dan ?, dalam hal ini, akan lebih cocok untuk menggunakan fungsi db_query_single. Fungsi ini akan menjalankan satu statement (STRING), yang dilengkapi parameter (ARRAY). Sebagai contoh:

```
>   var r = db_query_single(d_embed, "insert into
book(title) values(?)", ["Test judul buku"])
>  r
[1]

>   var r = db_query_single(d_pgsql, "insert into
book(title) values(?)", ["Test judul buku"])
>  r
[1]
```

Fungsi db_run_query tidak kita bahas. Fungsi ini dipanggil oleh db_insert (dan sejumlah fungsi lain). Berikut adalah isi selengkapnya (tanpa komentar dan dokumentasi) dari db_insert:

```
var db_insert = fn(a, b, c)
```

```
{
    var cmd = db_insert_(a, b, c)
    return db_run_query(a, cmd, false)
}
```

Bisa kita lihat, fungsi ini mendapatkan statement SQL dari db_insert_ dan memanggil db_run_query, yang selanjutnya akan memanggil fungsi bawaan query.

Akhirnya, kita sampai juga di fungsi bawaan query. Di Singkong, untuk menjalankan statement SQL, hanya terdapat satu fungsi bawaan, yaitu fungsi query tersebut. Semua fungsi bantu pada akhirnya akan memanggil fungsi ini.

Mari kita gunakan fungsi tersebut untuk menambahkan baris ke tabel:

```
> var r = query(d_embed, [ ["insert into book(title) values(?)", ["Test judul buku"]] ])
> r
[1]

> var r = query(d_pgsql, [ ["insert into book(title) values(?)", ["Test judul buku"]] ])
> r
[1]
```

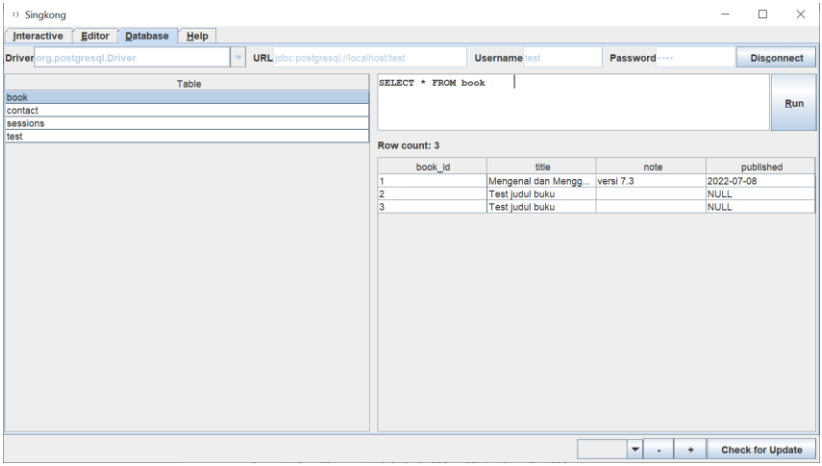
Perhatikanlah bahwa fungsi query ini menerima 2 argumen wajib, yaitu DATABASE dan ARRAY (dari ARRAY). Argumen keduanya, berdasarkan kode sebelumnya, dapat dijelaskan sebagai:

- Setiap statement diberikan sebagai sebuah ARRAY [statement, parameter]. Contoh kode sebelumnya yaitu: `["insert into book(title) values(?)", ["Test judul buku"]]`. Apabila statement tidak membutuhkan

parameter, maka lewatkanlah ARRAY kosong, sebagai contoh: [“insert …”, **[]**].

- Semua statement tersebut perlu kita simpan dalam sebuah ARRAY, sehingga menjadi **[** `["insert into book(title) values(?)", ["Test judul buku"]]` **]**.

Untuk melihat isi tabel book, kita dapat menggunakan database tool (tab Database ketika Singkong.jar dijalankan). Pilih (atau isikan) Driver, isikan URL (sesuai yang dibahas pada Contoh 2: Driver dan Koneksi Database), serta masukkan Username atau Password apabila diperlukan. Gambar pada awal bab ini adalah contoh untuk Apache Derby. Sementara, gambar berikut adalah contoh untuk PostgreSQL:



Seperti yang barangkali Anda duga, statement-statement yang dilewatkan pada fungsi query tidak harus sejenis. Bisa sederetan statement INSERT, UPDATE, dan lainnya.

Karena kita hanya membahas INSERT pada bab ini, perhatikanlah kode berikut:

```
> var r = query(d_embed, [ ["insert into book(title) values(?)", ["Test judul buku"]], ["Singkong", []] ])

> type(r)

"NULL"
> var r = query(d_pgsql, [ ["insert into book(title) values(?)", ["Test judul buku"]], ["Singkong", []] ])

> type(r)

"NULL"
```

Variabel r adalah NULL, yang artinya query gagal. Di log PostgreSQL, terdapat pesan kesalahan seperti berikut:

```
ERROR:  syntax error at or near "Singkong" at character 1

STATEMENT:  Singkong
```

Tentu saja, statement SQL “Singkong” tersebut gagal. Kalau berhasil, maka sistem databasenya rasanya perlu diragukan, bukan?

Yang perlu diperhatikan di sini adalah: ketika salah satu dari statement gagal, maka keseluruhan transaksi akan gagal. Kita bisa pastikan dengan database tool bahwa tidak ada baris yang bertambah pada tabel book.

Transaksi? Ya, ketika membuat nota penjualan misalnya, kita mungkin menulis ke dua tabel. Yang pertama adalah terkait nota tersebut, dan yang kedua adalah yang terkait rincian produk yang dijual. Apabila terjadi kesalahan dalam menulis ke rincian produk (tabel kedua), maka tentu saja, yang terkait nota penjualan (tabel pertama) perlu ikut tidak ditambahkan (walaupun sebelumnya telah berhasil).

Ini adalah alasan kenapa fungsi query didesain harus melewatkan semua statement dalam transaksi yang sama, sebagai sebuah ARRAY. Apabila bukan merupakan bagian dari suatu transaksi, panggillah query atau db_query_single berkali-kali. Ini juga sekaligus alasan kenapa terdapat fungsi bantu yang tugasnya hanya menghasilkan statement SQL (supaya, bisa diikutkan dalam transaksi).

