

Andre Pratama

MySQL Uncover

Panduan Belajar MySQL & MariaDB untuk Pemula

DuniaIlkom

MySQL Uncover

Panduan Belajar MySQL dan MariaDB untuk Pemula

DuniaIlkom

Buku ini dijual di <http://leanpub.com/htmluncover>

Versi ini diterbitkan pada 2017-12-16



Ini adalah sebuah buku [Leanpub](#). Leanpub memberdayakan penulis dan penerbit dengan proses Lean Publishing. [Lean Publishing](#) adalah model penerbitan ebook dalam-proses menggunakan piranti ringan dan sejumlah iterasi untuk memperoleh masukan dari pembaca, menerapkan pivot hingga Anda dapat mewujudkan komposisi buku yang pas dan menarik.

© 2015 - 2017 DuniaIlkom

Contents

Ucapan Terimakasih	i
Tentang Penulis	ii
Kata Pengantar	iii
Asumsi / Pengetahuan Dasar	v
Contoh Kode Program	vi
1. Berkenalan dengan MySQL	1
1.1 Pengertian Database	1
1.2 Pengertian Database Model	2
1.3 Pengertian Relational Model	3
1.4 RDBMS : Relational Database Management Systems	4
1.5 MySQL sebagai RDBMS	5
1.6 MySQL atau MariaDB?	7
2. Sejarah Perkembangan MySQL dan MariaDB	9
2.1 Kisah tentang UNIREG dan TcX	9
2.2 MySQL AB dan Open Source	11
2.3 MySQL AB sebagai “Start Up”	12
2.4 MySQL Sebagai Perusahaan Global	14
2.5 IPO yang Batal dan Akuisisi oleh Sun Microsystem	16
2.6 Raja Database yang Mulai Mendekat: Oracle	16
2.7 Gerakan “Save MySQL”	17
2.8 MySQL di bawah Oracle	18
2.9 MariaDB: Sebuah Harapan Baru	18
2.10 MariaDB Foundation dan MariaDB Corporation	19
2.11 Jadi, pilih MySQL atau MariaDB?	21
2.12 Bagaimana dengan MySQL?	21
3. Pengantar Teori Database dan SQL	23
3.1 Database, Attribute, dan Tuple	23
3.2 Candidate Key, Primary Key dan Foreign Key	24
3.3 Referential Integrity	26
3.4 Normalisasi Database	26
3.5 Entity Relationship Diagram (ERD)	27

CONTENTS

3.6	Table Index	28
3.7	Pengertian Structured Query Language (SQL)	28
3.8	Sejarah Structured Query Language (SQL)	28
3.9	Jenis-jenis Perintah Bahasa SQL	30
4.	Instalasi XAMPP, MySQL dan MariaDB	31
4.1	Instalasi XAMPP	31
4.2	Instalasi MySQL	38
4.3	Instalasi MariaDB	46
5.	MySQL Server dan MySQL Client	53
5.1	Folder Instalasi MySQL	53
5.2	File Setingan MySQL: my.ini	57
5.3	Arsitektur Client-Server MySQL	58
5.4	Menjalankan MySQL Server (mysqld.exe)	59
5.5	Menjalankan MySQL Server sebagai Windows Service	61
5.6	Menjalankan MySQL Client	65
5.7	Membuat Shorcut untuk Folder mysql/bin	69
6.	Dasar Bahasa Query MySQL	72
6.1	MySQL Manual	73
6.2	Aturan Penulisan Query MySQL	74
6.3	Membuat dan Menghapus Database	78
6.4	Membuat dan Menghapus Tabel	81
6.5	Menambahkan Data ke Tabel	84
6.6	Mengubah Data Tabel	87
6.7	Menghapus Data Tabel	88
6.8	Menampilkan Data Table	88
7.	Database, Character set dan Collation	92
7.1	Pengertian Character set dan Collation	92
7.2	ASCII Character Set	94
7.3	Extended-ASCII Character Set	96
7.4	Unicode Character Set	98
7.5	Character Set di dalam MySQL	100
7.6	Query Pembuatan Database	102
7.7	Mengubah Character set dan Collation	104
7.8	Folder Penyimpanan Database	105
8.	Tipe Data MySQL	106
8.1	Tipe Data Numeric	106
8.2	Tipe Data Integer	108
8.3	Tipe Data Decimal	115
8.4	Tipe Data Float dan Double	118
8.5	Tipe Data Bit	122
8.6	Tipe Data String	124
8.7	Tipe Data Char dan Varchar	127

CONTENTS

8.8	Tipe Data Binary dan Varbinary	130
8.9	Tipe Data Text	132
8.10	Tipe Data Blob	133
8.11	Tipe Data Date	133
8.12	Tipe Data Enum	138
8.13	Tipe Data Set	140
8.14	Latihan Tipe Data	142
8.15	Jawaban Latihan Tipe Data	144
9.	Atribut Tipe Data	147
9.1	Atribut NULL	147
9.2	Atribut NOT NULL	148
9.3	Atribut DEFAULT	150
9.4	Atribut SIGNED	153
9.5	Atribut UNSIGNED	154
9.6	Atribut ZEROFILL	154
9.7	Atribut CHARACTER SET dan COLLATION	155
9.8	Atribut BINARY	161
9.9	Atribut ON UPDATE	162
9.10	Atribut AUTO_INCREMENT	166
9.11	Atribut PRIMARY KEY	171
9.12	Atribut UNIQUE	172
9.13	Menggabungkan Atribut	174
10.	Server SQL Mode	176
10.1	Level Penerapan SQL Mode	176
10.2	Strict Mode	178
10.3	Menginput Lebih dari 1 Mode	182
10.4	Mengubah SQL Mode dari my.ini	182
11.	Membuat, Mengubah dan Menghapus Tabel	185
11.1	Format Dasar CREATE TABLE	185
11.2	CREATE TABLE ... IF NOT EXIST	193
11.3	CREATE TEMPORARY TABLE	194
11.4	CREATE TABLE ... LIKE	195
11.5	CREATE TABLE ... SELECT	197
11.6	Table Cloning	199
11.7	Table Character Set dan Collation	200
11.8	Storage Engine	201
11.9	TABLE OPTION ... AUTO_INCREMENT	211
11.10	TABLE OPTION ... COMMENT	212
11.11	Format Dasar ALTER TABLE	213
11.12	ALTER TABLE ... RENAME	213
11.13	RENAME TABLE ... TO	216
11.14	ALTER TABLE ... TABLE OPTION	217
11.15	ALTER TABLE ... MODIFY	219

CONTENTS

11.16	ALTER TABLE ... CHANGE	220
11.17	ALTER TABLE ... ADD COLUMN	224
11.18	ALTER TABLE ... DROP COLUMNS	227
11.19	Format Dasar DROP TABLE	228
11.20	DROP TABLE	228
11.21	Latihan Pembuatan Tabel	229
11.22	Jawaban Latihan Pembuatan Tabel	230
12.	Menginput Data Tabel	232
12.1	INSERT ... VALUE	232
12.2	INSERT ... SET	237
12.3	INSERT ... SELECT	239
12.4	ON DUPLICATE KEY UPDATE	241
12.5	Latihan Menginput Data Tabel	243
12.6	Jawaban Menginput Data Tabel	245
13.	Menampilkan Data Tabel	248
13.1	SELECT ... select_expression	248
13.2	SELECT ... FROM	251
13.3	SELECT ... AS	253
13.4	SELECT ... ORDER BY	255
13.5	SELECT ... LIMIT	260
13.6	SELECT ... WHERE	265
13.7	SELECT ... WHERE table1 = table2	274
13.8	SELECT ... WHERE ... IN	280
13.9	SELECT ... ANY, SELECT ... SOME, SELECT ... ALL	283
13.10	SELECT ... EXISTS	287
13.11	SELECT ... WHERE ... BETWEEN	288
13.12	SELECT ... WHERE ... LIKE	291
13.13	SELECT ... WHERE ... REGEXP	295
13.14	SELECT ... UNION	304
13.15	SELECT DISTINCT	309
13.16	SELECT COUNT(), MAX(), MIN(), AVG() dan SUM()	312
13.17	SELECT ... GROUP BY	317
13.18	SELECT ... JOIN	323
14.	Mengupdate dan Menghapus Data Tabel	335
14.1	UPDATE	336
14.2	UPDATE IGNORE	343
14.3	REPLACE	344
14.4	DELETE	346
14.5	TRUNCATE	349
15.	Index dan FullText Search	352
15.1	Pengertian Index	352
15.2	Cara Pembuatan Index	355

CONTENTS

15.3	Cara Menghapus Index	360
15.4	Jenis-jenis Index	360
15.5	Index Prefix	362
15.6	Multi-Column Index	363
15.7	Query EXPLAIN	364
15.8	FULLTEXT Index	368
15.9	Jenis Pencarian FULLTEXT Index	372
15.10	Multi-Column FULLTEXT Index	374
16.	Transaction dan Table Lock	377
16.1	Pengertian Transaction	377
16.2	Prinsip Kerja Transaction	378
16.3	Cara Menjalankan Transaction	378
16.4	Table Lock	392
16.5	Multiple Table Lock	398
17.	Primary Key dan Referential Integrity	401
17.1	Pengertian Primary Key	401
17.2	Membuat Primary Key	401
17.3	Menghapus Primary Key	404
17.4	Sifat Primary Key	405
17.5	Composite Primary Key	406
17.6	Primary Key dengan Index Prefix	407
17.7	Primary Key dengan Auto Increment	408
17.8	Pengertian Referential Integrity dan Foreign Key	409
17.9	Cara Membuat Referential Integrity	412
17.10	Penggunaan aturan ON UPDATE dan ON DELETE	417
17.11	Membuat Referential Integrity dengan ALTER TABLE	424
17.12	Menghapus Referential Integrity	427
17.13	Membuat nama Referential Integrity	428
17.14	Index di Parent Table	429
18.	Import dan Export Data	432
18.1	Import Data dengan LOAD DATA INFILE	432
18.2	Import Data dari File CSV	453
18.3	Export Data dengan SELECT ... INTO OUTFILE	462
18.4	Export Data dengan Mysqldump	465
18.5	Import Data dengan Mysqldump	468
19.	Pembuatan User dan Hak Akses (Privilege)	474
19.1	Melihat Daftar User MySQL	474
19.2	Cara Login ke MySQL Server	479
19.3	Membuat User MySQL	480
19.4	Membuat User dengan Batasan Host	484
19.5	Membuat User dengan Password	490
19.6	Mengenal Hak Akses (Privilege)	494

CONTENTS

19.7	Membuat Hak Akses (Privilege)	496
19.8	Menghapus Hak Akses (Privilege)	502
19.9	Permissible Privileges	505
19.10	Hak Akses untuk Seluruh Tabel dan Database	506
19.11	Hak akses WITH GRANT OPTION	507
19.12	SQL Mode: NO_AUTO_CREATE_USER	509
20.	MySQL Operator dan Function	513
20.1	Operator Aritmatika MySQL	513
20.2	Cara Pemanggilan Function MySQL	519
21.	MySQL Numeric Function	522
21.1	Function CEIL(), CEILING(), FLOOR(), ROUND() dan TRUNCATE()	522
21.2	Function SIN(), COS(), TAN(), ASIN(), ACOS(), ATAN() dan COT()	526
21.3	Function PI(), RADIANS() dan DEGREES()	527
21.4	Function LOG(), LOG2(), LOG10() dan LN()	527
21.5	Function POW(), POWER() dan EXP()	529
21.6	Function RAND()	530
21.7	Function SQRT()	533
21.8	Function ABS()	533
21.9	Function COUNT(), MIN(), MAX(), AVG(), SUM() dan STD()	534
22.	MySQL String Function	536
22.1	Function FORMAT()	536
22.2	Function LOWER(), LCASE(), UPPER() dan UCASE()	537
22.3	Function CHAR_LENGTH(), CHARACTER_LENGTH(), LENGTH() dan BIT_LENGTH()	538
22.4	Function CONCAT() dan CONCAT_WS()	539
22.5	Function LPAD() dan RPAD()	540
22.6	Function LTRIM(), RTRIM() dan TRIM()	542
22.7	Function LOCATE()	543
22.8	Function SUBSTR() dan SUBSTRING()	544
22.9	Function LEFT() dan RIGHT()	545
22.10	Function INSERT()	546
22.11	Function REVERSE()	546
23.	MySQL Date and Time Function	548
23.1	Function CURDATE() dan CURRENT_DATE()	548
23.2	Function CURTIME() dan CURRENT_TIME()	549
23.3	Function CURRENT_TIMESTAMP(), LOCALTIME(), LOCALTIMESTAMP() dan NOW()	549
23.4	Function DATE() dan TIME()	550
23.5	Function ADDDATE() dan DATE_ADD()	551
23.6	Function SUBDATE() dan DATE_SUB()	554
23.7	Function ADDTIME()	555
23.8	Function DATEDIFF()	555

CONTENTS

23.9	Function DATE_FORMAT()	556
23.10	Function STR_TO_DATE()	558
23.11	Function SEC_TO_TIME()	559
23.12	MySQL Function vs PHP Function	559
24.	MySQL VIEW	561
24.1	Pengertian VIEW	561
24.2	Membuat dan Menghapus VIEW	563
24.3	VIEW untuk JOIN dan Penambahan Data	567
24.4	Update data dari VIEW	570
24.5	Konsistensi Update VIEW dengan WITH CHECK OPTION	572
24.6	Latihan VIEW	574
24.7	Jawaban Latihan VIEW	576
25.	MySQL Stored Procedure dan Stored Function	578
25.1	Pengertian Stored Procedure dan Stored Function	578
25.2	Perlukah menggunakan Stored Procedure dan Stored Function?	579
25.3	MySQL Variable	581
25.4	Format Dasar Stored Procedure dan Stored Function	584
25.5	Perintah DELIMETER	585
25.6	Membuat Stored Procedure	587
25.7	Menghapus Stored Procedure	590
25.8	Stored Procedure Variable	590
25.9	Stored Procedure Parameter	592
25.10	Stored Procedure IF Statement	596
25.11	Stored Procedure IF ELSE Statement	597
25.12	Stored Procedure IF ELSEIF ELSE Statement	598
25.13	Stored Procedure Simple CASE Statement	600
25.14	Stored Procedure Complex CASE Statement	601
25.15	Stored Procedure WHILE Loop	603
25.16	Stored Procedure REPEAT Loop	605
25.17	Stored Function	606
25.18	Latihan Stored Procedure	607
25.19	Jawaban Latihan Stored Procedure	608
26.	MySQL Trigger	614
26.1	Pengertian Trigger	614
26.2	Format Dasar Penulisan Trigger	614
26.3	Membuat Trigger	615
26.4	Implisit Event Trigger	621
26.5	Menghapus Trigger	623
26.6	BEFORE atau AFTER?	624
26.7	Latihan Trigger	626
26.8	Jawaban Latihan Trigger	627
27.	Pengantar phpMyAdmin	629

CONTENTS

27.1	Pengertian phpMyAdmin	629
27.2	Keunggulan dan Kekurangan phpMyAdmin	630
27.3	Mengakses phpMyAdmin dari Localhost	630
27.4	Membuat Database dan Tabel	632
27.5	Mengisi Data Tabel	635
27.6	Menampilkan Isi Tabel	637
27.7	Menjalankan Kode SQL	638
	Penutup: MySQL Uncover	641
	Daftar Pustaka	643

Ucapan Terimakasih

Dalam kesempatan ini saya ingin mengucapkan terimakasih kepada Allah S.W.T karena dengan karuniaNya saya masih diberi kesempatan dan kesehatan untuk bisa menulis buku kelima DuniaIlkom: **MySQL Uncover**.

Selanjutnya kepada keluarga saya yang terus memberi motivasi dan dukungan tiada henti untuk terus mengembangkan DuniaIlkom.

Terakhir kepada rekan-rekan pembaca dan pengunjung setia DuniaIlkom. Terutama bagi yang telah memberikan donasi untuk membeli buku saya sebelumnya: **HTML Uncover**, **CSS Uncover**, **PHP Uncover** dan **JavaScript Uncover**. Karena dari *feedback* dan dukungan rekan-rekan lah saya bisa lanjut menulis eBook **MySQL Uncover** ini. Terimakasih :)

Padang Panjang, 2017

Penulis

Andre Pratama

www.duniaIlkom.com

Tentang Penulis

Andre Pratama



Saat ini memilih karir sebagai praktisi dan penulis di duniailkom.com. Menamatkan kuliah S1 Ilmu Komputer di Universitas Sumatera Utara pada tahun 2010. Sempat terjun ke dunia kerja melalui ODP (*Officer Development Program*) di Bank BUMN terbesar di Indonesia. Di sela-sela kesibukan, mendirikan situs duniailkom.com pada tahun 2012.