Kita sudah mencapai akhir bab? Sayangnya, belum.

Terkadang, dengan tabel dimana terdapat kolom dengan nilai bilangan bulat yang bertambah otomatis ketika baris baru ditambahkan, kita perlu mengetahui berapa nilai yang ditambahkan tersebut (last insert id). Misal, pada tabel book, kita memiliki kolom book_id. Kita tidak pernah menentukan secara eksplisit nilai untuk kolom ini, ketika menambahkan baris baru. Untuk mendapatkan nilai ini, perhatikanlah contoh berikut:

```
>   var r = db_insert(d_embed, "book", {"title":
"test"})
>  r
[1]
>  db_last_embed(d_embed)
5

>   var r = db_insert(d_pgsql, "book", {"title":
"test"})
>  r
[1]
>  db_last_postgresql(d_pgsql)
5
```

Pada contoh tersebut, kita menambahkan satu baris, ke masing-masing Apache Derby dan PostgreSQL. Nilai dari book_id ketika

menambahkan baris baru tersebut adalah 5. Anda dapat menggunakan database tool untuk memastikan.

Row count: 4

book_id	title	note	published
1	Mengenal dan Menggunakan Bahasa Pemrograman Singkong	versi 7.3	2022-07...
2	Test judul buku		NULL
3	Test judul buku		NULL
5	test		NULL

Selain db_last_embed dan db_last_postgresql, fungsi lain yang tersedia adalah db_last_derby. Yang dilakukan fungsi db_last_embed hanyalah memanggil db_last_derby.

Setiap sistem database bisa menerapkan cara berbeda. Sebagai contoh, pada Apache Derby, fungsi bantu memberikan statement “values IDENTITY_VAL_LOCAL()” dan pada PostgreSQL, fungsi bantu memberikan "select LASTVAL()".

Akhirnya, bab ini selesai. Pembahasan kita memang campur aduk. Judulnya menambahkan baris ke tabel. Tapi, diawali pembukaan yang rumit, menggunakan fungsi bantu, melihat source code fungsi bantu, memahami fungsi bawaan query, menggunakan database tool, transaksi, dan ditutup dengan last insert id. Penulis berharap Anda tidak berhenti membaca sampai di sini, karena bab-bab berikutnya akan lebih nyaman dibahas.

Halaman ini sengaja dikosongkan

Contoh 5: Mendapatkan Isi Tabel

Di awal buku ini, kita melihat bahwa, untuk mendapatkan isi dari sebuah tabel, kita hanya perlu memanggil fungsi db_select_all dari modul db_util. Argumen yang dibutuhkan pun hanya dua: DATABASE dan STRING (nama tabel). Apabila berhasil, fungsi mengembalikan ARRAY dan NULL apabila sebaliknya.

Mari lihat tiga baris kode berikut:

```
>  var d_embed = db_connect_embed("test")

>  var r = db_select_all(d_embed, "book")

>  r
```

```
[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman Singkong", "versi 7.3", 2022-07-08 00:00:00], [2, "Test judul buku", "", null], [3, "Test judul buku", "", null], [5, "test", "", null]]]
```

Sebagaimana dilihat, r adalah sebuah ARRAY (dari ARRAY (dari ARRAY)):

- Setiap baris dalam tabel, misal yang diwarnai hijau, adalah sebuah ARRAY. Perhatikanlah bahwa aturan pemetaan tipe data antara SQL dan Singkong berlaku.
- Lalu, keseluruhan data yang dikembalikan, yaitu seluruh baris dalam tabel dalam contoh ini, juga berupa ARRAY (kurung diwarnai merah).
- Dan, pada akhirnya, hasil query selalu berupa sebuah ARRAY (diwarnai biru). Ketika menambahkan baris, nilai kembalian adalah jumlah baris yang terefek, juga dalam sebuah ARRAY, misal [1]. Ketika mendapatkan isi tabel, juga tetap ARRAY, sebagaimana yang diwarnai merah.

Dengan segera, Anda mungkin akan bertanya: untuk ARRAY setiap baris, indeks mana yang mewakili kolom apa? Pertanyaan tersebut perlu dijawab dalam dua tahap:

- Untuk mendapatkan isi dari tabel, kita menggunakan statement SELECT. Di tahap pertama, kita memilih apakah akan memberikan statement tersebut secara langsung, atau menggunakan fungsi bantu yang disediakan.
- Setelah itu, kita meminta agar hasil statement yang diberikan mencakup nama kolom.

Untuk tahap pertama, dalam bentuk sangat sederhana, statement SELECT dapat berupa misal: `select * from book`. Karena cukup sederhana, fungsi bantu berikut db_select_all_ mungkin tidak diperlukan. Tetap disajikan, karena variabel s berikut akan menampung statement yang diperlukan berikutnya.

```
>  var s = db_select_all_(d_embed, "book")

>  s

["select * from book ", []]
```

Tahap kedua melibatkan penggunaan fungsi query dengan argumen opsional ketiga. Kita belum membahas ini sebelumnya. Perhatikanlah apabila kita tidak meminta secara eksplisit:

```
>  var r = query(d_embed, [s])

>  r

[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman
Singkong", "versi 7.3", 2022-07-08 00:00:00], [2,
"Test judul buku", "", null], [3, "Test judul buku",
"", null], [5, "test", "", null]]]
```

Keluarannya sama dengan penggunaan fungsi db_select_all sebelumnya.

Apabila kita lewatkan true sebagai argumen ketiga untuk query, maka kita akan mendapatkan ARRAY berikut:

```
>  var r = query(d_embed, [s], true)

>  r

[[[["BOOK_ID", "INTEGER"], ["TITLE", "VARCHAR"],
["NOTE", "VARCHAR"], ["PUBLISHED", "DATE"]], [1,
"Mengenal dan Menggunakan Bahasa Pemrograman
Singkong", "versi 7.3", 2022-07-08 00:00:00], [2,
"Test judul buku", "", null], [3, "Test judul buku",
"", null], [5, "test", "", null]]]
```

Lihatlah, elemen pertama (diwarnai hijau) dari ARRAY seluruh baris dalam tabel (kurung diwarnai merah), adalah ARRAY dari ARRAY [kolom, tipe].