Karena kecintaan akan menulis dan programming, akhirnya memutuskan untuk keluar dari dunia perbankan dan fokus mengelola duniailkom di akhir tahun 2014. Berusaha untuk menjadikan duniailkom sebagai salah satu media belajar programming dan ilmu komputer terbaik di Indonesia.

Berdomisili di kota Padang Panjang, Sumatera Barat, Andre bisa dihubungi melalui email duniailkom di duniailkom@gmail.com, facebook: facebook.com/belajar.duniailkom¹, dan twitter: twitter.com/duniailkom².

¹<https://www.facebook.com/belajar.duniailkom>

²<https://twitter.com/duniailkom>

Kata Pengantar

MySQL melengkapi 5 materi dasar web programming, yakni: HTML, CSS, PHP, MySQL dan JavaScript.

MySQL adalah salah aplikasi database dimana data website nantinya akan disimpan. Meskipun terdapat berbagai aplikasi database sejenis, di bidang web programming MySQL tetap mendominasi. MySQL bisa digunakan dengan gratis, namun aplikasi ini bukanlah database yang “seadanya”. Website besar seperti Wikipedia, Facebook, Twitter, dan Tesla juga memakai teknologi MySQL dalam [kegiatan operasional](#)³.

Dalam perkembangannya, hadir MariaDB sebagai “cloningan” dari MySQL. MariaDB dibuat oleh tim yang dulunya juga membuat MySQL. Ini terjadi karena berbagai kalangan khawatir akan nasib MySQL sejak diambil alih oleh Oracle (yang sebenarnya saingan dari database MySQL). Lebih lanjut mengenai kisah MySQL dan MariaDB akan dibahas dalam 1 bab khusus.

MariaDB hadir sebagai “*drop-in-replacement*” dari MySQL. Artinya, seluruh kode program yang dulunya dibuat untuk MySQL, bisa langsung berjalan di MariaDB tanpa perubahan apapun.

Faktanya, jika kita mendownload aplikasi [XAMPP](#)⁴, sekarang sudah di *bundle* dengan database MariaDB, bukan lagi MySQL. Anda mungkin tidak sadar akan hal ini, karena di sisi programming (PHP) dan query yang dipakai, tidak ada perbedaan antara MySQL dengan MariaDB.

Buku **MySQL Uncover** ini sebenarnya akan lebih pas jika saya tulis dengan judul **MariaDB Uncover**, karena sepanjang buku nanti saya akan memakai MariaDB bawaan XAMPP, bukan MySQL.

Akan tetapi, harus diakui nama MariaDB belum terlalu populer, masih kalah jauh daripada MySQL. Oleh karena itu saya tetap menamakan buku ini sebagai **MySQL Uncover**. Terlebih seluruh kode program dan query yang ada bisa berjalan baik di MySQL maupun MariaDB.

Dalam buku ini saya akan mengajak anda untuk membahas berbagai perintah atau query MySQL/MariaDB, mulai dari cara menginstall, menjalankan MySQL server dan MySQL Client, mengenal berbagai query seperti CREATE, ALTER, SELECT, UPDATE, hingga materi lanjutan seperti *referential integrity*, *view*, *stored procedure*, *trigger* serta pengantar aplikasi **phpMyAdmin**.

Semoga buku **MySQL Uncover** ini bisa menjadi buku pengantar terbaik dalam menguasai MySQL / MariaDB. Sampai jumpa di bab terakhir :)

³<https://www.mysql.com/customers/>

⁴<https://www.apachefriends.org>

Terimakasih untuk tidak memperbanyak / mengedarkan / men-copy eBook ini

Menulis sebuah buku hingga ratusan halaman butuh waktu yang tidak sebentar. Belum lagi saya harus berjuang mempelajari referensi yang kebanyakan dalam bahasa inggris. Ini saya lakukan agar pembaca bisa mendapatkan materi yang detail, update, dan berkualitas.

Saya menyadari kekurangan sebuah ebook adalah mudah dicopy-paste dan disebarluaskan. Tapi dengan eBook, harga buku bisa ditekan. Selain tidak perlu mencetak, eBook DuniaIlkom ini bisa di dapat dengan murah, termasuk bagi teman-teman di daerah yang ongkos kirimnya lumayan mahal (jika berbentuk buku fisik).

Atas dasar itulah saya mohon kerjasamanya dari rekan-rekan semua **untuk tidak memperbanyak, menggandakan, atau meng-upload ulang buku ini di forum, situs dan media lain**.

Saya juga berharap rekan-rekan tidak memposting materi apapun yang ada di dalam buku ini. Jika ingin sebagai bahan artikel untuk postingan blog/situs, silahkan ambil materi yang ada di website duniaIlkom (jangan yang dari buku).

Apabila rekan-rekan memperoleh buku ini **bukan** dari DuniaIlkom, saya mohon bantuan donasinya untuk membeli versi asli. Donasi pembelian buku ini adalah sumber mata pencarian saya untuk menafkahsi keluarga. Lisensi atau hak guna buku ini hanya untuk 1 orang, yakni yang telah membeli langsung ke **duniaIlkom@gmail.com**.

Dengan kualitas yang ditawarkan, harga buku ini cukup terjangkau. Buku ini saya buat dengan waktu yang tidak sebentar, hingga berbulan-bulan, dan kadang sampai tengah malam. Bantuan donasi dari rekan-rekan yang membeli buku secara resmi sangat saya hargai, selain mendapat ilmu yang berkah, ini juga bisa menjadi penyemangat saya untuk terus berkarya dan menghadirkan ebook-ebook programming berkualitas lainnya.

Untuk yang membeli dari DuniaIlkom, saya ucapan banyak terimakasih :)

Anda diperbolehkan untuk:

- Mencetak eBook ini untuk keperluan pribadi dan dibaca sendiri.
- Mencopy eBook ini ke laptop/smartphone/laptop milik sendiri.
- Membuat ringkasan buku untuk digunakan sebagai bahan ajar (bukan keseluruhan isi buku).

Anda tidak dibolehkan untuk:

- Mencetak eBook ini untuk dibaca oleh orang lain, walaupun gratis.
- Mencopy eBook ini untuk dijual ulang, maupun dibagikan kepada orang lain dengan gratis.
- Membeli buku ini untuk dibaca bersama-sama (lisensi buku ini hanya untuk 1 orang).
- Mengambil sebagian atau seluruh isi buku untuk di publish ke blog, situs, artikel, dan media publik lain.
- Membagikan eBook ini kepada murid/siswa/mahasiswa (jika digunakan untuk bahan pengajaran).

Asumsi / Pengetahuan Dasar

Untuk bisa mempelajari MySQL atau MariaDB, tidak diperlukan keahlian programming khusus. Meskipun saya yakin jika anda tertarik mempelajari database, sedikit banyak sudah familiar dengan bahasa pemrograman, terutama PHP.

Buku **MySQL Uncover** ini akan fokus mendalami fitur dan perintah query di dalam MySQL. Disini tidak akan dibahas mengenai teori seputar perancangan database maupun diagram entity seperti ERD (Entity Relationship Diagram). Pengetahuan seputar teori perancangan database seperti ini akan sangat membantu dalam mendalami materi yang ada, tapi tidak harus.

Sepanjang buku saya lebih banyak menyebut “MySQL”. Dalam mayoritas situasi, ini juga berarti “MySQL dan MariaDB”.

Contoh Kode Program

Seluruh contoh kode program yang ada di buku **MySQL Uncover** bisa di download dari folder sharing *Google Drive* yang saya kirim pada saat pembelian: **belajar_mysql.zip**.

Sebagian besar materi MySQL di ketik menggunakan **cmd Windows**, oleh karena itu saya merangkum setiap perintah query ke dalam 1 file text: **query.txt**.

Cara lain adalah, anda bisa langsung copy-paste contoh kode program yang ada di eBook ini ke dalam cmd Windows. Silahkan block kode program yang ingin di copy, kemudian tekan kombinasi tombol **CRTL + C**.

Untuk men-paste ke **cmd Windows**, bisa dengan klik kanan atau tekan kombinasi tombol **CRTL + V**.

Alternatif yang lebih saya sarankan adalah dengan mengetik ulang seluruh kode program. Tujuannya agar anda lebih cepat paham sekaligus bisa menghafal fungsi dari kode-kode tersebut.

- i** Jika anda mencoba menulis ulang kode program yang ada di buku, dan ternyata tidak jalan, besar kemungkinan terdapat penulisan yang salah.

Di dalam programming, 1 karakter saja yang kurang (apakah itu berupa titik, spasi, tanda koma, atau penulisan huruf besar/kecil), kode program tidak akan jalan sempurna. Jika ini yang terjadi, silahkan anda copy perintah tersebut dari eBook atau dari file-file dalam folder **belajar_mysql.zip**.

1. Berkenalan dengan MySQL

MySQL merupakan salah satu aplikasi database yang paling populer saat ini, terutama di web programming. MySQL mewakili huruf “M” dari LAMP stack (Linux, Apache, MySQL, dan PHP).

LAMP stack sendiri merupakan sebuah sistem pengembangan web yang paling banyak dipakai. Jika anda berencana membuat website yang memerlukan database, MySQL juga sudah tersedia di webhosting (**cPanel**) dan siap digunakan.

Dalam bab pertama buku **MySQL Uncover** ini saya akan membahas materi dasar seperti apa itu Database, mengenal Database Model, pengertian RDBMS, hingga alasan kenapa kita menggunakan MySQL.

1.1 Pengertian Database

Dikutip dari [wikipedia¹](#): “A database is an organized collection of data”. Sebuah database adalah kumpulan data yang terorganisasi. Dalam bahasa Indonesia, database ini dikenal juga dengan sebutan **Basis Data**.

Secara tradisional, data tidak harus berada di komputer. Masih banyak kantor dan perusahaan yang menyimpan catatan absen, penjualan serta laporan bulanan menggunakan kertas. Jika data-data ini dikelompokkan dan disusun menggunakan sebuah cara tertentu, itu juga sudah bisa dikatakan sebagai database.

Terdapat berbagai kelemahan jika menyimpan data di media kertas. Mulai dari mudah tercecer, membutuhkan tempat yang besar (jika data tersebut banyak), hingga susahnya mencari informasi yang dibutuhkan dengan cepat.

Perkembangan teknologi, khususnya komputer menyediakan solusi untuk menyimpan data-data ini. Daripada menulis di kertas, data tersebut bisa disimpan ke dalam komputer. Bagaimana caranya?

Salah satu solusi adalah menggunakan aplikasi *spreadsheet* seperti **Microsoft Excel**. Data hasil penjualan bisa di-input ke dalam tabel excel dengan mudah. Untuk data lain, kita tinggal membuat file excel baru. Solusi ini sangat praktis dan mudah untuk dipelajari.

Solusi kedua adalah menyimpan data di aplikasi khusus database seperti **MySQL**.

Menyimpan data di Microsoft Excel dan MySQL memiliki tujuan yang berbeda. Untuk data yang sederhana, tabel di excel sudah mencukupi. Namun jika data tersebut cukup kompleks, saling berhubungan, serta harus diakses banyak orang sekaligus, menyimpan data di MySQL akan lebih pas.

Alasan lain yang tak kalah penting adalah, aplikasi seperti MySQL bisa digabung dengan bahasa pemrograman. Bahasa pemrograman ini nantinya berfungsi untuk membuat interface atau tampilan aplikasi, sedangkan datanya disimpan ke dalam MySQL.

¹<https://en.wikipedia.org/wiki/Database>

Di dalam web programming, bahasa pemrograman yang dimaksud adalah **HTML, CSS, PHP** dan **JavaScript**. Ke empat bahasa ini digunakan untuk membuat tampilan form. Setelah itu PHP akan memproses data yang disimpan ke dalam database MySQL. Inilah alur lengkap pemrosesan form di sebuah website.

1.2 Pengertian Database Model

MySQL merupakan sebuah aplikasi **RDBMS**, singkatan dari *Relational Database Management System*. Pengertian sederhana dari RDBMS adalah: aplikasi database yang menggunakan prinsip relasional. Mari kita bahas pengertian dari “**Relational Database**” terlebih dahulu.

Relational Database adalah suatu jenis database model. Sekali lagi, dengan mengutip [Wikipedia²](#):

“A database model is the theoretical foundation of a database and fundamentally determines in which manner data can be stored, organized, and manipulated in a database system”.

Terjemahan bebasnya:

Database model adalah teori dasar seputar bagaimana data itu akan disimpan, disusun, dan dimanipulasi dalam sebuah sistem database.

Dalam teori database, banyak cara yang dikembangkan untuk menyimpan data di komputer, mulai dari **flat model**, **hierarchical model**, **network model**, **relational model**, hingga **object oriented model**.

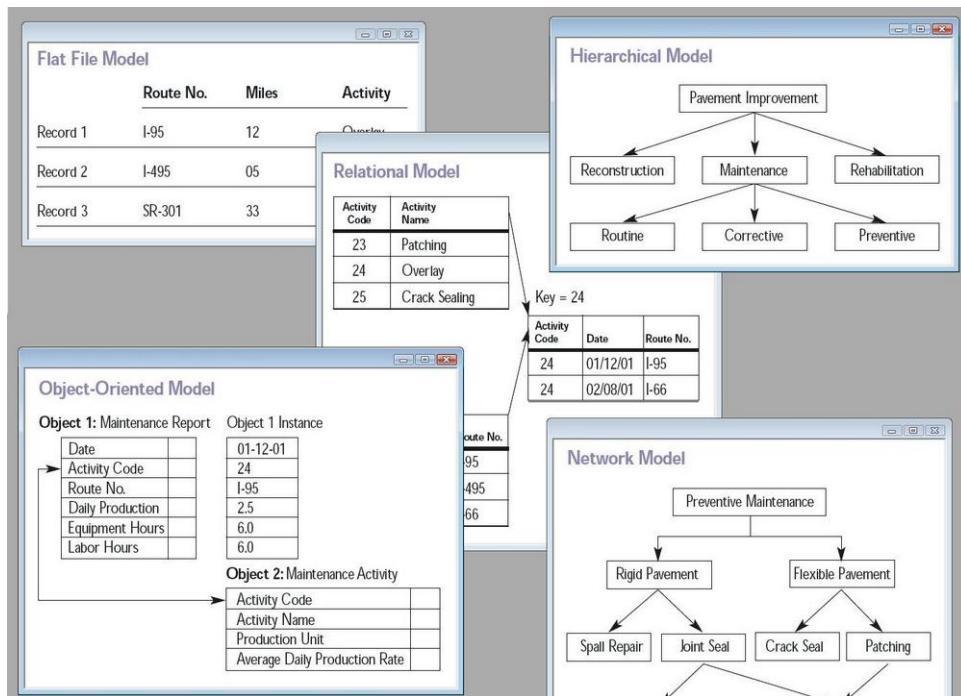
Flat model adalah istilah lain dari tabel sederhana seperti di Microsoft Excel, yakni menyimpan data tanpa aturan dan cara penulisan tertentu.

Dalam **hierarchical model**, data disusun seperti pohon terbalik, dimana data terorganisasi dari atas ke bawah. Model database ini digunakan pada sistem database awal, seperti **Information Management System (IMS)** oleh IBM pada tahun 1966. Sedangkan **network model** merupakan pengembangan dari hierarchical model.

Relation model menyusun data ke dalam bentuk tabel yang saling terhubung. Tabel-tabel inilah yang nantinya menjadi sebuah database.

Object oriented model menggunakan pendekatan object dalam menyimpan data. Setiap data dianggap sebagai object yang memiliki berbagai property. Kurang lebih mirip seperti konsep OOP dalam bahasa pemrograman berbasis object.

²http://en.wikipedia.org/wiki/Database_model



Gambar: Berbagai jenis database model, sumber: wikipedia.org



Pembahasan lebih lanjut tentang **Database Model**, dapat dibaca di wikipedia: [Database Model](#)³.

1.3 Pengertian Relational Model

Dari sekian banyak pemodelan data, **Relational Model** masih yang populer dan paling banyak dipakai.

Konsep **Relational Database Model** diajukan pertama kali oleh peneliti IBM dan ahli matematika, **Dr. Edgar F. Codd**⁴ pada tahun 1969. Pada awalnya Dr. Codd mencari cara baru untuk menangani data dalam jumlah besar. Untuk kasus seperti ini, hierarchical dan network model tidak lagi efektif.

Sesuai dengan keahliannya, Dr.Codd memakai pendekatan matematis dan berusaha mencari cara untuk menyelesaikan permasalahan yang sering timbul dalam database, seperti redundansi data (data sama yang berulang), hubungan antar data, dan ketergantungan kepada urutan di media penyimpanan.

Pada Juni 1970, Dr.Codd mengajukan ide tentang relational model dalam sebuah paper berjudul "[A Relational Model of Data for Large Shared Databanks](#)"⁵. Konsep relational database model berasal dari 2 cabang ilmu matematika : **set theory** dan **first-order predicate logic**.