Dengan demikian, potongan kode berikut dapat digunakan untuk mendapatkan nama kolom dan isi tabel:

```
>  var c = r[0][0]

>  c

[["BOOK_ID", "INTEGER"], ["TITLE", "VARCHAR"],
["NOTE", "VARCHAR"], ["PUBLISHED", "DATE"]]
>  var rs = slice(r[0], 1, len(r[0]))

>  rs

[[1, "Mengenal dan Menggunakan Bahasa Pemrograman
Singkong", "versi 7.3", 2022-07-08 00:00:00], [2,
"Test judul buku", "", null], [3, "Test judul buku",
"", null], [5, "test", "", null]]
```

Dengan demikian, menampilkan dalam COMPONENT table cukup dengan sejumlah baris kode berikut. Kita mulai dengan mendapatkan daftar kolom. Langkah ini diperlukan, karena definisi daftar kolom dalam COMPONENT table adalah STRING dengan nama kolom dipisahkan koma. Potongan kode berikut, beserta penjelasannya, terdapat juga dalam buku-buku gratis: *Contoh dan Penjelasan Bahasa Singkong: Dasar-dasar Aplikasi GUI* dan *Contoh dan Penjelasan Bahasa Singkong: Mahir Bekerja dengan GUI*.

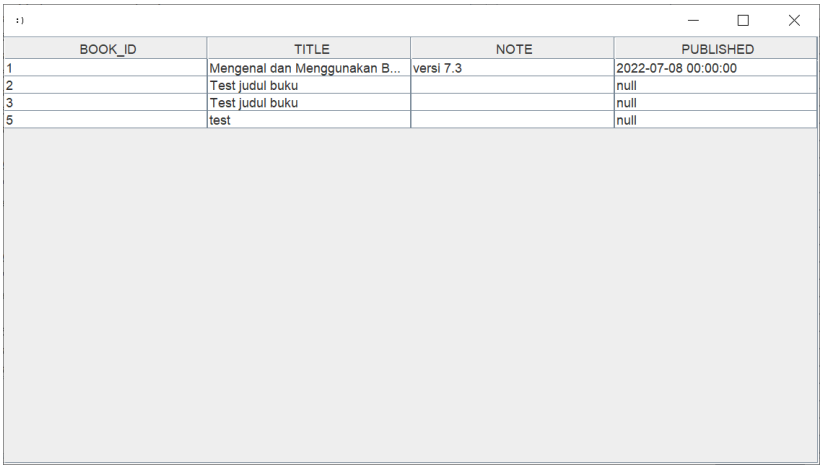
```
>  var h = [[]]
>  each(c, fn(e, i) { set(h, 0, h[0]+e[0]) })
>  h
[["BOOK_ID", "TITLE", "NOTE", "PUBLISHED"]]


>  var hl = join(",", h[0])
>  hl
"BOOK_ID,TITLE,NOTE,PUBLISHED"
```

Setelah itu, kita membuat COMPONENT table dan menambahkan ke frame:

```
>  var t = component("table", hl)
>  add(t)
```

Kemudian, kita tentukan isi COMPONENT table dengan config, dan menampilkannya:

```
>  config(t, "contents", rs)
>  show()
```



BOOK_ID	TITLE	NOTE	PUBLISHED
1	Mengenal dan Menggunakan B...	versi 7.3	2022-07-08 00:00:00
2	Test judul buku		null
3	Test judul buku		null
5	test		null

Sayangnya, kita belum bisa menutup bab ini. Kenapa? Yang pertama, kita mungkin tidak membutuhkan semua baris yang ada dalam suatu tabel. Lalu, juga tidak semua kolom. Setelah itu, kita mungkin akan urutkan dengan cara tertentu. Dari total baris yang kita dapatkan, kita mungkin akan batasi berapa sekaligus, mulai dari baris berapa. Dan, pada akhirnya, kita mungkin perlu mendapatkan hasil join dengan tabel lain.

Kita akan bahas semua alasan tersebut, kecuali yang hasil join, karena pada dasarnya, yang membedakan adalah statement SQL nya.

Menentukan daftar kolom

Pada statement SELECT, kita dapat menentukan daftar kolom (atau aliasnya) yang akan didapatkan. Di contoh sebelumnya, kita SELECT *, dimana * merepresentasikan semua kolom. Fokus pembahasan kita adalah kolom dalam satu tabel, namun tentunya SQL mendukung SELECT daftar kolom pada sejumlah tabel, termasuk hasil join.

Ketika bekerja dengan statement SQL tanpa fungsi bantu, hanya dengan fungsi bawaan query, kita bisa deretkan nama kolom seperti SELECT book_id, title. Perhatikanlah contoh berikut:

```
>  var d_embed = db_connect_embed("test")

>   query(d_embed, [ ["select book_id,title from book", []] ])

[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman Singkong"], [2, "Test judul buku"], [3, "Test judul buku"], [5, "test"]]]
```

Bisa kita lihat, yang dilakukan hanyalah mengganti * dengan book_id, title. Ingin yang lebih mudah? Mari kita gunakan fungsi bantu.

Untuk kepentingan daftar kolom saja, kita bisa gunakan fungsi bantu db_select (atau db_select_ yang mengembalikan statement SQL, diformat miring), seperti berikut:

```
> db_select_(d_embed, "book", ["book_id", "title"], [], "")

["select book_id,title from book ", []]

>  db_select(d_embed, "book", ["book_id", "title"], [], "")

[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman Singkong"], [2, "Test judul buku"], [3, "Test judul buku"], [5, "test"]]]
```

Argumen pertama (DATABASE) dan kedua (STRING nama tabel) cukup jelas. Sebenarnya, argumen ketiga, ARRAY kolom, juga jelas. Hanya saja, kita perlu berikan ARRAY kosong apabila ingin mendapatkan semua kolom. Untuk argumen ke-4 (ARRAY kosong) dan ke-5 (STRING kosong) bisa diabaikan untuk saat ini.