³https://en.wikipedia.org/wiki/Database_model

⁴http://en.wikipedia.org/wiki/Edgar_F._Codd

⁵<http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf>

Sebuah relational database model, menyimpan data dalam ‘relasi’, atau lebih kita kenal sebagai **tabel**. Setiap tabel terdiri dari tuple (*record*) dan attribut (*field*). Kedua istilah ini tidak lain sebutan untuk **baris** dan **kolom** di dalam sebuah tabel.



The diagram illustrates the concept of a relational database model. It shows two tables side-by-side, connected by a red curved arrow pointing from the top table to the bottom table.

NIM	Nama	Tempat Lahir	Tanggal Lahir	Fakultas	Jurusan
15002032	Rina Kumala Sari	Jakarta	28/06/1997	Ekonomi	Akuntansi
13012012	James Situmorang	Medan	02/04/1995	Kedokteran	Kedokteran Gigi
14005011	Riana Putria	Padang	23/11/1996	FMIPA	Kimia
15021044	Rudi Permana	Bandung	22/08/1994	FASILKOM	Ilmu Komputer
15003036	Sari Citra Lestari	Jakarta	31/12/1997	Ekonomi	Manajemen

No	NIM	IP Semester 1	IP Semester 2	IP Semester 3	...	IPK
1	15002032	3.7	3.1			3.4
2	13012012	2.5	3.0	2.9		2.7
3	14005011	3.5	2.9	3.3		3.1
4	15021044	2.2	3.2	3.1		2.9
5	15003036	3.8	3.2			3.5

Gambar: Bentuk tabel dalam relational database model

Relational database model inilah yang paling populer dan banyak diimplementasi dalam berbagai aplikasi database saat ini, termasuk MySQL. Aplikasi database yang menggunakan konsep relational model inilah yang disebut sebagai **Relational Database Management Systems (RDBMS)**.

1.4 RDBMS : Relational Database Management Systems

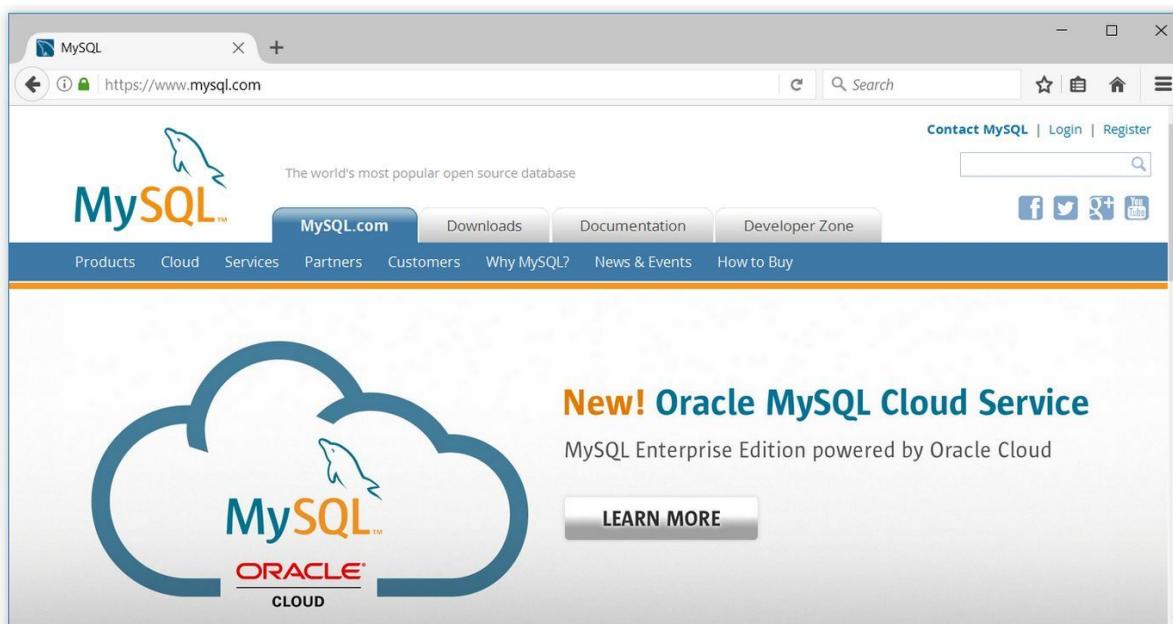
Relational Database Management Systems (RDBMS) adalah software/aplikasi yang menggunakan relational database model sebagai dasarnya.

Sejak 1970an, RDBMS sudah banyak digunakan. Dua RDBMS pertama adalah **System R**, yang dikembangkan oleh IBM, dan **INGRES** (Interactive Graphics Retrieval System) yang dikembangkan oleh University of California di Berkeley. Keduanya pada awal 1970an.

Setelah keunggulan Relational Database banyak dikenal, berbagai perusahaan mulai berlalih dari hierarchical dan network database model ke relational database model. Pada tahun 1980an, **Oracle RDBMS** lahir, dan diikuti oleh pesaingnya saat itu, **IBM DB2 RDBMS**.

Jika pada tahun 1980an RDBMS hanya dapat digunakan dalam sistem mainframe perusahaan besar (sejenis “super komputer”), saat ini dengan semakin majunya perkembangan teknologi di sisi hardware, PC-based RDBMS sudah banyak tersedia, seperti MySQL yang bisa diinstall di komputer / laptop biasa.

1.5 MySQL sebagai RDBMS



Gambar: Tampilan website www.mysql.com

Bagi web developer, kalau bicara database kemungkinan besar akan membicarakan MySQL. Kenapa harus MySQL? Bagaimana dengan aplikasi database lain? Jawaban singkatnya, MySQL merupakan RDBMS yang gratis dan mudah digunakan (user friendly).

MySQL juga bukan satu-satunya RDBMS, list lengkapnya ada di [wikipedia](#)⁶. Diantaranya adalah: **Oracle, Sybase, Microsoft Access, Microsoft SQL Server, PostgreSQL** dan juga **MariaDB**.

MySQL bersifat gratis dan open source. Artinya setiap orang boleh menggunakan dan mengembangkan aplikasi ini. MySQL di support oleh ribuan programmer dari seluruh dunia, dan merupakan sebuah aplikasi RDBMS yang lengkap, cepat, dan reliabel.

Berikut beberapa keunggulan MySQL dibandingkan dengan RDBMS lain:

Speed

Sebuah studi dari eWeek di February 2002 yang membandingkan performa kecepatan MySQL dengan RDBMS lain seperti **Microsoft SQL Server 2000, IBM DB2, Oracle 9i** dan **Sybase**. Berikut kesimpulannya:

“MySQL has the best overall performance and that MySQL scalability matches Oracle ... MySQL had the highest throughput, even exceeding the numbers generated by Oracle.”

Terjemahannya:

⁶http://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems

“MySQL memiliki performa terbaik secara keseluruhan. Skalabilitas MySQL menya-mai Oracle ... MySQL mempunyai nilai output yang paling tinggi, bahkan melebihi nilai yang dihasilkan oleh Oracle.”

MySQL memiliki kecepatan yang lebih dibandingkan pesaing yang berbayar. Bagi anda ingin membaca paper tersebut, tersedia di sini: [MySQL Performance Benchmarks⁷](#).

Reliability

Biasanya sesuatu yang gratis susah diandalkan, bahkan banyak bug dan sering hang. Tidak demikian dengan MySQL. Karena sifatnya yang open source, setiap orang dapat berkontribusi memeriksa bug dan melakukan test case untuk berbagai skenario. MySQL bisa digunakan untuk sistem 24 jam online, multi-user dan data ratusan GB.

Skalability

MySQL dapat memproses data yang sangat besar dan kompleks tanpa ada penurunan performa yang berarti, juga mendukung sistem multi-prosesor. MySQL dipakai oleh perusahaan-perusahaan besar di dunia, seperti Epson, New York Times, Wikipedia, Google, Facebook, bahkan NASA.

User Friendly

Instalasi dan mempelajari MySQL cukup mudah. Download aplikasi MySQL dan install, kita bisa menggunakan MySQL dalam waktu kurang dari 5 menit. Namun MySQL juga menyediakan berbagai konfigurasi lanjutan jika diperlukan.

Portability and Standard Compliance

Database MySQL dapat dengan mudah berpindah dari satu sistem ke sistem lainnya. Misalkan dari sistem Windows ke Linux. Aplikasi MySQL juga dapat berjalan di hampir semua sistem operasi modern, mulai dari Linux, Unix, BSD, Mac OS, hingga Windows.

Multiuser Support

Dengan menerapkan arsitektur *client-server*. Ribuan pengguna dapat mengakses database MySQL dalam waktu yang bersamaan.

Internationalization

Atau dalam bahasa sederhananya, mendukung beragam bahasa. Dengan dukungan penuh terhadap **unicode**, maka aksara non-latin seperti jepang, cina, dan korea bisa digunakan di dalam MySQL.

⁷ <http://www.jonahharris.com/osdb/mysql/mysql-performance-whitepaper.pdf>

Wide Application Support

Biasanya database RDBMS tidak dipakai sendirian, tapi “ditemani” dengan aplikasi atau bahasa pemrograman. Bahasa pemrograman ini digunakan untuk membuat interface (tampilan visual dari database).

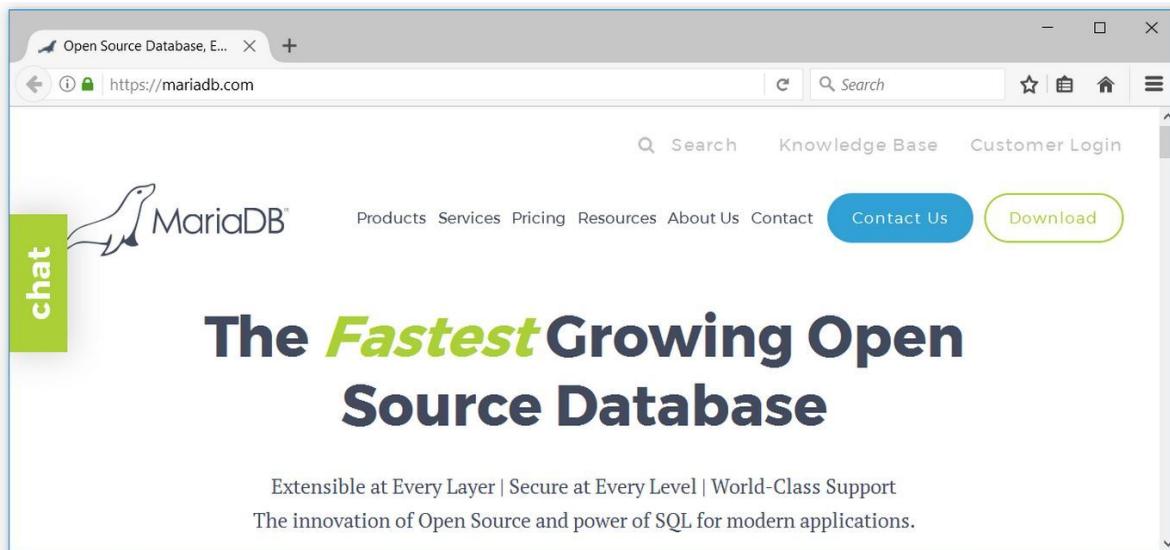
Terdapat beragam bahasa pemrograman yang bisa digunakan untuk berkomunikasi dengan MySQL, seperti C, C++, C#, Java, Delphi, Visual Basic, Perl Python dan PHP. Kesemua itu di dukung oleh API (Application Programming Interface) oleh MySQL.

Dalam buku ini kita akan lebih banyak membahas penggunaan MySQL dengan bahasa pemrograman PHP.

Open Source Code

Kitapun bisa (jika mampu dan mengerti), mengembangkan MySQL dengan mempelajari kode programnya. MySQL di kembangkan dengan bahasa C and C++. Bahkan sudah terdapat berbagai variasi rasa RDBMS baru yang dikembangkan dari code MySQL, diantaranya: Drizzle, MariaDB, Percona Server dan OurDelta.

1.6 MySQL atau MariaDB?



Gambar: Tampilan website [www.mariadb.com](https://mariadb.com)

Pada tahun 2010, MySQL dibeli oleh perusahaan Oracle. Hal ini membuat khawatir banyak kalangan. Karena sebagaimana yang kita tau, Oracle merupakan perusahaan database terbesar di dunia, yang salah satu saingannya adalah MySQL.

Banyak yang khawatir bahwa suatu saat pengembangan MySQL akan dihentikan oleh Oracle. Untuk mengantisipasi hal ini, dikembangkanlah MariaDB sebagai pengganti MySQL.

Pada dasarnya, MariaDB adalah *cloningan* MySQL. Boleh dibilang bahwa MariaDB berisi MySQL yang diberi “merk” MariaDB serta penambahan fitur dan perbaikan performa. Seluruh materi yang dibahas dalam buku ini bisa berjalan baik di MariaDB maupun MySQL.

Jika anda sebelumnya sudah mempelajari PHP, terdapat fungsi seperti `mysqli_connect()` yang digunakan untuk berkomunikasi dengan MySQL. Fungsi `mysqli_connect()` ini juga bisa digunakan untuk MariaDB, tanpa modifikasi apapun.

Yang juga cukup unik, MariaDB ini dikembangkan oleh pihak yang dulunya juga membuat MySQL. Lebih jauh tentang kisah MySQL dan MariaDB ini akan saya bahas dalam bab berikutnya: **Sejarah Perkembangan MySQL dan MariaDB**.

2. Sejarah Perkembangan MySQL dan MariaDB

“*Tidak kenal maka tidak sayang*”, untuk itulah dalam bab ini saya ingin membahas tentang sejarah perkembangan salah satu aplikasi database populer di dunia: MySQL. Kita akan lihat bagaimana awal kemunculan, hingga lahirnya MariaDB sebagai alternatif pengganti dari MySQL.

2.1 Kisah tentang UNIREG dan TcX

Cerita tentang MySQL bermula pada tahun 1981. Michael Widenius (Monty), seorang programmer asal Finlandia, mengembangkan sebuah sistem database sederhana yang dinamakan **UNIREG**¹. UNIREG ini memakai koneksi low-level *ISAM database engine* dengan *indexing* (baca: sistem database sederhana yang mendukung *index*).



Gambar: Michael Widenius (Monty), sumber: wikimedia.org

Pada tahun 1994, perusahaan kecil di swedia bernama TcX ingin mengembangkan aplikasi berbasis web. TcX sendiri juga didirikan oleh Monty. Bisa dibilang TcX merupakan perusahaan “startup” yang sedang dirintis oleh Monty.

Dalam mengembangkan TcX, Monty tetap menggunakan sistem UNIREG. Merasa performa UNIREG kurang baik untuk aplikasi web, ia mencari alternatif sistem database lain. Salah satunya mSQL (miniSQL), sebuah aplikasi database yang tidak terlalu mahal dan hampir open source, maksudnya jika anda membeli aplikasi ini, juga mendapatkan source code.

¹https://docstore.mik.ua/oreilly/weblinux2/mysql/ch01_02.htm

Aplikasi mSQL ternyata juga memiliki kekurangan yang membuat performanya tidak jauh berbeda dari UNIREG, alasannya karena mSQL tidak mendukung *indexing*.



Index merupakan fitur untuk mempercepat pencarian data yang saat ini umum tersedia di aplikasi database modern. Baik MySQL maupun MariaDB sudah mendukung indexing.

Monty mencoba menghubungi **David Hughes**, programmer yang mengembangkan mSQL. Ia menanyakan apakah David Hughes tertarik untuk membuat *konektor* antara mSQL dengan UNIREG, supaya fitur indexing bisa dipakai di mSQL.

Pada saat itu Hughes menolak, dengan alasan ia telah mengembangkan teknologi indexing sendiri yang nantinya akan dipakai dalam mSQL versi 2.

Karena penolakan ini, Monty memutuskan untuk merancang dan mengembangkan sendiri sistem database baru. Konsep “*don’t reinvent the wheel*”, dipakai dalam mengembangkan aplikasi database ini. Daripada membuat semuanya dari nol, Monty menggabungkan source code UNIREG dan mSQL API (yang memang boleh digunakan).

Hasilnya, perintah dasar dari database baru yang dikembangkan TcX (Monty), menjadi mirip dengan mSQL. Hal tersebut membawa banyak keuntungan. Berbagai aplikasi library pihak ketiga yang pada awalnya ditujukan untuk mSQL, juga bisa dipakai ke sistem ini, tanpa modifikasi apapun.

Selang beberapa waktu kemudian, ikut bergabung **David Axmark** dan **Allan Larsson**. Bertiga dengan Monty, pada tahun 1995 TcX merilis database mereka: **MySQL**. Versi pertama dari MySQL sudah memiliki nama **MySQL 3.11**. Penomoran dibawah itu, seperti MySQL 1.0, MySQL 2.0 hanya digunakan secara internal dalam perusahaan.