Fungsi-fungsi bantu berikutnya adalah fungsi yang spesifik terhadap sistem database tertentu. Sebagaimana disebutkan sebelumnya, sistem database relasional bisa menambahkan fitur ataupun sintaks tertentu yang tidak ada di sistem database lain. Salah satu yang terdampak dalam hal ini adalah statement SELECT. Apabila, seperti contoh ini, kita bekerja dengan Apache Derby, maka kita bisa menggunakan fungsi-fungsi:

db_select_derby_
db_select_derby
db_select_embed_
db_select_embed

Catatan: fungsi db_select_embed_ memanggil db_select_derby_. Fungsi yang diakhiri _ mengembalikan statement SQL seperti dicontohkan sebelumnya.

Apabila bekerja dengan PostgreSQL, maka kita bisa menggunakan:

db_select_postgresql_
db_select_postgresql

Kita akan membahas penggunaan fungsi-fungsi tersebut di akhir bab ini.

Memilih baris

Setelah kolom, rasanya sangat wajar apabila kita hanya ingin mendapatkan baris tertentu, sesuai kriteria yang kita tentukan. Apabila sebuah tabel mengandung jutaan baris, dan kita tidak membutuhkan semua baris tersebut, kita dapat tentukan kriterianya dengan ekspresi boolean (benar/salah) tertentu. Pada SQL, untuk

statement SELECT, kita menggunakan klausa WHERE. Sebagai contoh:

```
> query(d_embed, [ ["select book_id,title from book where book_id = 1", []] ])

[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman Singkong"]]]
```

Dalam contoh tersebut, ekspresinya adalah `book_id = 1`. Dengan demikian, karena book_id kita unik, maka maksimum jumlah baris yang dikembalikan adalah satu. Operator seperti OR dan AND juga dapat digunakan.

Ketika menggunakan fungsi bantu db_select (atau db_select_), klausa WHERE diberikan sebagai argumen ke-4. Sebelumnya, kita menggunakan ARRAY kosong. Untuk ekspresi `book_id = 1`, perhatikanlah contoh kode berikut:

```
> db_select(d_embed, "book", ["book_id", "title"], [ ["book_id =", 1, ""] ], "")

[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman Singkong"]]]
```

Masing-masing ekspresi diberikan sebagai ARRAY dengan tiga elemen:

- Kolom dan operator. Misal “book_id =”, atau “book_id >”, dan seterusnya.
- Nilai operan.
- STRING kosong atau operator untuk ekspresi berikutnya. Misal “AND” atau “OR”.

Berikut adalah contoh dengan lebih dari satu kriteria:

```
> db_select(d_embed, "book", ["book_id", "title"], [ ["book_id =", 1, "or"], ["title like ", "%Test%", ""] ], "")

[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman Singkong"], [2, "Test judul buku"], [3, "Test judul buku"]]]
```

Kriteria kita, yaitu book_id = 1 (`["book_id =", 1, "or"]`) atau title mengandung 'Test' `["title like ", "%Test%", ""]`, dengan operator LIKE, dimana % mewakili nol atau lebih karakter), tentunya akan mengembalikan, selain baris dengan book_id = 1, juga baris-baris yang titlenya mengandung 'Test'. Perhatikanlah ini case-sensitive, sehingga baris dengan book_id = 5 dan title = 'test', tidak dalam daftar baris yang didapatkan.

Catatan:
Query berikut akan gagal (NULL):

```
> query(d_embed, [ ["select book_id,title from book where book_id = 1 or title like %Test%", []] ])
```

Sementara, query berikut, akan berhasil:

```
> query(d_embed, [ ["select book_id,title from book where book_id = 1 or title like '%Test%'", []] ])
[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman Singkong"], [2, "Test judul buku"], [3, "Test judul buku"]]]
```

Apabila dicermati, perbedaannya hanyalah `title like %Test%` dan `title like '%Test%'` . Dengan statement SQL langsung,

apabila kutip diperlukan, maka harus diberikan. Dalam hal ini, terjadi kesalahan sintaks.

Perhatikanlah kebalikannya ketika menggunakan fungsi bantu. Query berikut tidak sesuai yang kita harapkan:

```
>  db_select(d_embed, "book", ["book_id",
"title"], [ ["book_id =", 1, "or"], ["title like
", "'%Test%'", ""] ], "")
[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman
Singkong"]]]
```

Perhatikanlah penggunaan kutip yang tidak diperlukan, pada bagian yang diformat tebal dan diwarnai merah.

Catatan lain: Ketika menggunakan PostgreSQL, ILIKE dapat digunakan untuk pencocokan secara tidak case-sensitive. Ini adalah ekstensi yang disediakan oleh PostgreSQL.

Contoh:

```
>  var d_pgsql = db_connect("postgresql",
"//localhost/test", "test", "test")
>  db_select(d_pgsql, "book", ["book_id",
"title"], [ ["book_id =", 1, "or"], ["title ilike
", "%Test%", ""] ], "")
[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman
Singkong"], [2, "Test judul buku"], [3, "Test
judul buku"], [5, "test"]]]
```

Pengurutan

Ketika menambahkan data satu buku ke dalam tabel book, sebuah buku yang ditambahkan belakangan mungkin saja merupakan buku yang terbit lebih dulu. Tidaklah selalu mungkin kita menambahkan buku sesuai urutan waktu terbitnya.

Kalaupun misal mungkin, tidaklah selalu juga bahwa ketika menampilkan isi tabel, harus selalu sesuai urutan waktu terbit. Kadang, kita ingin menampilkan sesuai urutan judul buku. Atau, waktu terbit, lalu judul buku.

Pada SQL, untuk statement SELECT, kita bisa menambahkan klausa ORDER BY, yang akan menentukan urutan. Dari sisi urutan tampil, kita bisa bagi menjadi kecil ke besar (ascending, ASC, default) atau sebaliknya (descending, DESC). Ada variasi terkait data NULL, yang akan kita singgung belakangan.

Lalu, tentu saja ada nama kolom yang akan diurutkan. Kombinasikan setiap nama kolom dengan ASC atau DESC, maka data dari tabel dapat ditampilkan terurut sesuai kriteria, bagaimanapun baris-baris ditambahkan sebelumnya.

Mari kita urutkan isi tabel book sesuai title secara ascending:

```
>  query(d_embed, [ ["select * from book order by title", []] ])

[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman Singkong", "versi 7.3", 2022-07-08 00:00:00], [3, "Test judul buku", "", null], [2, "Test judul buku", "", null], [5, "test", "", null]]]
```

Kemudian, secara descending:

```
>  query(d_embed, [ ["select * from book order by title desc", []] ])

[[[5, "test", "", null], [3, "Test judul buku", "", null], [2, "Test judul buku", "", null], [1, "Mengenal dan Menggunakan Bahasa Pemrograman Singkong", "versi 7.3", 2022-07-08 00:00:00]]]
```

Atau ketika lebih dari satu kolom:

```
>  query(d_embed, [ ["select * from book order by published asc, title desc", []] ])

[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman Singkong", "versi 7.3", 2022-07-08 00:00:00], [5, "test", "", null], [3, "Test judul buku", "", null], [2, "Test judul buku", "", null]]]
```

Catatan: Apabila tidak disebutkan secara eksplisit, opsi NULL FIRST berlaku apabila pengurutan dilakukan secara DESC. Dan, NULL LAST pada pengurutan ASC.

Ingin menggunakan fungsi bantu db_select atau db_select_? Kita akan menggunakan argumen ke-5 yang sebelum ini selalu berupa STRING kosong. Setelah menggunakan SQL secara langsung, tentunya ini bukan lagi hal baru. Contoh:

```
> db_select(d_embed, "book", [], [], "order by title desc")

[[[5, "test", "", null], [3, "Test judul buku", "", null], [2, "Test judul buku", "", null], [1, "Mengenal dan Menggunakan Bahasa Pemrograman Singkong", "versi 7.3", 2022-07-08 00:00:00]]]
```

Berapa baris sekaligus, dari mana

Bagian ini akan mengakhiri pembahasan kita. Akhirnya. Tapi, ini juga merupakan pembahasan yang setidaknya, bisa berbeda antar sistem database relasional yang digunakan.

Tabel book hanya ada beberapa baris. Ini tentu tidak jadi kendala ketika kita menampilkan isinya keseluruhan. Beda cerita kalau ada jutaan baris untuk ditampilkan semuanya, sekaligus. Terlepas dari

besarnya beban secara teknis, bagaimanapun kita mengurutkan, pengguna mungkin akan sulit membacanya.

Apa yang kita lakukan? Kita bisa tampilkan per sekian baris. Misal kalau ada 100 baris, kita tampilkan per sepuluh. Begitu pengguna pindah ke sepuluh baris berikut, jumlah yang diminta tampil tetap sama, hanya saja, dari posisi baris yang berbeda.

Pada Apache Derby, kita menggunakan klausa FETCH FIRST dan OFFSET. Pada PostgreSQL, kita dapat menggunakan klausa LIMIT dan OFFSET.

Mari kita contohkan pada tabel book kita. Walaupun terlalu sedikit, mari kita tampilkan dari 3 baris pertama. Pada Apache Derby (perhatikanlah yang diformat tebal):

```
>   db_select(d_embed, "book", [], [], "order by book_id fetch first 3 rows only")

[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman Singkong", "versi 7.3", 2022-07-08 00:00:00], [2, "Test judul buku", "", null], [3, "Test judul buku", "", null]]]
```

Pada PostgreSQL (juga yang diformat tebal):

```
>   db_select(d_pgsql, "book", [], [], "order by book_id fetch first 3 rows only")

[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman Singkong", "versi 7.3", 2022-07-08 00:00:00], [2, "Test judul buku", "", null], [3, "Test judul buku", "", null]]]
```

Loh, kok sama saja? Benar. Tapi, kita bisa gunakan LIMIT, seperti contoh berikut:

```
db_select(d_pgsql, "book", [], [], "order by book_id
limit 3")
```

```
[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman
Singkong", "versi 7.3", 2022-07-08 00:00:00], [2,
"Test judul buku", "", null], [3, "Test judul buku",
"", null]]]
```

Bagaimana kalau pengguna navigasi ke halaman berikutnya? Klausa OFFSET akan berperan (diformat tebal):

```
>   db_select(d_embed, "book", [], [], "order by
book_id offset 0 rows fetch first 3 rows only")
```

```
[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman
Singkong", "versi 7.3", 2022-07-08 00:00:00], [2,
"Test judul buku", "", null], [3, "Test judul buku",
"", null]]]
```

Misal kita berada pada halaman 1, pada halaman yang menampilkan 3 baris. Maka, offset dapat dihitung sebagai ((<halaman> - 1) * 3) atau ((1-1) * 3) atau (0 * 3) atau 0.

Ketika kita berada pada halaman 2, maka offset adalah ((2-1) * 3) atau (1 * 3) atau 3. Mari kita terapkan:

```
>   db_select(d_embed, "book", [], [], "order by
book_id offset 3 rows fetch first 3 rows only")
```

```
[[[5, "test", "", null]]]
```

Pada PostgreSQL, akan serupa:

```
>   db_select(d_pgsql, "book", [], [], "order by
book_id offset 0 limit 3")
```

```
[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman
Singkong", "versi 7.3", 2022-07-08 00:00:00], [2,
"Test judul buku", "", null], [3, "Test judul buku",
"", null]]]
```

```
>   db_select(d_pgsql, "book", [], [], "order by
book_id offset 3 limit 3")

[[[5, "test", "", null]]]
```

Catatan: Kita juga dapat menggunakan OFFSET tanpa FETCH FIRST. Namun, karena pembahasan kita adalah tentang menampilkan per berapa baris dalam sekian halaman, OFFSET kita bahas terakhir untuk pindah halaman.

Contoh:

```
>   db_select(d_pgsql, "book", [], [], "order by
book_id offset 1")
[[[2, "Test judul buku", "", null], [3, "Test judul
buku", "", null], [5, "test", "", null]]]

>   db_select(d_embed, "book", [], [], "order by
book_id offset 1 rows")
[[[2, "Test judul buku", "", null], [3, "Test judul
buku", "", null], [5, "test", "", null]]]
```

Kita sudah selesai dengan bab ini? Sayangnya, belum. Berikut adalah pembahasan terakhir kita.

Bagaimana kalau kita lupa dengan klausa FETCH FIRST? Bagaimana kalau kita ingin tampil sekian baris, mulai dari baris ke berapa, dengan cara lebih mudah? Sebagaimana disebutkan sebelumnya, kita bisa menggunakan fungsi-fungsi bantu spesifik sistem database tertentu.