Gambar: David Axmark, sumber: wikipedia.org

2.2 MySQL AB dan Open Source

Sesaat setelah perilisan MySQL, TcX berubah nama menjadi **MySQL AB**, dengan **Michael Widenius, David Axmark** dan **Allan Larsson** sebagai pendiri. Titel “AB” dibelakang MySQL adalah singkatan dari “*Aktiebolag*”, istilah PT (Perseroan Terbatas) bagi perusahaan Swedia.

MySQL AB menjadi perusahaan di belakang MySQL yang bertanggung jawab dalam mengembangkan, memasarkan, dan menyediakan dukungan untuk MySQL. MySQL sendiri dirilis dengan “dual licensing”, atau dua lisensi yakni versi gratis dan versi berbayar.

Lisensi pertama di rilis dibawah **GNU GPL** (General Public License – atau dikenal juga dengan *Gak Pakai Lisensi*). Lisensi GPL ini membebaskan siapa saja menggunakan MySQL tanpa perlu membayar royalti kepada MySQL AB. Tapi dengan beberapa syarat tertentu. Misalnya, jika anda menggunakan MySQL dalam aplikasi yang anda buat, aplikasi tersebut juga harus berada di bawah lisensi GPL.

Lisensi kedua ditujukan bagi perusahaan komersil maupun pengembang software yang berniat menjual aplikasi dan menggunakan MySQL sebagai databasenya. Untuk keperluan ini, anda diharuskan membeli lisensi komersial dari MySQL AB. Lebih lanjut tentang permasalahan seputar lisensi MySQL, dapat mengunjungi situs MySQL atau ke artikel ini: [When are you required to have a commercial MySQL license?](#)²

MySQL AB juga memegang hak *copyright* dari source code MySQL dan pemilik hak merk dagang “MySQL”. Dengan kata lain, walaupun kita memiliki source code MySQL, namun sistem database maupun aplikasi yang kita buat tidak boleh menggunakan merk “MySQL” tanpa membayar royalti kepada pihak MySQL AB. Hal ini pula yang menjelaskan mengapa salah satu aplikasi administrasi MySQL berbasis web **PhpMyAdmin**, tidak menggunakan kata “MySQL” pada nama programnya.

MySQL AB juga merupakan sebuah perusahaan yang unik. Sangat mirip seperti startup modern saat ini. Pada saat pendirian, karyawan perusahaan hanya ada 3 orang, yakni **Michael Widenius, David Axmark** dan **Allan Larsson**. Saat pendirian tersebut, perusahaan ini juga tidak memiliki kantor (virtual company).

Pada tahun 2000, jumlah karyawan bertambah menjadi 15 orang. Ke-15 orang ini semuanya memiliki background sebagai programmer dan mayoritas bekerja secara virtual dari rumah masing-masing.

²<https://www.xaprb.com/blog/2009/02/17/when-are-you-required-to-have-a-commercial-mysql-license/>

2.3 MySQL AB sebagai “Start Up”



Gambar: Logo MySQL

Perlahan tapi pasti, popularitas MySQL sebagai aplikasi database makin meningkat. Mayoritas website perlu database untuk menyimpan data, dan MySQL merupakan pilihan yang paling pas (dan juga gratis).

Melihat potensi besar di MySQL, banyak perusahaan yang menawar untuk membeli MySQL AB. Pada tahun 1999 terdapat penawaran hingga US \$50 juta, tapi ditolak oleh Monty dan MySQL AB. Alasannya, mereka masih ingin mengembangkan MySQL lebih jauh lagi.

Pada tahun 2001, MySQL AB mencatat bahwa aplikasi MySQL telah digunakan oleh 1 juta pengguna aktif di seluruh dunia.

Namun terdapat tantangan yang harus disikapi. Semakin populernya MySQL, menuntut perusahaan untuk menambah karyawan, tidak hanya programmer tetapi juga karyawan non IT untuk menangani support, dokumentasi, dan penjualan.

Fitur-fitur yang ada juga harus ditambah, diperbaiki dan disempurnakan, karena aplikasi database baru (dan juga gratis) seperti PostgreSQL berpotensi menyaingi MySQL.

Untuk situasi seperti ini, terdapat [3 pilihan bagi MySQL AB³](#):

1. Melanjutkan bisnis seperti biasa dan tumbuh perlahan.
2. Menjual MySQL ke perusahaan yang lebih besar (yang sebelumnya sudah pernah ditolak).
3. Mencari investor.

MySQL AB memilih pilihan ke-3, yakni mencari investor. Caranya sama seperti perusahaan startup sekarang, yakni dengan membuka sesi pendanaan dari perusahaan yang berminat. Pada tahun 2001, MySQL AB berhasil mendapat dana 4 juta euro dari berbagai investor (sekitar Rp 196 miliar).

Dana hasil investasi ini digunakan untuk berbagai keperluan, salah satunya mempekerjakan CEO profesional: **Marten Mickos**, yang meskipun baru berumur 38 tahun, sudah berpengalaman memimpin berbagai perusahaan di Swedia.

Marten Mickos sendiri memiliki latar belakang sales dan marketing, bukan programmer. Sehingga diharapkan bisa memimpin MySQL AB menjadi perusahaan skala global, terutama di sisi bisnis.

³<http://myy.haaga-helia.fi/~dbms/dbtechnet/download/MySQL-MariaDB-story-haaga-helia.pdf>

Juga sama seperti kebanyakan perusahaan startup saat ini, investor yang menanamkan modalnya ke MySQL AB tentu ingin mendapatkan keuntungan. Keuntungan ini baru bisa cair ketika perusahaan dijual, atau saat IPO. Rencananya, MySQL AB akan IPO di tahun 2008.



Gambar: Marten Mickos, sumber: productivemag.com

Mengenal IPO Perusahaan

Sekedar informasi tambahan, saya ingin membahas sedikit tentang IPO, yang merupakan singkatan dari **Initial Public Offering**. IPO adalah mekanisme untuk “menjual” perusahaan di bursa saham. Pengetahuan ini perlu untuk bisa mengikuti kisah MySQL berikutnya.

Jika anda berencana mendirikan sebuah startup, IPO adalah tahap ketika jerih payah kita bisa terbayar. Yakni saat pihak lain ikut bergabung menjadi pemilik dengan cara membeli saham perusahaan yang sebelumnya kita miliki. Berapa harganya? Tergantung seberapa besar perusahaan tersebut. Bisa sekian miliar, atau bahkan triliunan.

Penah dengar berita kalau Traveloka, Tokopedia atau Go-Jek mendapat dana investasi triliunan rupiah? Beritanya bisa kesini: [Ini Daftar Investor Rp 7,2 Triliun Go-Jek^a](#) dan [Tokopedia Umumkan Peroleh Pendanaan Rp 14,7 Triliun dari Alibaba^b](#).

Kenapa ada yang mau memberikan dana sedemikian besar?

Karena si investor berharap, uang tersebut akan kembali berlipat-lipat saat Traveloka, Tokopedia atau Go-Jek terus berkembang dan suatu saat melakukan IPO. Atau hingga ada pihak lain yang membeli saham dengan harga lebih tinggi lagi.

Ketika kita mendirikan perusahaan yang berbentuk PT, ada modal awal yang disetor. Modal ini nantinya digunakan untuk membeli aset perusahaan, seperti gedung, kendaraan, menggaji karyawan, dsb. Bukti kepemilikan dari modal yang disetor inilah yang dikenal sebagai saham.

Saham bisa dimiliki oleh satu, dua, atau banyak orang tergantung siapa yang mau memberikan dananya ke perusahaan. Dalam kasus MySQL AB kita anggap saat pendiriannya saham perusahaan dimiliki oleh 3 orang: Michael Widenius, David Axmark dan Allan Larsson. Masing-masing orang memiliki saham sebesar 33 - 34 %.

Seiring tumbuhnya perusahaan, nilai saham ini juga akan terus meningkat. Tentunya sepanjang perjalanannya, perusahaan membeli gedung baru, mesin baru, yang dihitung sebagai aset.

Uang pembelian ini berasal dari keuntungan perusahaan.

Jika perlu modal atau uang tambahan, saham ini bisa dijual kepada pihak lain yang berminat. Harga saham dipatok tergantung seberapa besar perusahaan sudah berkembang.

Untuk MySQL AB, ada investor yang mau menyetorkan dana sebesar 4 juta euro. Tergantung kesepakatan, bisa jadi si investor akan mendapatkan saham sekitar 10%. Angka 10% ini dikurangi dari jatah pemilik saham yang sudah ada saat itu.

Kenapa apa ada yang mau membeli 10% saham MySQL seharga 4 juta euro? Alasannya, sang investor berharap perusahaan MySQL AB bisa terus tumbuh dan 10% ini suatu saat bisa menjadi 8 juta atau bahkan 100 juta euro.

Siapa yang mau membeli 10% saham MySQL seharga 100 juta euro? Ini bisa terjadi jika keseluruhan perusahaan MySQL dijual seharga 1 miliar euro.

Atau MySQL AB bisa melakukan IPO dengan menjual sebagian sahamnya ke publik. Jika ini dilakukan, sang investor tadi bisa menjual kepemilikan 10% saham ke siapa saja yang berminat di bursa saham. Di Indonesia, terdapat **Bursa Efek Indonesia (BEI)** sebagai tempat perusahaan melakukan IPO.

Beberapa waktu ke depan, kita bisa melihat perusahaan startup seperti Traveloka, Tokopedia atau Go-Jek melakukan IPO. Mungkin mereka sedang menunggu waktu yang pas untuk melakukan hal tersebut.

Alternatifnya, Traveloka, Tokopedia atau Go-Jek bisa saja dibeli oleh pihak lain. Inilah yang terjadi pada **Tokobagus** dan **Berniaga** yang dibeli **OLX**, atau situs media social **Koprol** yang dibeli yahoo!.

Khusus untuk Koprol, ini menjadi perusahaan teknologi informasi pertama Indonesia yang dibeli perusahaan asing. Nilainya tidak disebutkan secara pasti, tapi beredar kabar bisa mencapai US \$1 juta atau sekitar Rp 13 miliar ([Berapa Duit Sih Yahoo! Membeli Koprol?](#)^c). Sayangnya, Koprol tidak bisa berkembang sejak di akuisisi yahoo! dan kini telah ditutup.

Di luar negeri, praktik pembelian startup ini sudah hal yang lumrah. Aplikasi chatting **WhatsApp** dibeli **Facebook** seharga US \$19 miliar (sekitar Rp 247 triliun). **Instagram** dibeli **Facebook** seharga US \$1 miliar (sekitar Rp 13 triliun). Dan situs media social **LinkedIn** yang dibeli **Microsoft** seharga US \$26,2 miliar (sekitar Rp 340 triliun). Nilai uang yang sangat fantastis.

^a<http://tekno.kompas.com/read/2016/08/05/15280007/ini.daftar.investor.rp.7.2.triliun.go-jek>

^b<http://tekno.kompas.com/read/2017/08/17/20315867/tokopedia-umumkan-peroleh-pendanaan-rp-14-7-triliun-dari-alibaba>

^c<http://tekno.kompas.com/read/2011/01/07/17450352/berapa.duit.sih.yahoo.membeli.koprol>

2.4 MySQL Sebagai Perusahaan Global

Sejak di nakhodai oleh Marten Mickos sebagai CEO, MySQL AB terus berkembang⁴.

Di tahun 2002, MySQL AB membuka kantor di Amerika Serikat (selain kantor pusat di Swedia).

⁴<http://buytaert.net/the-history-of-mysql-ab>

Saat itu pengguna tercatat sekitar 3 juta instalasi MySQL di seluruh dunia, dengan pendapatan sekitar US \$6.5 juta dari 1,000 pengguna berbayar.

Di tahun 2003, MySQL AB kembali membuka pendanaan dan mendapat US \$19.5 juta dari **Benchmark Capital** dan **Index Ventures**. MySQL di download hingga 30.000 setiap hari dengan 4 juta instalasi aktif. Pendapatan tahun itu sekitar US \$12 juta.

Untuk lebih meningkatkan pendapatan, MySQL mencoba berfokus ke pasar corporate (perusahaan besar) serta menawarkan lisensi berlangganan yang harus diperbarui secara berkala. MySQL AB menutup tahun 2004 dengan pendapatan sekitar US \$20 juta.

MySQL versi 5 dirilis pada tahun 2005 dengan berbagai fitur baru, seperti *stored procedures*, *triggers*, *views*, *cursors*, *distributed transactions*, *federated storage engines*, dll. Pendapatan MySQL AB tahun 2005 sekitar US \$34 juta yang berasal dari 3400 pengguna berbayar.

Kemajuan MySQL ini turut mengusik sang raksasa database: **Oracle**. Karena tentu saja bisa menjadi ancaman di kemudian hari. Sebagai langkah awal, di tahun 2005 Oracle membeli **Innibase**, sebuah perusahaan kecil yang mengembangkan sistem **InnoDB** untuk MySQL.

Pada tahun 2006, CEO MySQL AB Marten Mickos mengungkapkan bahwa Oracle ingin membeli MySQL. Penawaran ini ditolak, dan direspon oleh CEO Oracle **Larry Ellison** sebagai berikut:

“We’ve spoken to them, in fact we’ve spoken to almost everyone. Are we interested? It’s a tiny company. I think the revenues from MySQL are between \$30 million and \$40 million. Oracle’s revenue next year is \$15 billion.”

Terjemahan:

“Kami sudah berbicara dengan mereka, dan faktanya kami sudah berbicara dengan semua orang. Apakah kami tertarik? Itu sebuah perusahaan kecil. Saya rasa pendapatan MySQL berkisar antara \$30 juta hingga \$40 juta. Pendapatan Oracle tahun depan adalah \$15 miliar”.

Kembali, untuk menjegal MySQL, Oracle membeli **Sleepycat**, perusahaan yang mengembangkan *Berkeley DB transactional storage engine* untuk MySQL. Namun hal ini tidak berdampak besar karena Berkeley DB relatif jarang dipakai.

Marten Mickos mengungkap fakta bahwa MySQL bersiap untuk melakukan IPO di tahun 2008. Pendapatan nanti di taksir bisa mencapai US \$100 juta di tahun 2008 nanti.

Di tahun 2006 ini MySQL AB kembali membuat pendanaan ketiga yang diisukan sebesar US \$300 juta. MySQL sendiri sudah memiliki 320 orang karyawan di [25 negara](#).⁵ 70 persen diantaranya bekerja dari rumah.

Dengan 8 juta pengguna aktif, MySQL AB menutup tahun 2006 dengan pendapatan sebesar [US \\$50 juta](#)⁶. Untuk tahun 2007 (satu tahun sebelum jadwal IPO), pendapatan MySQL AB sudah mencapai US \$70 juta.

⁵http://archive.fortune.com/2006/05/31/magazines/fortune/mysql_greatteams_fortune/index.htm

⁶<https://www.cnet.com/news/mysql-hits-50-million-revenue-plans-ipo/>

2.5 IPO yang Batal dan Akuisisi oleh Sun Microsystem

Memasuki tahun 2008, adalah jadwal untuk IPO MySQL. Namun hal ini tidak menyurutkan berbagai perusahaan untuk menawar MySQL. Salah satunya **Sun Microsystem**, yang juga mengembangkan bahasa pemrograman **JAVA**.

Menurut Monty, di akuisisi oleh Sun lebih baik daripada IPO. Alasannya, Sun telah lama dikenal sebagai perusahaan yang mendukung Open Source. Bahasa pemrograman JAVA juga dikembangkan sebagai project Open Source. Sehingga tidak ada kekhawatiran bahwa MySQL akan menjadi aplikasi yang tertutup. Sun juga perusahaan besar dan diyakini bisa mengembangkan MySQL lebih jauh lagi.

Merasa hal tersebut merupakan pilihan yang terbaik, pada tanggal 16 January 2008, Sun Microsystem resmi membeli MySQL AB seharga **US \$ 1 miliar (sekitar Rp 13 triliun)**. Seluruh tim yang ada di MySQL AB (termasuk Monty) juga ikut menjadi karyawan Sun.



Gambar: Akuisisi MySQL oleh Sun Microsystem

2.6 Raja Database yang Mulai Mendekat: Oracle

Harapan agar MySQL bisa menjadi lebih maju di bawah Sun ternyata tidak terlaksana. Dikarenakan berbagai alasan, Michael Widenius (Monty) dan David Axmark, 2 pendiri MySQL AB memutuskan keluar dari Sun (artinya juga keluar dari MySQL). Termasuk mantan CEO MySQL Marten Mickos yang juga menyusul keluar dari Sun.

Secara bertahap, programmer awal yang mengembangkan MySQL juga banyak yang hengkang. Terlebih beredar kabar bahwa Oracle ingin mengakuisisi Sun. Yang secara tidak langsung juga akan membeli MySQL.

Dan hal itu benar-benar terjadi. Pada tanggal 20 April 2009, atau satu tahun setelah MySQL dibeli Sun, Oracle mengakuisisi Sun Microsystem dengan harga **US \$ 7.4 miliar (sekitar Rp 96 triliun)**. Proses akuisisi ini disetujui oleh pemerintah Amerika Serikat pada tanggal 20 Agustus 2009.

2.7 Gerakan “Save MySQL”

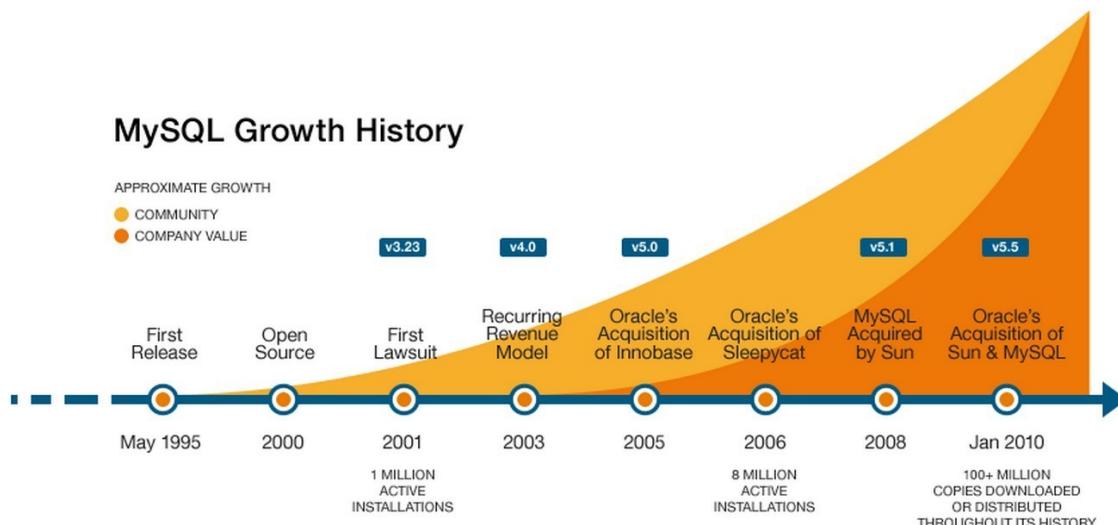
Kabar pembelian Sun oleh Oracle menjadi perhatian banyak pihak, terutama kalangan programmer (dan juga Monty) yang khawatir dengan masa depan MySQL. Sangat mungkin MySQL akan dihentikan oleh Oracle karena merupakan produk database saingan.

Proses akuisisi perusahaan sebesar Sun butuh waktu dan proses yang cukup lama, karena Sun tidak hanya berkantor di Amerika, tetapi juga Eropa dan di beberapa negara lain. Sekedar tambahan, proses akuisisi perusahaan besar harus disetujui pemerintah tempat dimana perusahaan itu berada. Ada aturan anti-monopoli yang harus dipenuhi.

Monty membuat sebuah gerakan agar Komisi Ekonomi Eropa menolak proses akuisisi ini. Gerakan yang dinamakan “Save MySQL” berhasil mendapat dukungan dari lebih 50.000 programmer dari seluruh dunia. Berbagai pemimpin organisasi Free Software juga ikut dalam gerakan ini.

Oracle tentu juga ikut bernegosiasi. Agar bisa disetujui, Oracle berjanji untuk terus mengembangkan MySQL, setidaknya hingga tahun 2015. Selain itu MySQL juga tetap menyediakan dual lisensi sebagaimana yang ada selama ini, dimana versi yang gratis (lisensi GPL) tetap tersedia. Akhirnya, komisi Uni Eropa menyetujui proses akuisisi pada tanggal 21 Januari 2010.

Menurut bocoran dokumen WikiLeaks, persetujuan akuisisi Sun oleh Oracle di Uni Eropa juga dipengaruhi karena ada tekanan dari departemen kehakiman Amerika Serikat. Dimana pemerintah Amerika Serikat juga bertindak atas desakan dari Oracle.



Gambar: Sejarah Perjalanan MySQL, sumber: datasciencecentral.com

2.8 MySQL di bawah Oracle



Gambar: MySQL - Oracle

Sesuai dengan janji Oracle, MySQL tetap terus dikembangkan. Meskipun sudah lewat dari tahun 2015, Saat ini MySQL dengan lisensi GPL (free) masih tersedia. Nama besar MySQL juga sudah terlanjur populer. Di berbagai kampus IT, hampir selalu menggunakan MySQL sebagai aplikasi database.

MySQL masih tetap menjadi kepanjangan dari huruf "M" dari sistem pengembangan web open source LAMP : Linux, Apache, MySQL dan PHP. Akan tetapi huruf "M" ini secara perlahan akan berganti menjadi **MariaDB**.

2.9 MariaDB: Sebuah Harapan Baru



Gambar: Logo MariaDB

Proses akuisisi MySQL oleh Oracle menjadi ancaman tersendiri bagi kalangan penggiat open source, termasuk mayoritas web programmer di seluruh dunia. Bagaimana jika suatu saat MySQL tidak lagi tersedia secara gratis dan menjadi aplikasi tertutup?

Khawatir akan situasi ini, Michael Widenius (Monty), yang tidak lain adalah sang pendiri MySQL, membuat perusahaan baru: **Monty Program AB**. Perusahaan ini dibentuk dengan tujuan mengembangkan aplikasi database baru yang diberi nama **MariaDB**.

Mirip seperti kelahiran MySQL, MariaDB juga tidak dibuat dari awal, tetapi menggunakan source code MySQL yang sudah tersedia (dan memang boleh dipakai oleh siapa saja, selama dalam lisensi GPL).

Dalam istilah project Open Source, MariaDB merupakan "*fork project*" dari MySQL. MariaDB pada dasarnya merupakan aplikasi MySQL dengan sedikit modifikasi, dan tetap berfungsi sebagaimana layaknya MySQL.

Dengan demikian, MariaDB bisa menjadi "*drop-in replacement*", yakni bisa langsung menggantikan MySQL tanpa perlu perubahan di sisi code program. Pengguna cukup melakukan backup,

menghapus aplikasi MySQL, menginstall MariaDB, dan mengcopy database lama. Tidak perlu perubahan apapun di sisi coding.

Inspirasi nama MariaDB berasal dari anak perempuan Monty, “**Maria**”. Nama MySQL sendiri juga berasal dari “**My**”, anak perempuan pertama Monty.

Di lain pihak, beberapa mantan karyawan MySQL (yang juga keluar dari Sun/Oracle) mendirikan perusahaan baru yang diberi nama **SkySQL AB**. Perusahaan ini menawarkan jasa support untuk MySQL (sebagai alternatif dari Oracle) serta berkontribusi dalam mengembangkan MariaDB. Dengan kata lain, SkySQL AB menyediakan jasa support untuk MySQL dan MariaDB.



Gambar: Logo SkySQL, sumber: t3n.de

Untuk **Monty Program AB**, Monty berusaha mengumpulkan tim yang dulunya mengembangkan MySQL agar ikut bergabung mengembangkan MariaDB. Tim ini juga termasuk karyawan SkySQL AB. Pada tanggal 22 Januari 2009, versi pertama dari MariaDB, yakni **MariaDB 5.1** resmi dirilis.

Penamaan versi MariaDB pada awalnya dirancang mengikuti versi MySQL. Jika MySQL mengeluarkan versi 5.2, maka akan diikuti oleh MariaDB versi 5.2. Seluruh fitur MySQL 5.2 juga tersedia di MariaDB 5.2 plus berbagai fitur tambahan. Penamaan versi seperti ini terus berlangsung hingga MySQL 5.5 dan MariaDB 5.5 yang dirilis pada tahun 2002.

Setelah itu, versi MariaDB “lompat” ke 10.0. Perubahan ini dilakukan karena proses modifikasi MySQL memakan waktu yang cukup lama. MariaDB ingin lepas dari MySQL dengan menambahkan fitur baru yang tidak ada di MySQL.

Fitur tambahan ini sebagian besar hanya untuk keperluan advanced yang sangat spesifik. Untuk penggunaan sehari-hari, bisa dianggap bahwa MariaDB “sama persis” dengan MySQL.

2.10 MariaDB Foundation dan MariaDB Corporation

Apa yang terjadi kepada MySQL membuat Monty merancang sebuah sistem supaya MariaDB tidak bisa dibeli oleh perusahaan lain. Caranya adalah dengan membuat pemisahan antara perusahaan yang mencari keuntungan dan organisasi non profit untuk pengembangan MariaDB.

Pada bulan Desember 2012, **Michael Widenius (Monty)**, **David Axmark**, and **Allan Larsson** (tiga orang pendiri MySQL AB dulu), mengumumkan pendirian organisasi non profit yang diberi nama **MariaDB Foundation**.



Gambar: Logo MariaDB Foundation

Tujuan pendirian MariaDB Foundation adalah untuk melindungi pengembangan MariaDB dan memastikan tidak ada satu pihak yang bisa mengontrol pengembangan MariaDB. MariaDB akan tetap menjadi project Open Source di bawah MariaDB Foundation.

Di bulan April 2013, **Monty Program AB** bergabung dengan **SkySQL AB** dan menjadi pendorong pengembangan MariaDB. Selang 1 tahun sesudahnya, pada 1 Oktober 2014, **SkySQL AB** berubah nama menjadi **MariaDB Corporation**⁷. Perubahan nama ini mencerminkan fokus SkySQL AB ke MariaDB.

Sampai disini, terdapat 2 buah badan yang memiliki nama MariaDB, yakni **MariaDB Foundation** dan **MariaDB Corporation**. Apa bedanya?

MariaDB Foundation dibentuk sebagai pelindung (*guardian*) dari MariaDB. Proses pengembangan database server MariaDB dilakukan disini. MariaDB Foundation bersifat non profit (tidak mencari keuntungan). Sedangkan **MariaDB Corporation** adalah perusahaan yang mencari keuntungan dari support untuk pemakai MariaDB.

Dengan pemisahan seperti ini, MariaDB Corporation bisa saja dibeli oleh perusahaan lain, akan tetapi pengembangan aplikasinya tetap berada di MariaDB Foundation.

Trik seperti ini juga digunakan oleh beberapa perusahaan open source lainnya. Pernah dengar **Mozilla Foundation**? ia adalah organisasi non profit yang membawahi **Mozilla Corporation**. Sama juga seperti **WordPress** yang dikembangkan secara non profit (wordpress.org), dengan perusahaan **Automattic** sebagai tempat wordpress mencari keuntungan (wordpress.com).

Masalah trademark serta komplikasi hukumnya memang cukup memusingkan. Jika berminat, anda bisa membaca langsung dari blog Monty: [MariaDB foundation trademark agreement](#)⁸.

MariaDB Foundation beralamat di mariadb.org⁹. Disinilah kita akan mendownload aplikasi database server MariaDB.

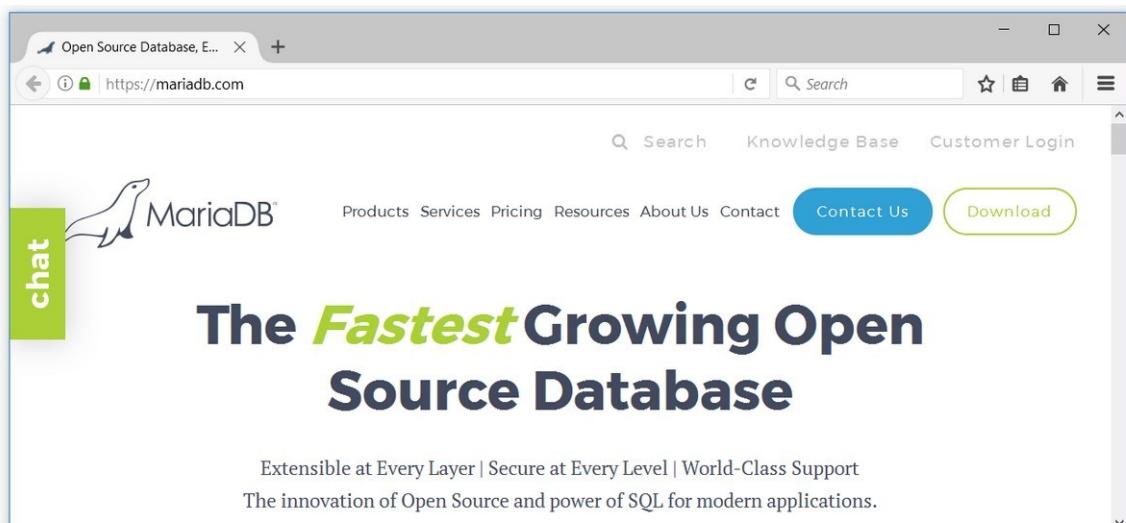
Sedangkan **MariaDB Corporation** beralamat di mariadb.com¹⁰. Website ini khusus untuk pelanggan berbayar MariaDB, terutama perusahaan besar yang butuh support langsung seperti konsultasi, training, dan maintenance server.

⁷ <https://mariadb.org/why-skysql-becoming-mariadb-corporation-will-be-good-for-the-mariadb-foundation/>

⁸ <http://monty-says.blogspot.co.id/2014/10/mariadb-foundation-trademark-agreement.html>

⁹ <https://mariadb.org>

¹⁰ <https://mariadb.com>



Gambar: Tampilan situs mariadb.com

2.11 Jadi, pilih MySQL atau MariaDB?

Menurut saya, kedepannya MariaDB akan makin banyak dipakai. Terlebih lagi, tim yang dulunya mengembangkan MySQL (termasuk pendiri MySQL), ada dibalik MariaDB.

Berdasarkan klaim dari Monty, MariaDB merupakan versi MySQL yang lebih cepat, lebih efisien, dan memiliki berbagai fitur tambahan yang tidak ada di MySQL.

MariaDB juga telah mendapat [pendanaan¹¹](#) hingga [US\\$ 40 juta¹²](#) dari investor besar seperti Intel, California Technology Ventures, Finnish Industry Investment, Open Ocean Capital dan Spintop Private Partners.

Beberapa aplikasi (terutama yang berbasis open source) juga mulai beralih ke MariaDB. Aplikasi populer untuk pengembangan web programming seperti XAMPP sudah tidak lain menggunakan MySQL dan menukarnya dengan MariaDB. Wikipedia juga menyatakan akan beralih dari MySQL ke MariaDB. Secara perlahan aplikasi lain akan menyusul.

Jika anda Googling dengan keyword “MySQL vs MariaDB”, hampir seluruhnya menyatakan MariaDB lebih baik, terutama di sisi performa dan fitur. Contohnya bisa kesini: [MariaDB versus MySQL - Features¹³](#) dan [10 reasons to migrate to MariaDB \(if still using MySQL\)¹⁴](#).

2.12 Bagaimana dengan MySQL?

Sejak berada di bawah Oracle, pengembangan MySQL tidak cukup cepat. Selain itu banyak pihak masih khawatir suatu saat Oracle bisa menghentikan MySQL.

¹¹<https://mariadb.com/about-us/investors>

¹²<https://www.crunchbase.com/organization/skysql#/entity>

¹³<https://mariadb.com/kb/en/mariadb/mariadb-vs-mysql-features/>

¹⁴<https://seravo.fi/2015/10-reasons-to-migrate-to-mariadb-if-still-using-mysql>

Meskipun begitu, nilai jual “MySQL” masih sangat populer. Nama MySQL juga sudah terlanjur terkenal. Butuh waktu bagi MariaDB agar bisa menggantikan MySQL. Berbagai kalangan yang tidak update tentang perkembangan teknologi juga masih akrab dengan MySQL.

Dalam penggunaan sehari-hari, MySQL dan MariaDB ini tidak banyak berbeda. Seluruh materi yang akan anda pelajari dalam buku ini bisa diterapkan ke MySQL maupun MariaDB. Untuk web programming seperti PHP, seluruh fungsi MySQL yang ada juga sudah otomatis didukung di dalam MariaDB. Fungsi seperti `mysqli_connect()` bisa langsung terhubung ke MySQL maupun MariaDB, tanpa perubahan code apapun.

Selain itu, hampir seluruh fitur yang ada di MySQL, juga tersedia di MariaDB.

Dalam bab ini kita telah mengikuti kisah perjalanan MySQL sejak awal berdirinya hingga hadirnya MariaDB.

Ketika pertama kali melihat XAMPP sudah tidak lagi menggunakan MySQL, saya juga penasaran apa yang terjadi. Dan makin aneh ketika yang menemukan bahwa yang mengembangkan MariaDB juga pendiri MySQL. Ternyata ada cerita yang cukup berliku di baliknya.