Fungsi bantu spesifik sistem database

Fungsi db_select_derby, db_select_embed (yang memanggil db_select_derby), dan db_select_postgresql (beserta variasi dengan nama fungsi diakhiri _) berusaha mempermudah dengan:

- ORDER BY dalam bentuk ARRAY (argumen ke-5).
- OFFSET dalam NUMBER (argumen ke-6).
- FETCH FIRST dalam NUMBER (argumen ke-7).

Contoh pada Apache Derby:

```
> db_select_embed(d_embed, "book", [], [],
["book_id asc"], 0, 3)

[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman
Singkong", "versi 7.3", 2022-07-08 00:00:00], [2,
"Test judul buku", "", null], [3, "Test judul buku",
"", null]]]

> db_select_embed(d_embed, "book", [], [],
["book_id asc"], 3, 3)

[[[5, "test", "", null]]]
```

Dan, contoh pada PostgreSQL:

```
> db_select_postgresql(d_pgsql, "book", [], [],
["book_id asc"], 0, 3)

[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman
Singkong", "versi 7.3", 2022-07-08 00:00:00], [2,
"Test judul buku", "", null], [3, "Test judul
buku", "", null]]]

> db_select_postgresql(d_pgsql, "book", [], [],
["book_id asc"], 3, 3)

[[[5, "test", "", null]]]
```

Contoh 6: Mengubah Isi Tabel

Pada tabel book, terdapat satu buku dengan title ‘test’. Apabila kita ingin mengubah baris tersebut (misal title menjadi ‘testing’), kita perlu menentukan kriterianya terlebih dahulu. Dan, sebagaimana kita bahas di bab sebelumnya, kita gunakan klausa WHERE untuk kebutuhan tersebut. Yang kita tahu untuk baris tersebut adalah:

- book_id = 5
- title = ‘test’
- note = ‘’
- published = null

Apabila tidak terdapat baris lain dengan title = ‘test’, maka tentu dapat digunakan sebagai klausa WHERE. Yang artinya, semua baris dengan title = ‘test’ akan terdampak. Atau, kita bisa gunakan book_id = 5.

Sampai di sini, mari kita bahas sesuatu yang sulit dahulu. Kenapa? Karena bab ini akan singkat. Jadi, kita mengulang sedikit catatan dari contoh 3.

Kolom book_id pada tabel book, pada Apache Derby, didefinisikan sebagai: `integer not null generated always as identity (start with 1, increment by 1)`. Pada Apache Derby, `generated always` adalah identify, akan unik, dan kita tidak bisa mengisikan nilainya. Apabila kita isikan, maka contoh pesan kesalahan akan mengandung ‘Attemp to modify an identity column’. Jadi, kriteria book_id = 5 akan pasti merujuk pada baris buku yang kita maksud.

Sementara, book_id pada tabel book, pada Apache Derby, didefinisikan sebagai `serial`. Dalam hal ini, fungsi bantu `db_create_table_postgresql` tidak menambahkan constraint UNIQUE ataupun PRIMARY KEY untuk mencegah nilai

duplikat. Jadi, book_id = 5 bisa saja merujuk pada baris lain (apabila diisikan secara eksplisit demikian). Memang tidak, untuk tabel book kita di PostgreSQL. Tapi, tanpa constraint yang disebut sebelumnya, kita tidak asumsikan nilai unik.

Kembali ke klausa WHERE, sebagaimana dicontohkan di bab sebelumnya, tentu kita bisa kombinasikan dengan operator misal OR atau AND. Jadi, book_id = 5 AND title = 'test' misalnya.

Statement yang digunakan adalah UPDATE. Dengan demikian, tanpa bertele-tele lagi, dengan SQL langsung, berikut adalah statement selengkapnya (dengan WHERE title = 'test') pada Apache Derby:

```
>  query(d_embed, [ ["update book set title=? where title=?", ["testing", "test"]] ])
```

```
[1]
```

Pada PostgreSQL, yang berbeda hanya argumen DATABASE nya:

```
>  query(d_pgsql, [ ["update book set title=? where title=?", ["testing", "test"]] ])
```

```
[1]
```

Isi tabel telah berubah:

```
>  db_select(d_embed, "book", [], [["book_id = ", 5, ""]], "")
```

```
[[[5, "testing", "", null]]]
```

```
>  db_select(d_pgsql, "book", [], [["book_id = ", 5, ""]], "")
```

```
[[[5, "testing", "", null]]]
```

Cobalah jalankan UPDATE sekali lagi untuk kedua sistem database tersebut:

```
> query(d_embed, [ ["update book set title=? where title=?", ["testing", "test"]] ])
```

```
[0]
```

```
> query(d_pgsql, [ ["update book set title=? where title=?", ["testing", "test"]] ])
```

```
[0]
```

Perhatikanlah kembalian dari pemanggilan fungsi-fungsi tersebut, yang diformat tebal, yang merupakan jumlah baris yang terdampak. Pertama, tentu ada 1 baris yang terdampak. Tapi, setelah title pada baris tersebut berubah menjadi ‘testing’, tidak ada lagi baris dengan title = ‘test’. Dengan demikian, pada contoh terakhir, tidak ada lagi baris yang terdampak.

Bab ini selesai sampai di sini? Sekali lagi, tidak. Masih ada fungsi bantu yang mungkin bisa mempermudah, yaitu db_update (dan db_update_). Fungsi-fungsi tersebut membutuhkan:

- DATABASE
- STRING (nama tabel)
- ARRAY (klausa WHERE)
- HASH (nilai baru, dimana key adalah kolom (dalam STRING) dan value adalah nilai).

Mari kita gunakan baris yang sama, untuk mengubah title kembali menjadi ‘test’, dengan penggunaan fungsi bantu.

```
> db_update(d_embed, "book", [ ["title = ", "testing", ""] ], {"title": "test"})
```

```
[1]
```

```
>    db_update(d_pgsql,  "book",  [  ["title  =  ",
"testing", ""] ], {"title": "test"})

[1]
```

Barangkali lebih mudah. Tapi, ada satu hal yang perlu diperhatikan di sini. Pada HASH nilai baru, pastikanlah key diberikan sebagai STRING. Jadi, pada contoh sebelumnya, key adalah “title” dan bukan title (yang merupakan fungsi bawaan).