Jika ada tertarik membaca kisah MySQL dan MariaDB langsung dari sang “creator”, bisa mendownload sebuah slide presentasi yang dibuat sendiri oleh Monty: [The MySQL-MariaDB story¹⁵](#).

Dalam bab selanjutnya kita akan membahas sedikit teori terkait database serta pengantar bahasa SQL (Structured Query Language).

¹⁵ <http://myy.haaga-helia.fi/~dbms/dbtechnet/download/MySQL-MariaDB-story-haaga-helia.pdf>

3. Pengantar Teori Database dan SQL

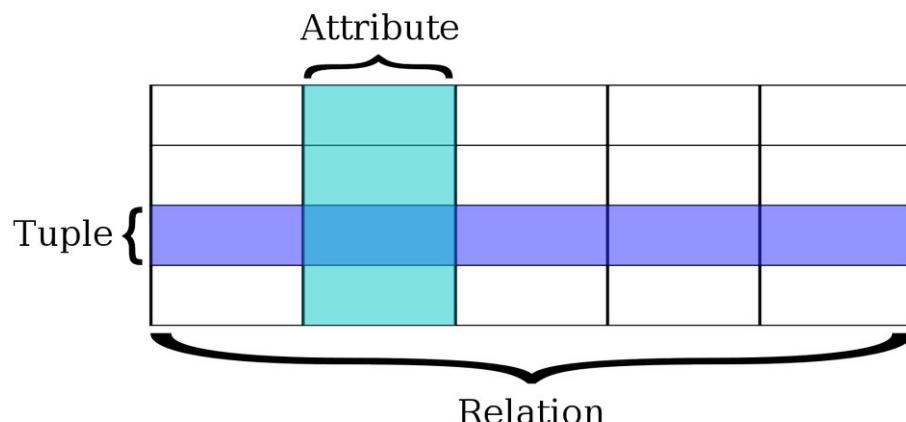
Sebelum kita masuk ke cara instalasi MySQL dan MariaDB, saya ingin membahas sekilas tentang **teori database** serta **pengertian bahasa SQL**. Materi ini dirasa cukup penting karena terdapat istilah-istilah khusus yang akan kita pakai sepanjang pembahasan nantinya.

- i** Istilah-istilah yang saya bahas disini baru sebagian kecil dari teori database. Materi tentang database sendiri sangat luas. Di jurusan ilmu komputer / IT, terdapat mata kuliah khusus yang membahas database. **DBA (Database Administrator)** juga bisa menjadi pilihan karir yang *kerjaannya* hanya menangani masalah database. Jika anda berminat untuk mempelajari hal ini lebih lanjut, bisa mencari buku khusus yang membahas teori database.

3.1 Database, Attribute, dan Tuple

Dalam *relational database model*, sebuah **database** adalah *kumpulan relasi yang saling terhubung satu sama lain*. **Relasi** merupakan istilah dalam relational database, tapi kita lebih familiar menyebutnya sebagai **tabel**.

Setiap tabel terdiri dari **baris** dan **kolom**. Dalam *relational database*, kolom atau column disebut juga sebagai **attribute**, sedangkan baris atau row disebut sebagai **tuple**. Istilah ini hanya sekedar penamaan. Agar lebih mudah, saya tetap menggunakan istilah tabel, kolom dan baris pada pembahasan sepanjang buku ini.



Gambar: Ilustrasi relation, attribute dan tuple, sumber: wikipedia.org

Selain itu terdapat juga *record* dan *field*. **Record** adalah satu baris tabel (sama artinya dengan row atau tuple), sedangkan **field** merujuk kepada satu data di dalam sel tabel.

Perhatikan contoh tabel berikut:

ID Prov	Nama Provinsi	Ibu Kota Provinsi	Luas Wilayah	Populasi	Tanggal Diresmikan
1	Jakarta	Jakarta	740	10.012.271	28 Agustus 1961
2	Banten	Serang	9.019	11.704.877	4 Oktober 2000
3	Kalimantan Selatan	Banjarmasin	36.805	3.922.790	7 Desember 1956
4	Kepulauan Riau	Tanjung Pinang	8.084	1.917.415	25 Oktober 2002
5	Papua Barat	Manokwari	114.566	849.809	4 Oktober 1999
6	Maluku Utara	Sofifi	42.960	1.138.667	4 Oktober 1999

Gambar: Tabel provinsi

Tabel diatas berisi 6 nama provinsi di Indonesia. Nama-nama kolom yakni ID Prov, Nama Provinsi, Ibu Kota Provinsi, Luas Wilayah, Populasi dan Tanggal Diresmikan adalah **column** atau **tuple**.

Setiap baris provinsi beserta datanya merupakan sebuah **record** atau **row**, seperti 3, Kalimantan Selatan, Banjarmasin, 36.805, 3.922.790, 7 Desember 1956. Yakni seluruh data di baris ke-3.

Sedangkan 1 data sel tabel seperti Manokwari, 11704877 atau 4 Oktober 2000 adalah sebuah **field**.

Di dalam MySQL, setiap data memiliki karakteristik masing-masing yang dikenal sebagai tipe data. Untuk kolom Nama Provinsi dan Ibu Kota Provinsi, bisa disimpan ke dalam tipe data teks (CHAR). Untuk kolom ID Prov, Luas Wilayah dan Populasi bisa disimpan ke dalam tipe data angka bulat (INTEGER). Sedangkan untuk kolom Tanggal Diresmikan bisa disimpan ke dalam tipe data tanggal (DATE).



Data tentang Nama Provinsi serta nama Kabupaten dan Kota, saya ambil dari Wikipedia: [Daftar Provinsi di Indonesia](https://id.wikipedia.org/wiki/Daftar_Provinsi_di_Indonesia)¹ dan [Daftar Kabupaten dan Kota di Indonesia](https://id.wikipedia.org/wiki/Daftar_Kabupaten_dan_kota_di_Indonesia)².

3.2 Candidate Key, Primary Key dan Foreign Key

Kekuatan dari relational database ada di kemampuan menggabungkan berbagai tabel. Untuk aplikasi sederhana kadang ada yang hanya perlu satu tabel saja, tapi dalam kebanyakan aplikasi kita butuh beberapa tabel.

Sebagai contoh, tabel sebelumnya berisi nama-nama provinsi. Bagaimana dengan daftar nama kota dan kabupaten? Ini layak menjadi tabel kedua. Daftar kecamatan menjadi tabel ketiga, serta daftar kelurahan menjadi tabel keempat. Tanpa pemisahan seperti ini, tabel kita menjadi sangat besar dan susah dikelola.

Setiap baris di dalam tabel setidaknya harus memiliki sebuah kolom unik. Unik disini maksudnya tidak boleh ada data yang berulang (data yang sama). Kolom ID Prov, Nama Provinsi dan Ibu

¹https://id.wikipedia.org/wiki/Daftar_Provinsi_di_Indonesia

²https://id.wikipedia.org/wiki/Daftar_Kabupaten_dan_kota_di_Indonesia

Kota Provinsi adalah kolom yang unik. Sangat kecil kemungkinan (dan sepertinya tidak akan pernah ada) terdapat nama provinsi dan nama ibu kota provinsi yang sama.

Kolom tabel yang memenuhi syarat ini (tidak memiliki data yang berulang), dikenal sebagai **candidate key** (kunci kandidat).

Candidate key adalah satu atau beberapa kolom di dalam tabel yang bisa mengidentifikasi tiap baris dari tabel tersebut. Selain tidak boleh ada data yang sama, kolom yang dikategorikan sebagai candidate key juga tidak boleh memiliki data kosong.

Dalam sebuah tabel, bisa jadi terdapat beberapa candidate key. Dari daftar candidate key ini, dipilih salah satu yang berfungsi sebagai **primary key** (kunci utama). **Primary key** adalah kolom yang akan menjadi identitas dari setiap baris tabel.

Pemilihan kolom mana yang akan menjadi primary key bisa mempertimbangkan beberapa aspek, karena inilah kolom yang akan berfungsi sebagai penghubung satu tabel dengan tabel lain. Dalam tabel nama provinsi sebelumnya, kolom ID Prov lebih cocok dijadikan sebagai **primary key**.

Bagaimana dengan kolom Nama Provinsi? Ini pun bisa dijadikan primary key, tergantung keputusan kita mana yang dirasa paling pas selama syarat-syaratnya dipenuhi. Yakni tidak boleh ada data yang sama dan tidak boleh kosong.

Istilah key berikutnya yakni *foreign key*. **Foreign key** (kunci tamu) adalah primary key dari tabel lain yang berada di tabel saat ini. Untuk bisa memahaminya, saya akan tambah satu tabel lagi:

Tabel Provinsi

ID Prov	Nama Provinsi	Ibu Kota Provinsi	Luas Wilayah	Populasi	Tanggal Diresmikan
1	Jakarta	Jakarta	740	10.012.271	28 Agustus 1961
2	Banten	Serang	9.019	11.704.877	4 Oktober 2000
3	Kalimantan Selatan	Banjarmasin	36.805	3.922.790	7 Desember 1956
4	Kepulauan Riau	Tanjung Pinang	8.084	1.917.415	25 Oktober 2002
5	Papua Barat	Manokwari	114.566	849.809	4 Oktober 1999
6	Maluku Utara	Sofifi	42.960	1.138.667	4 Oktober 1999

Tabel Kota dan Kabupaten

ID Kota	Nama Kota / Kabupaten	Pusat pemerintahan	Luas Wilayah	Populasi	ID Prov
1	Jakarta Barat	Kembangan	127	2.260.341	1
2	Jakarta Pusat	Menteng	50	861.531	1
3	Jakarta Selatan	Kebayoran Baru	141	2.164.070	1
4	Jakarta Timur	Cakung	188	2.693.896	1
5	Jakarta Utara	Koja	137	1.468.840	1
6	Kabupaten Tangerang	Tigaraksa	960	3.154.790	2
7	Kota Cilegon	Kota Cilegon	175	374.559	2
8	Kota Tangerang	Kota Tangerang	165	1.798.601	2
9	Kabupaten Fakfak	Fakfak	14.320	66.828	5
10	Kabupaten Raja Ampat	Waisai	71.605	27.071	5

Gambar: Tabel Provinsi + Tabel Kota dan Kabupaten

Sekarang terdapat tambahan tabel baru yang berisi nama kota dan kabupaten. Dapatkah anda menebak kota Cilegon itu berada di provinsi mana?

Perhatikan kolom ID Prov di sisi paling kanan tabel kota dan kabupaten. Untuk baris kota Cilegon, kolom ID Prov bernilai 2. Sekarang, kita tinggal lihat di tabel provinsi, provinsi mana yang juga memiliki ID Prov 2. Hasilnya adalah Banten.

Inilah mekanisme relasi di dalam relational database. Kolom ID Prov yang terdapat di tabel kota dan kabupaten bertindak sebagai **foreign key** untuk tabel tersebut. Kemudian, kolom manakah yang menjadi **primary key** dari tabel kota dan kabupaten? Yup, kolom **ID Kota**.

Prinsip yang sama juga bisa digunakan untuk membuat tabel kecamatan serta tabel kelurahan. Setiap tabel akan memiliki **primary key** dan juga **foreign key** (apabila dibutuhkan).



Dalam prakteknya nanti, *primary key* dan *foreign key* tidak harus ditulis. MySQL tetap membolehkan kita membuat tabel tanpa keduanya. Meskipun secara teori, setiap tabel setidaknya harus memiliki sebuah *primary key*.

3.3 Referential Integrity

Salah satu fungsi dari **primary key** dan **foreign key** adalah untuk membuat data yang valid.

Dari dua tabel kita sebelum ini, terdapat suatu aturan yang harus diikuti agar data tabel tidak berantakan. Yakni, untuk bisa menginput data baru ke dalam tabel kota dan kabupaten, **nama provinsinya harus ada terlebih dahulu di tabel provinsi**, karena kita perlu data ID Prov.

Di dalam tabel provinsi hanya terdapat 6 provinsi. Apabila, saya ingin menginput kota Medan ke dalam tabel kota dan kabupaten, saya tidak bisa mendapatkan data ID Prov. Karena data provinsi Sumatera Utara belum ada di tabel provinsi.

Dengan kata lain, saya harus menginput nama provinsi Sumatera Utara terlebih dahulu di tabel provinsi, kemudian barulah kota Medan bisa diinput ke dalam tabel kota dan kabupaten. Batasan seperti ini dikenal sebagai **referential integrity**.

Referential integrity adalah penerapan aturan bahwa untuk setiap *foreign key* yang terdapat pada suatu tabel, harus ada nilainya di tabel asal kolom tersebut.

Dengan menggunakan MySQL, kita bisa membuat aturan seperti ini. Hasilnya akan terjadi error jika seseorang menginput data ID Prov yang nilainya belum tersedia.

3.4 Normalisasi Database

Normalisasi database (*database normalization*) adalah proses penyusunan kolom dan tabel untuk meminimalkan redundansi data (data yang berulang). Normalisasi akan membagi tabel besar menjadi beberapa tabel kecil yang saling terhubung. Hal ini dilakukan agar mudah dalam mengatur, serta mengorganisasi data.

Materi tentang normalisasi database sebenarnya cukup penting. Karena disinilah akan dibahas bagaimana cara mendesain struktur tabel untuk membuat database yang efisien.

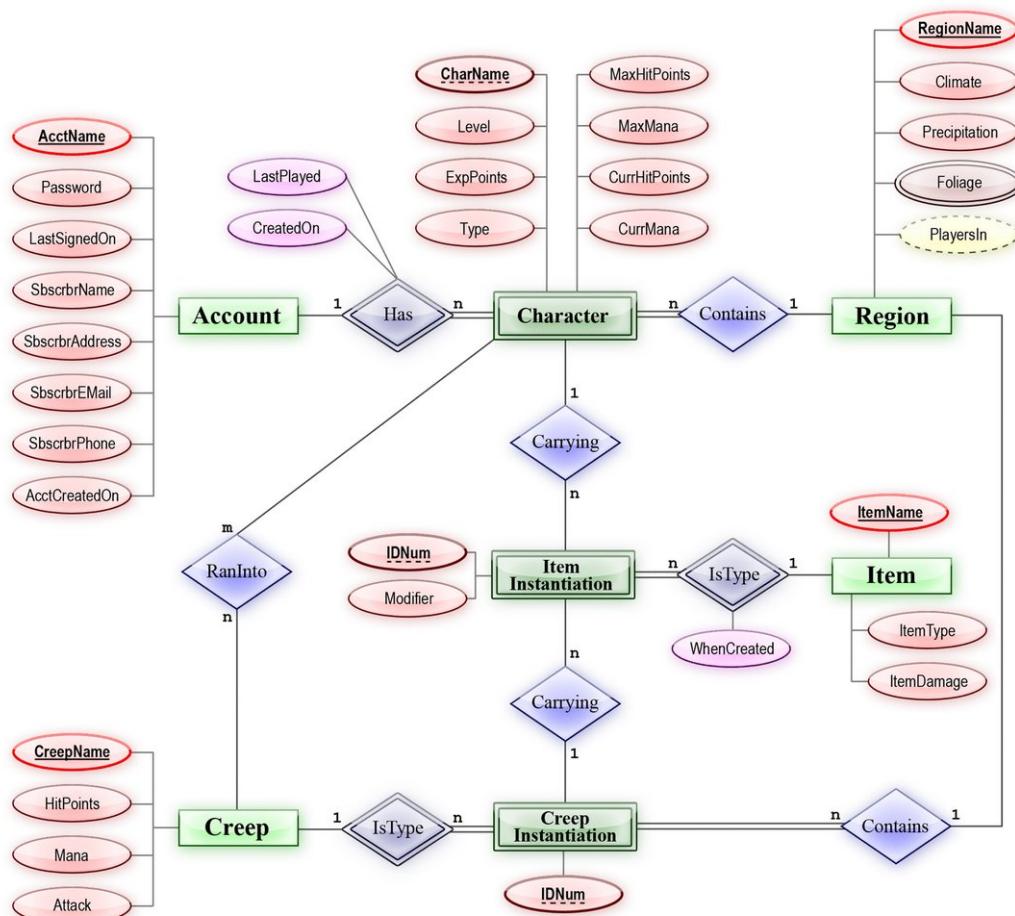
Namun materi ini juga lumayan rumit dan bikin cepat ngantuk (yang kuliah di IT pasti sudah punya pengalaman). Oleh karena itu saya tidak akan membahas lebih jauh tentang normalisasi database. Jika anda tertarik, bisa cari buku yang khusus membahas teori database.

3.5 Entity Relationship Diagram (ERD)

ERD (Entity Relationship Diagram) adalah sebuah diagram yang menggambarkan bagaimana struktur design database yang akan dibuat. Dengan kata lain, ERD adalah *blueprint* dari database. Ibarat arsitek, ERD merupakan gambar rumah yang akan dibangun.

Di dalam ERD bisa terlihat semua tabel yang menyusun suatu database, termasuk *primary key* masing-masing tabel, *foreign key*, serta kolom apa saja yang tersedia di setiap tabel.

Sama seperti normalisasi database, teori seputar ERD juga tidak akan saya bahas dalam buku ini. Jika anda tertarik, bisa cari buku yang khusus membahas teori database (kita hanya fokus ke praktik penggunaan aplikasi database dengan MySQL).



Gambar: Contoh sebuah diagram ERD

3.6 Table Index

Di dalam database, **index** adalah sebuah struktur data yang diimplementasikan oleh RDBMS (seperti MySQL) untuk mempercepat proses pembacaan data.

Untuk membuat index, kita tinggal memanggil perintah tertentu dan selesai. MySQL akan mengurus sisanya.

Index menggunakan algoritma tertentu dan “menempel” ke sebuah kolom dari suatu tabel. Di dalam MySQL, kolom yang ditetapkan sebagai *primary key* akan otomatis di-index.

Di dalam satu tabel, bisa saja terdapat beberapa kolom yang di index, bahkan semua kolom bisa di-index. Pertanyaannya, jika index bisa mempercepat proses pembacaan, kenapa tidak semua kolom saja kita index?

Index mempercepat proses pembacaan data namun memperlama proses perubahan dan penambahan data. Ini karena untuk setiap data yang ditambah atau diubah, index harus ditulis ulang oleh MySQL.

Lebih jauh tentang index akan saya bahas kembali saat kita praktek dengan MySQL nanti.

3.7 Pengertian Structured Query Language (SQL)

SQL (Structured Query Language) adalah bahasa pemrograman khusus yang digunakan untuk berkomunikasi dengan aplikasi RDBMS (seperti MySQL). Bahasa SQL terlihat seperti perintah sederhana yang berisi instruksi dalam bahasa Inggris, seperti "SELECT nama_provinsi, populasi FROM provinsi".

Bahasa SQL yang akan kita praktikkan dalam buku ini tidak hanya dipakai oleh MySQL atau MariaDB saja, tetapi juga oleh aplikasi RDBMS lain seperti Oracle, Microsoft SQL Server, PostgreSQL, dll. Perintah SQL ini sering juga disingkat dengan sebutan ‘query’.

Sekitar 90% dari materi yang akan kita pelajari, berisi berbagai perintah bahasa SQL.

3.8 Sejarah Structured Query Language (SQL)

Bahasa SQL juga dirancang oleh Dr. Edgar F. Codd, ilmuwan yang juga mengembangkan konsep relational database model.



Gambar: Dr. Edgar F. Codd, sumber: flickr.com

Bersamaan dengan paper Dr. Edgar F. Codd pada tahun 1969 yang berisi dasar teori relational database, beliau juga mengajukan sebuah bahasa yang disebut **DSL/Alpha** untuk memanajemen data dalam relational database. Ide dari Dr.Codd ini diimplementasikan oleh IBM dengan membuat bahasa prototipe sederhana **DSL/Alpha** yang disebut **SQUARE**.

Pada tahun 1970, tim yang beranggotakan peneliti IBM **Donald D. Chamberlin** dan **Raymond F. Boyce**, mengembangkan **SQUARE** lebih lanjut menjadi **SEQUEL** (Structured English Query Language). **SEQUEL** ini digunakan untuk mengoperasikan prototipe RDBMS pertama IBM, **System R**.

Di kemudian hari, **SEQUEL** berubah nama menjadi **SQL** karena permasalahan merk dagang (trademark) dengan sebuah perusahaan pesawat di Inggris yang terlebih dahulu telah memakai nama **SEQUEL**.

Pada akhir 1970an, perusahaan **Relational Software, Inc.** (sekarang **Oracle Corporation**) melihat potensi bahasa **SQL** dan mengembangkan sendiri versi **SQL** untuk RDBMS mereka. **Oracle V2** (versi 2) yang dirilis Juni 1979 merupakan RDBMS komersial pertama yang mengimplementasikan **SQL**.

Dengan kemudahan yang ditawarkan, **SQL** juga dipakai oleh berbagai RDBMS dengan versi **SQL** mereka masing-masing. Hal ini menimbulkan permasalahan karena penerapan bahasa **SQL** yang tidak seragam.

Pada tahun 1986, badan standar Amerika, **ANSI** (American National Standards Institute) merancang sebuah standar untuk bahasa **SQL**. Satu tahun setelahnya, **ISO** (International Organization for Standardization) juga mengeluarkan standar untuk **SQL**.

Versi terakhir standar **SQL** dirilis pada 2011, yang dinamakan **SQL 2011**. Dengan standar ini diharapkan ada keseragaman bahasa **SQL** antar aplikasi RDBMS. Artinya, bahasa **SQL** yang dipakai untuk database MySQL, juga bisa digunakan pada database Oracle maupun Microsoft SQL Server.

Meskipun sudah ada sebuah standar baku, aplikasi RDBMS juga tetap menambah ‘fitur’ **SQL** selain standar yang ada. MySQL juga memiliki perintah **SQL** yang tidak standar (hanya bisa

berjalan di MySQL saja). Namun setidaknya dasar bahasa SQL itu hampir sama. Perintah SQL untuk membuat tabel misalnya, juga dapat digunakan baik di Oracle maupun MySQL.

3.9 Jenis-jenis Perintah Bahasa SQL

Perintah atau instruksi dalam bahasa SQL dapat dikelompokkan berdasarkan jenis dan fungsinya. Terdapat 3 kelompok perintah dasar SQL: **Data Definition Language**, **Data Manipulation Language** dan **Data Control Language**.

- **Data Definition Language** (DDL) adalah jenis instruksi SQL yang berkaitan dengan pembuatan struktur tabel dan database. Termasuk diantaranya CREATE, DROP, ALTER, dan RENAME.
- **Data Manipulation Language** (DML) adalah jenis instruksi SQL yang berkaitan dengan data yang ada dalam tabel. Yakni bagaimana cara menginput data, menghapus data, update data serta membaca data yang tersimpan di dalam database. Contoh perintah SQL untuk DML adalah SELECT, INSERT, DELETE, dan UPDATE.
- **Data Control Language** (DCL) adalah jenis instruksi SQL yang berkaitan dengan manajemen hak akses dan pengguna (user). Perintah ini digunakan untuk membatasi siapa saja yang dapat mengakses database dan tabel. Perintah SQL yang termasuk ke dalam kategori DCL adalah GRANT dan REVOKE.

Selain ketiga jenis perintah SQL diatas, terdapat juga 2 kategori tambahan: **Transaction Control Language**, dan **Programmatic SQL**.

- **Transaction Control Language** (TCL) adalah perintah SQL untuk proses transaksi. Proses transaksi ini pada dasarnya digunakan untuk membuat beberapa perintah query yang dianggap sebagai satu kesatuan. Jika ada masalah, seluruh proses bisa dibatalkan. Termasuk ke dalam TCL adalah query COMMIT, ROLLBACK, dan SET TRANSACTION.
- **Programmatic SQL** berkaitan dengan sub program (*stored procedure*) maupun penjelasan mengenai struktur database (meta data table). Contoh perintah seperti adalah DECLARE, EXPLAIN, PREPARE, dan DESCRIBE.

Hingga akhir buku nanti, kita akan membahas sebagian besar dari perintah-perintah SQL ini. Namun sebelum itu tentu harus ada aplikasi database. Dalam bab selanjutnya, kita akan menginstall aplikasi yang dibutuhkan, yakni XAMPP, MySQL dan MariaDB.

4. Instalasi XAMPP, MySQL dan MariaDB

Dalam bab ini kita akan masuk ke proses instalasi XAMPP, MySQL dan MariaDB. Kenapa ada 3 aplikasi?

Saya yakin sebagian besar dari pembaca buku ini ingin belajar MySQL untuk pembuatan website (digabung dengan PHP). Aplikasi XAMPP adalah pilihan yang paling praktis, karena di dalamnya sudah terdapat web server Apache, database server MySQL dan bahasa pemrograman PHP.

Karena alasan itu pula sepanjang buku nantinya saya juga menggunakan MySQL bawaan XAMPP. Terlebih di dalam XAMPP sudah tersedia *PHPmyAdmin*. **PHPmyAdmin** adalah aplikasi berbasis web yang bisa digunakan untuk berkomunikasi dengan database MySQL maupun MariaDB.

Namun jika anda ingin menggunakan MySQL bukan untuk web programming, atau ingin mempelajari MySQL secara terpisah, bisa dengan menginstall aplikasi MySQL maupun MariaDB secara manual.



Disini saya akan jelaskan cara menginstall XAMPP, MySQL dan MariaDB. Anda hanya butuh salah satu aplikasi ini, tidak harus semuanya. Saya merekomendasikan XAMPP, karena jauh lebih praktis.

Jika di komputer / PC anda sudah terinstall aplikasi XAMPP, MySQL atau MariaDB, anda boleh melewati bab ini.

4.1 Instalasi XAMPP

XAMPP adalah salah satu aplikasi yang dikenal sebagai **AMP Stack**. AMP merupakan singkatan dari Apache Web Server, MySQL Database Server, dan PHP. Dalam pengembangan web, ketiga aplikasi ini sangat populer digunakan.

Di dalam XAMPP, sudah tersedia satu paket lengkap aplikasi pemrograman web, yang diwakili dari namanya. X (berarti cross-platform, maksudnya tersedia dalam berbagai sistem operasi), Apache Web Server, MySQL Database Server, PHP dan Perl. Tanpa menggunakan XAMPP, kita harus menginstall setiap aplikasi secara terpisah.

Karena di dalam XAMPP sudah tersedia juga database MySQL, maka kita tidak perlu lagi menginstall MySQL secara terpisah.

Silahkan download XAMPP dari website resminya di [apachefriends.org](https://www.apachefriends.org)¹. Pada halaman awal website *apachefriends*, terdapat judul besar: XAMPP - Apache + MariaDB + PHP + Perl. Betul,

¹<https://www.apachefriends.org>

saat ini XAMPP sudah beralih dari MySQL ke MariaDB. Jadi, ketika menginstall XAMPP yang akan terinstall adalah MariaDB, bukan lagi MySQL.

Pada saat buku ini saya tulis, versi terakhir dari XAMPP adalah **XAMPP 7.1.7**. Penamaan versi XAMPP saat ini mengikuti versi PHP yang terdapat di dalamnya. Dengan kata lain, XAMPP 7.1.7 berisi PHP 7.1.7. Selain itu, XAMPP 7.1.7 terdiri dari web server Apache 2.4.26 dan database server MariaDB 10.1.25.

Jika komputer/laptop yang anda gunakan terbilang baru (terutama Windows 8 dan 10), silahkan download XAMPP 7.1.7 atau versi diatasnya. Dalam beberapa kasus, seperti yang pernah saya alami, XAMPP 7.1.7 tidak bisa berjalan (error). Jika anda juga mengalami hal ini, silahkan coba install XAMPP 5.6.31.

XAMPP 5.6.31 juga menggunakan MariaDB 10.1.25 (sama seperti di XAMPP 7.1.7), yang membedakan hanya versi PHPnya saja (PHP 5.6). Jika anda menggunakan Windows XP, terpaksa harus menggunakan XAMPP 1.8.2 yang berisi PHP 5.5.36. Versi XAMPP diatas itu tidak bisa berjalan di Windows XP.

Untuk memulai proses instalasi, silahkan download XAMPP dari apachefriends.org. Atau langsung download dari salah satu link berikut:

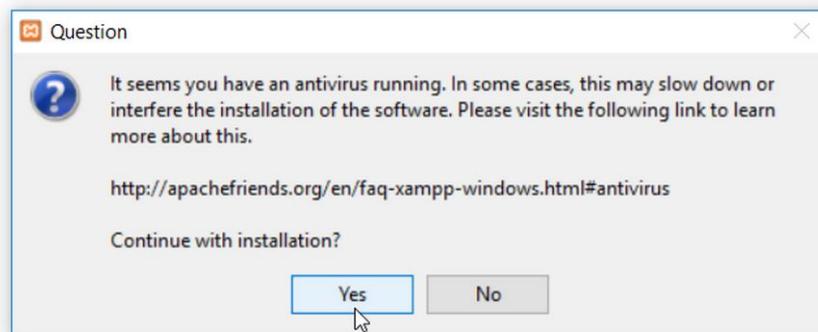
- [XAMPP 7.1.7²](https://www.apachefriends.org/xampp-files/7.1.7/xampp-win32-7.1.7-0-VC14-installer.exe) (123 MB), dengan PHP versi 7.1.7 dan MariaDB 10.1.25.
- [XAMPP 5.6.31³](https://www.apachefriends.org/xampp-files/5.6.31/xampp-win32-5.6.31-0-VC11-installer.exe) (108 MB), dengan PHP versi 5.6.31 dan MariaDB 10.1.25.
- [XAMPP 1.8.2⁴](https://sourceforge.net/projects/xampp/files/XAMPP%20Windows/1.8.2/xampp-win32-1.8.2-6-VC9-installer.exe/download) (115 MB), dengan PHP versi 5.5.36 dan MySQL 5.5.36.



Seluruh file installer XAMPP adalah untuk versi 32-bit. Saat ini belum tersedia versi khusus untuk sistem 64-bit. Namun hal ini tidak akan menjadi masalah karena aplikasi 32-bit tetap bisa berjalan di sistem 64-bit.

Setelah file aplikasi XAMPP tersedia, kita sudah bisa mulai proses instalasi. File XAMPP yang saya gunakan adalah: **xampp-win32-7.1.7-0-VC14-installer.exe**, bisa jadi versi yang anda dapatkan akan lebih baru. Silahkan double klik file ini.

Jika menggunakan anti-virus, akan tampil jendela berikut:



Gambar: Saran untuk mematikan anti virus

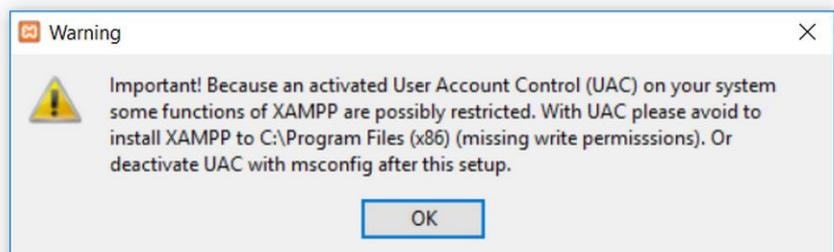
²<https://www.apachefriends.org/xampp-files/7.1.7/xampp-win32-7.1.7-0-VC14-installer.exe>

³<https://www.apachefriends.org/xampp-files/5.6.31/xampp-win32-5.6.31-0-VC11-installer.exe>

⁴<https://sourceforge.net/projects/xampp/files/XAMPP%20Windows/1.8.2/xampp-win32-1.8.2-6-VC9-installer.exe/download>

Jendela peringatan diatas berisi penjelasan bahwa saat ini program anti virus sedang berjalan. Program anti virus bisa jadi membuat proses instalasi berjalan lambat atau mengganggu proses instalasi XAMPP. Anda boleh mematikan anti virus untuk sementara (sekitar 10 menit), atau klik saja tombol Yes.