Terakhir, kita bisa saja mengubah keseluruhan isi tabel, dengan cara tidak memberikan klausa WHERE. Lalu, kita juga bisa mengubah nilai untuk beberapa kolom sekaligus.

Sampai di sini, pembahasan selesai. Sebagai bonus, contoh berikutnya akan lebih singkat.

Contoh 7: Menghapus Isi Tabel dan Tabelnya

Sebagai contoh, kita ingin menghapus baris yang dibahas pada bab sebelumnya, yaitu book_id = 5 atau title = 'test'. Klausa WHERE akan sama, sehingga kita tidak perlu mengulang pembahasan.

Statement yang akan digunakan adalah DELETE. Perhatikanlah contoh berikut:

```
>    query(d_embed,  [  ["delete  from  book  where
title=?", ["test"]] ])

[1]
>    query(d_pgsql,  [  ["delete  from  book  where
title=?", ["test"]] ])

[1]
```

Bisa kita lihat, satu baris terdampak. Kini, baris tersebut tidak ada lagi:

```
>   db_select(d_embed,  "book",  [],  [["title  =  ",
"test", ""]], "")

[[]]
>   db_select(d_pgsql,  "book",  [],  [["title  =  ",
"test", ""]], "")

[[]]
```

Sebagaimana contoh sebelumnya, kita masih punya fungsi bantu yaitu: db_delete (dan db_delete_) yang membutuhkan:

- DATABASE
- STRING (nama tabel)
- ARRAY (klausa WHERE)

Mari kita hapus baris-baris yang mengandung ‘Test’:

```
>    db_delete(d_embed,  "book",  [  ["title  like",
"%Test%", ""] ])

[2]
>    db_delete(d_pgsql,  "book",  [  ["title  like",
"%Test%", ""] ])

[2]
```

Dua baris terdampak karena kita memang memiliki dua baris dengan title “Test judul buku”. Kini, tabel book hanya berisi satu baris:

```
>  db_select_all(d_embed, "book")

[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman
Singkong", "versi 7.3", 2022-07-08 00:00:00]]]

>  db_select_all(d_pgsql, "book")

[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman
Singkong", "versi 7.3", 2022-07-08 00:00:00]]]
```

Kita juga bisa menghapus keseluruhan isi tabel, dengan cara tidak memberikan klausa WHERE.

Dan, terakhir, judul bab ini menyebutkan isi tabel dan tabelnya. Walaupun menghapus keseluruhan isinya, kita tidak menghapus tabelnya itu sendiri. Untuk menghapus tabelnya, statement yang digunakan adalah DROP TABLE. Contoh pada tabel contact:

```
>  query(d_embed, [ ["drop table contact", []] ])

[0]

>  query(d_pgsql, [ ["drop table contact", []] ])

[0]
```

Contoh 8: Nilai Kembalian Fungsi Query

Ini adalah bab terakhir kita, yang mungkin akan berguna apabila kita perlu mengetahui apa yang dapat kita lakukan dengan hasil query. Spesifiknya adalah pada pemanggilan fungsi bawaan query.

Sebagai contoh, apabila hasilnya adalah berupa isi tabel, kita bisa tampilkan. Apabila hasilnya adalah berapa baris yang terdampak, kita bisa informasikan. Tampaknya lebih mudah daripada memeriksa statement yang diberikan.

Hasil query, setelah dijalankan, akan berupa ARRAY atau NULL. Yang terakhir adalah ketika terjadi kesalahan. Sementara, ketika berhasil, ARRAY dimaksudkan sebagai ARRAY dari setiap statement yang diberikan (yang dapat berupa ARRAY atau NUMBER baris terdampak). Kita tahu bahwa fungsi query menerima ARRAY dari statement, dengan setiap statement dapat diberikan parameter. Perhatikanlah contoh berikut:

```
> query(d_embed, [ ["select * from book", []],
["select book_id, title from book where book_id=?",
[1]] ])

[ [[1, "Mengenal dan Menggunakan Bahasa Pemrograman
Singkong", "versi 7.3", 2022-07-08 00:00:00]], [[1,
"Mengenal dan Menggunakan Bahasa Pemrograman
Singkong"]] ]
```

Dapat kita lihat:

- Diwarnai merah: ARRAY hasil kembalian dari query karena berhasil.
- Diwarnai hijau: ARRAY dari `["select * from book", []]`, yang mana, karena isi tabel, juga berupa ARRAY dengan

contoh elemen: `[1, "Mengenal dan Menggunakan Bahasa Pemrograman Singkong", "versi 7.3", 2022-07-08 00:00:00]`

- Diwarnai biru: penjelasan berlaku seperti pada poin sebelumnya. Kita melihat yang sama ketika membahas bagaimana mendapatkan isi tabel.

Sekarang, bagaimana kalau array STATEMENT tidak memberikan hasil yang sejenis? Perhatikanlah contoh berikut:

```
>   query(d_embed, [ ["select * from book", []], ["delete from book where book_id=?", [5]] ])

[[[1, "Mengenal dan Menggunakan Bahasa Pemrograman Singkong", "versi 7.3", 2022-07-08 00:00:00]], 0]
```

Pada bagian yang diformat tabel, bisa kita lihat bahwa kembaliannya adalah berupa NUMBER. Kita melihat contoh ini misalnya pada menambahkan baris atau mengubah/menghapus isi tabel.

Sekarang, bagaimana kalau argumen ketiga dari query adalah true, untuk statement-statement yang mengembalikan ARRAY?

```
>   query(d_embed, [ ["select * from book", []], ["select book_id, title from book where book_id=?", [1]] ], true)

[[[["BOOK_ID", "INTEGER"], ["TITLE", "VARCHAR"], ["NOTE", "VARCHAR"], ["PUBLISHED", "DATE"]], [1, "Mengenal dan Menggunakan Bahasa Pemrograman Singkong", "versi 7.3", 2022-07-08 00:00:00]], [[["BOOK_ID", "INTEGER"], ["TITLE", "VARCHAR"]], [1, "Mengenal dan Menggunakan Bahasa Pemrograman Singkong"]]]
```

```
>   query(d_pgsql, [ ["select * from book", []],
["select book_id, title from book where book_id=?",
[1]] ], true)
```

```
[[[["book_id", "serial"], ["title", "varchar"],
["note", "varchar"], ["published", "date"]], [1,
"Mengenal dan Menggunakan Bahasa Pemrograman
Singkong", "versi 7.3", 2022-07-08 00:00:00]],
[[["book_id", "serial"], ["title", "varchar"]], [1,
"Mengenal dan Menggunakan Bahasa Pemrograman
Singkong"]]]
```

Kita sudah pernah membahas ini ketika mendapatkan isi tabel. Bahwa elemen pertama adalah definisi kolom.

Tentu saja, untuk query yang mengembalikan NUMBER, argumen true tersebut juga tetap relevan:

```
>   query(d_embed, [ ["select * from book", []],
["delete from book where book_id=?", [5]] ], true)
```

```
[[[["BOOK_ID", "INTEGER"], ["TITLE", "VARCHAR"],
["NOTE", "VARCHAR"], ["PUBLISHED", "DATE"]], [1,
"Mengenal dan Menggunakan Bahasa Pemrograman
Singkong", "versi 7.3", 2022-07-08 00:00:00]], 0]
```

```
>   query(d_pgsql, [ ["select * from book", []],
["delete from book where book_id=?", [5]] ], true)
```

```
[[[["book_id", "serial"], ["title", "varchar"],
["note", "varchar"], ["published", "date"]], [1,
"Mengenal dan Menggunakan Bahasa Pemrograman
Singkong", "versi 7.3", 2022-07-08 00:00:00]], 0]
```

Tetap 0, sebuah NUMBER, bukan ARRAY.

Jadi, satu atau lebih statement, terlepas dari mendapatkan definisi kolom atau tidak, kita selalu bisa mengandalkan tipe dari masing-

masing elemen ARRAY ketika query berhasil. Sehingga, apabila kita iterasi setiap elemen tersebut, kita bisa dapatkan tipenya:

```
>  var r = query(d_embed, [ ["select * from book", []], ["delete from book where book_id=?", [5]] ], true)

>  each(r, fn(e, i) { println(type(e)) })

ARRAY

NUMBER
```

Seru, bukan? Satu atau lebih statement, apabila NUMBER, kita cukup informasikan berapa baris terdampak. Ketika berupa ARRAY, apabila disertakan definisi kolom, kita bisa set kolom pada COMPONENT tabel terlebih dahulu berdasarkan elemen pertama dalam ARRAY. Lalu tampilkan isinya.

Daftar Pustaka

Captain, F. A. (2013). *Six-Step Relational Database Design: A Step By Step Approach to Relational Database Design and Development*.

Noprianto. (2020). *Mengenal dan Menggunakan Bahasa Pemrograman Singkong*. PT. Stabil Standar Sinergi.

Noprianto, Iskandar, K., & Soewito, B. (2022). *Contoh dan Penjelasan Bahasa Singkong: Dasar-dasar Aplikasi GUI.* PT. Stabil Standar Sinergi.

Noprianto, Wartika, & Gaol, F. L. (2022). *Contoh dan Penjelasan Bahasa Singkong: Mahir Bekerja dengan GUI.* PT. Stabil Standar Sinergi.

Preece, J., Sharp, H., & Rogers, Y. (2015). *Interaction design: beyond human-computer interaction*. John Wiley & Sons.

The Apache Software Foundation. (2014). *Derby Reference Manual*. https://db.apache.org/derby/docs/10.10/ref/index.html

The PostgreSQL Global Development Group. (2021). *PostgreSQL 9.6.24 Documentation*. https://www.postgresql.org/docs/9.6/index.html

The PostgreSQL Global Development Group. (2023). *PostgreSQL 15.2 Documentation.* https://www.postgresql.org/docs/15/index.html

Buku ini berisi sejumlah contoh source code dan penjelasan langkah demi langkah yang mudah dipahami untuk membuat program komputer yang terhubung ke sistem database relasional, dengan bahasa pemrograman Singkong.

Contoh-contoh yang dibahas mencakup pengenalan sistem database relasional, koneksi, query, dan topik lanjutan. Anda tidak perlu memahami database untuk dapat mengikuti pembahasan dalam buku ini. Akan tetapi, sebagai buku lanjutan, pemahaman akan bahasa Singkong dan bekerja dengan GUI akan membantu.

Melengkapi contoh program, buku ini juga membahas pengantar untuk interaksi manusia dan komputer.

Dr. Noprianto mengembangkan bahasa pemrograman Singkong dan interpreternya sejak akhir 2019.

Beliau menyukai pemrograman, dan mendirikan serta mengelola perusahaan pengembangan software dan teknologi Singkong.dev (PT. Stabil Standar Sinergi).

Noprianto menyelesaikan pendidikan doktor ilmu komputer dan telah menulis beberapa buku pemrograman (termasuk Python, Java, dan Singkong).

Buku dan softwarenya dapat didownload dari https://nopri.github.io

Dr. Maria Seraphina Astriani meraih gelar Doktor di bidang Computer Science serta memiliki pengalaman profesional dan membantu berbagai perusahaan di Indonesia maupun manca negara khususnya di bidang solusi Information Technology (IT) dengan menggunakan teknologi website/desktop/mobile.

Minat penelitiannya meliputi human-computer interaction, IT solution, machine learning, dan pattern recognition.

Beliau juga membagikan pengalaman di dalam dunia IT dengan mengajar sejak tahun 2010.

Dr. Fredy Purnomo, S.Kom., M.Kom., lahir di Blora, adalah lulusan program studi Doctor of Computer Science dari Binus University, program studi Magister Teknologi Informasi Universitas Indonesia, dan program studi Teknik Informatika dari Binus University.

Beliau berkarir sebagai Faculty Member sejak tahun 1999 di Teknik Informatika dan Sistem Informasi. Saat ini, beliau menjabat sebagai Dekan School of Computer Science di Universitas Bina Nusantara.

Penelitiannya berfokus pada area Multimedia, UI/UX, Game Development, dan Smart City.

singkong.dev
Alamat: Puri Indah Financial Tower
Lantai 6, Unit 0612
Jl. Puri Lingkar Dalam Blok T8
Kembangan, Jakarta Barat 11610
Email: info@singkong.dev

ISBN 978-602-52770-5-4

9 786025 277054

Rp. 50.000