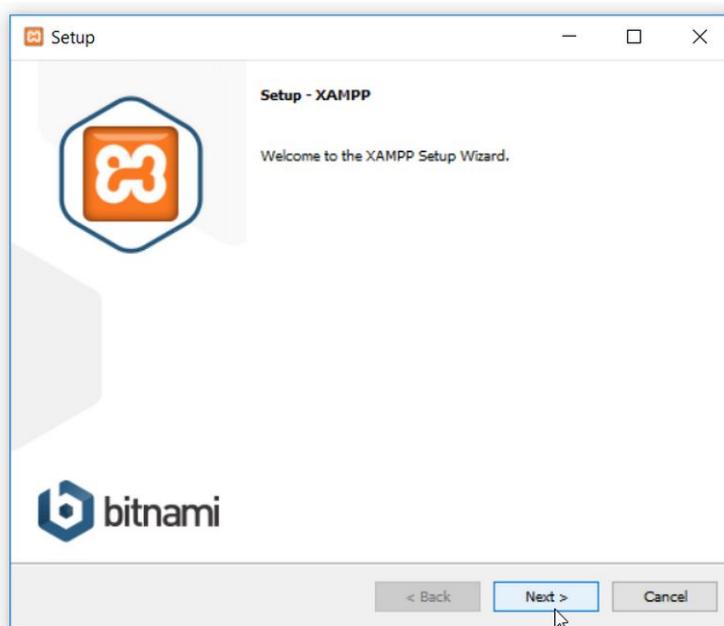
Jendela peringatan berikutnya adalah tentang UAC (User Account Control):



Gambar: Peringatan UAC jika XAMPP di-instal di C:\Program Files)

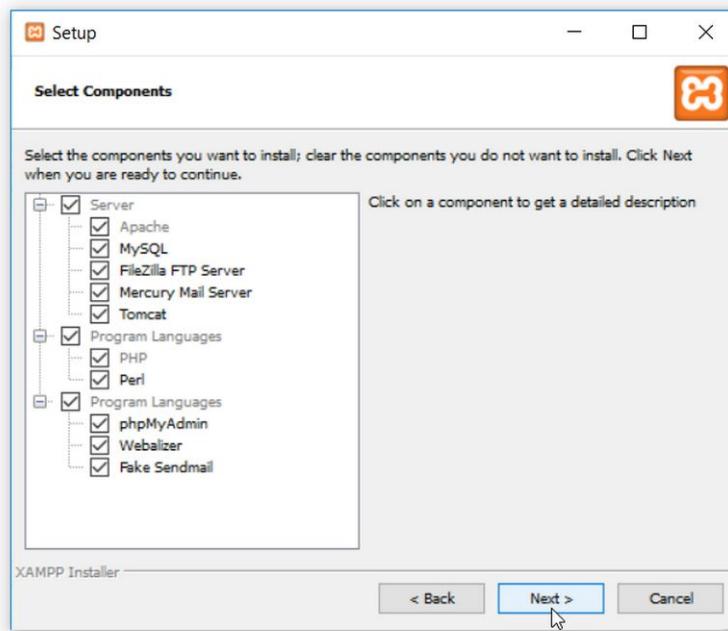
Peringatan ini berkaitan dengan proteksi Windows. Jika XAMPP diinstall di folder C:\Program Files (x86), akan terjadi pembatasan hak akses yang bisa menyebabkan XAMPP berjalan tidak normal. Secara default, XAMPP 7.1.7 akan diinstall ke C:\xampp, jadi hal ini seharusnya tidak jadi masalah. Kecuali anda mengubahnya ke folder C:\Program Files (x86) yang tidak saya sarankan.

Klik tombol OK untuk melanjutkan.



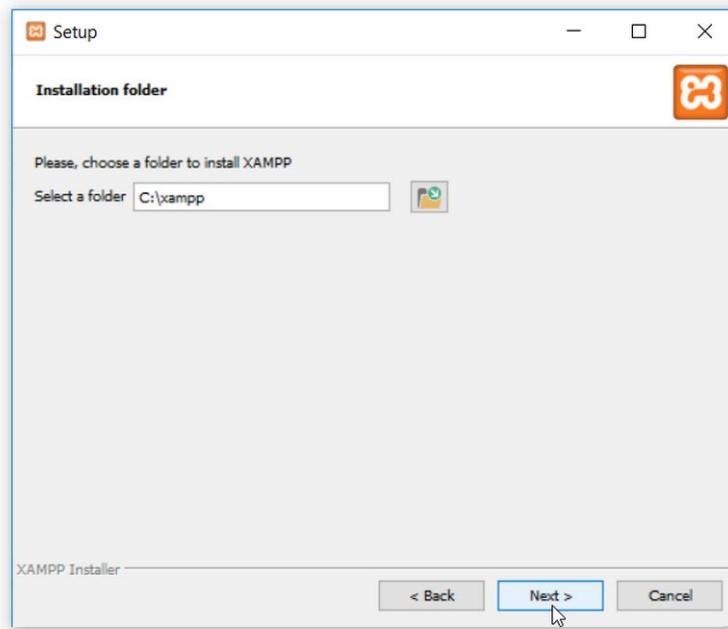
Gambar: Jendela awal instalasi XAMPP

Jendela awal instalasi akan muncul, klik saja tombol Next.



Gambar: Jendela Select Component

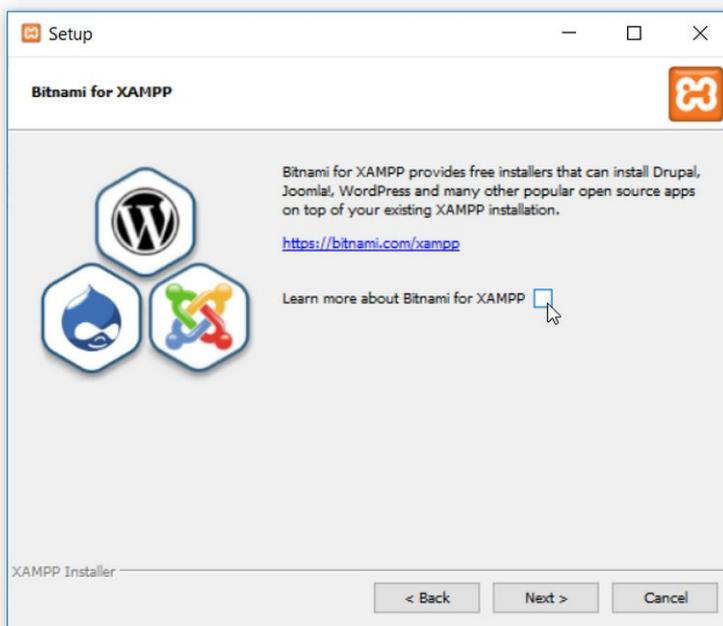
Jendela berikutnya adalah “Select Component”. Pada bagian ini kita bisa memilih aplikasi apa saja yang akan diinstall. Dalam tahap ini saya membiarkan semua pilihan. Klik tombol Next untuk melanjutkan.



Gambar: Jendela Installation Folder

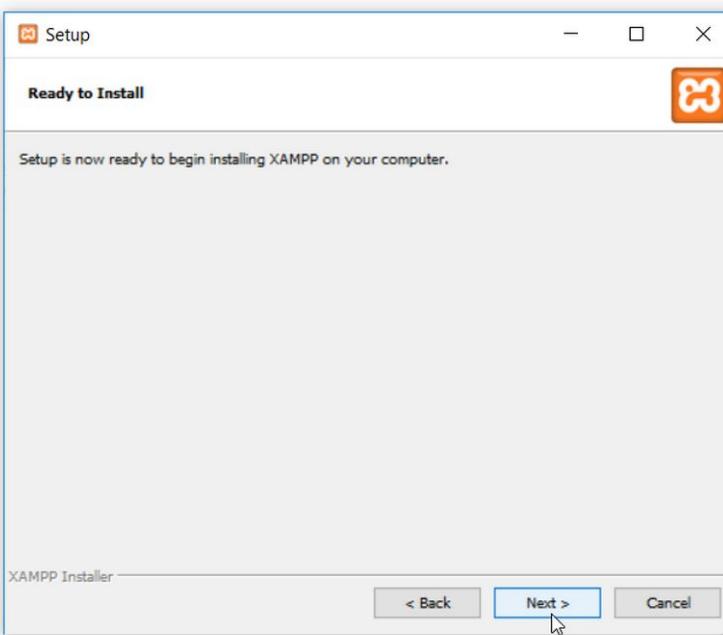
Pada jendela “Installation Folder” kita bisa menentukan folder tempat file XAMPP akan diinstall. Secara default, lokasinya di C:\xampp, anda bebas jika ingin mengubahnya.

Sebagai contoh, jika anda ingin menginstall beberapa aplikasi XAMPP, bisa memisahkannya di folder yang berbeda, seperti C:\xampp_7 untuk XAMPP 7 dan C:\xampp_5 untuk XAMPP 5.6



Gambar: Jendela Bitnami for XAMPP

Tampilan berikutnya adalah jendela “Bitnami for XAMPP”. Bitnami adalah aplikasi AMP stack yang juga menjadi sponsor XAMPP. Saat ini kita tidak memerlukannya. Hapus pilihan “*learn more about Bitnami for XAMPP*”, kemudian klik tombol Next.

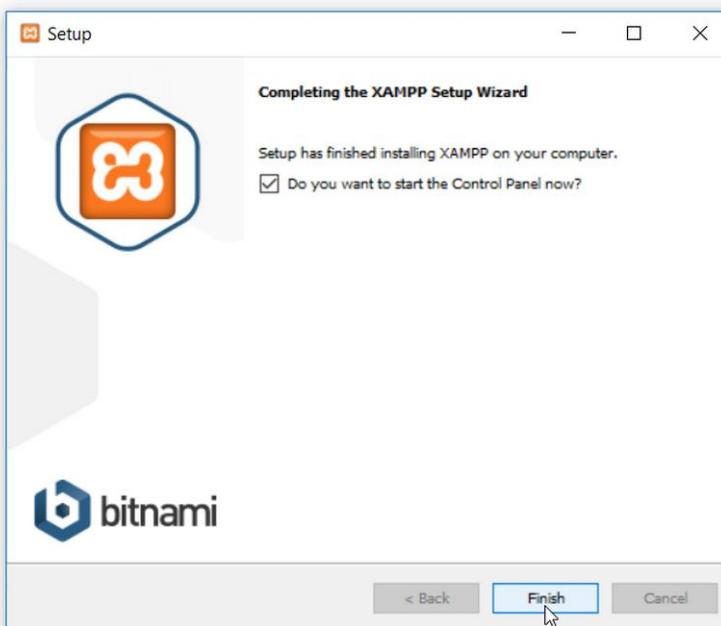


Gambar: Jendela Ready to Install

Jendela “Ready to Install” berisi konfirmasi untuk mulai menginstall XAMPP. Langsung saja klik tombol Next dan XAMPP akan memulai proses instalasi.



Gambar: XAMPP sedang dalam proses instalasi



Gambar: XAMPP telah selesai diinstall

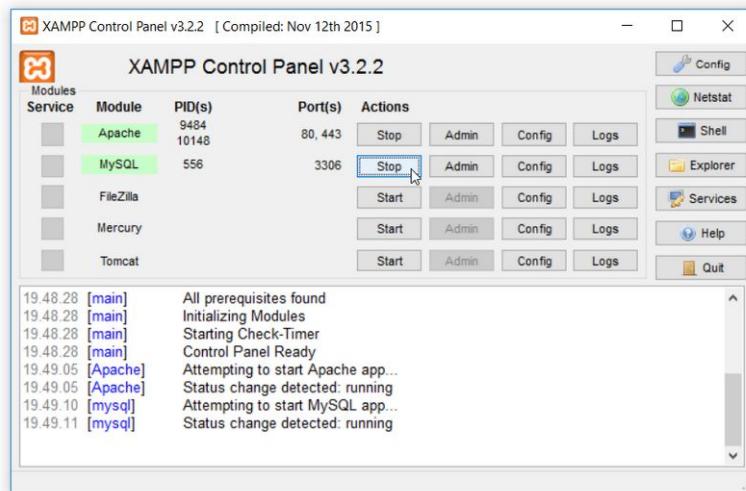
Jika jendela “Completing the XAMPP Setup Wizard” telah tampil, berarti proses instalasi XAMPP sudah selesai. Pada bagian ini kita akan langsung mencoba aplikasi XAMPP, sehingga biarkan pilihan check list “Do you want to start the Control Panel now?”, kemudian klik tombol **Finish**.

Jika anda membiarkan pilihan “Do you want to start the Control Panel now?” pada jendela terakhir proses instalasi XAMPP, akan tampil jendela XAMPP Control Panel.

Sesuai dengan namanya, jendela XAMPP Control Panel adalah jendela yang digunakan

untuk mengontrol apa saja modul XAMPP yang akan atau sedang berjalan. Jika jendela ini tidak tampil, anda bisa mengaksesnya dari menu START->All Programs->XAMPP->XAMPP Control Panel.

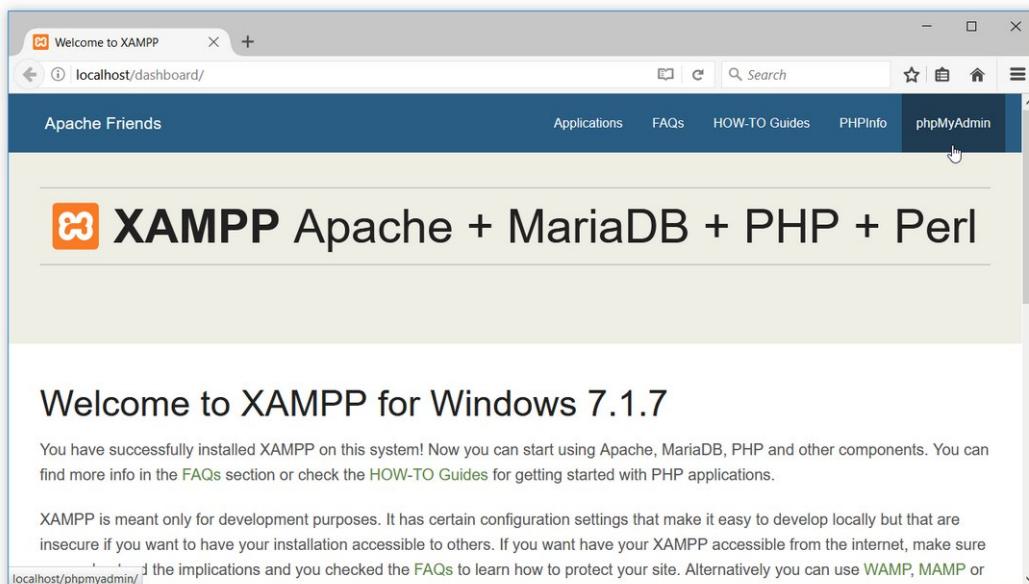
Untuk menguji instalasi XAMPP, silahkan klik tombol START pada modul Apache dan MySQL. Jika tidak ada masalah, akan tampil warna hijau pada kolom **module**, seperti gambar berikut:



Gambar: Cara menjalankan Apache web server dan MySQL Server

Tombol START akan berubah menjadi tombol STOP. Tombol stop digunakan untuk mematikan Apache web server dan MySQL database server.

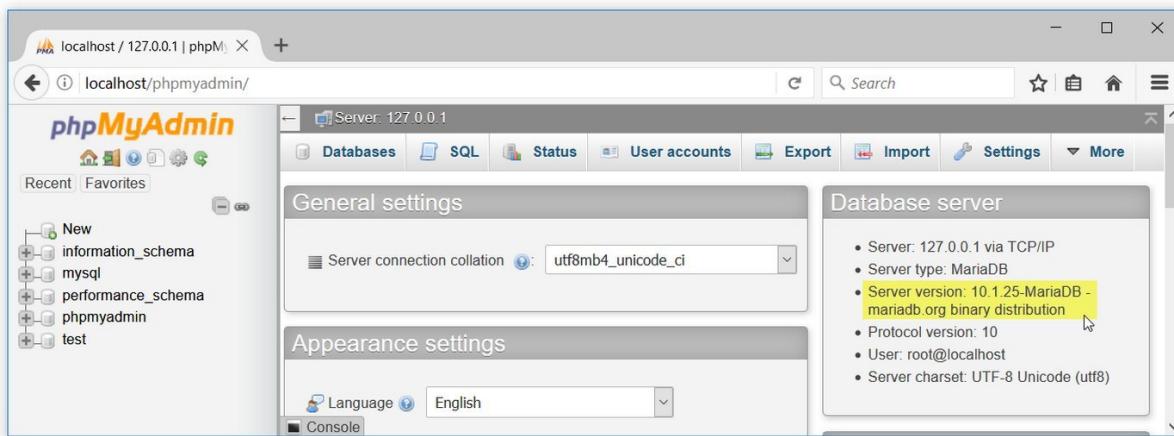
Untuk mengujinya, silahkan buka web browser dan ketik alamat <http://localhost> di address bar, kemudian tekan enter. Jika yang tampil adalah jendela pembuka XAMPP, maka semuanya telah terinstall dengan baik.



Gambar: Halaman awal localhost XAMPP

Untuk menguji apakah database MySQL (atau tepatnya MariaDB) sudah terkoneksi dan bisa di-

akses, klik menu phpMyAdmin di sudut kanan atas, atau ketik alamat <http://localhost/phpmyadmin>. Jika tampil jendela berikut, berarti database MariaDB sudah terhubung dengan Apache Web Server serta dengan PHP:



Gambar: Halaman phpMyAdmin XAMPP

Proses instalasi XAMPP sudah selesai.

4.2 Instalasi MySQL

Bagian ini khusus bagi anda yang berniat menggunakan MySQL bukan untuk web programming, atau ingin menginstall MySQL secara terpisah (tidak menggunakan versi MariaDB dari paket XAMPP).

Salah satu keuntungan dari cara ini adalah, kita bisa menginstall MySQL versi yang paling update. Jika menggunakan XAMPP, maka harus menunggu sampai XAMPP meng-update paket instalasinya.

Sebagai contoh, pada saat buku ini ditulis, versi terakhir dari MySQL adalah 5.7.18, dan untuk MariaDB sudah tersedia versi 10.2.5. Di paket XAMPP, yang tersedia baru MariaDB 10.1.25.

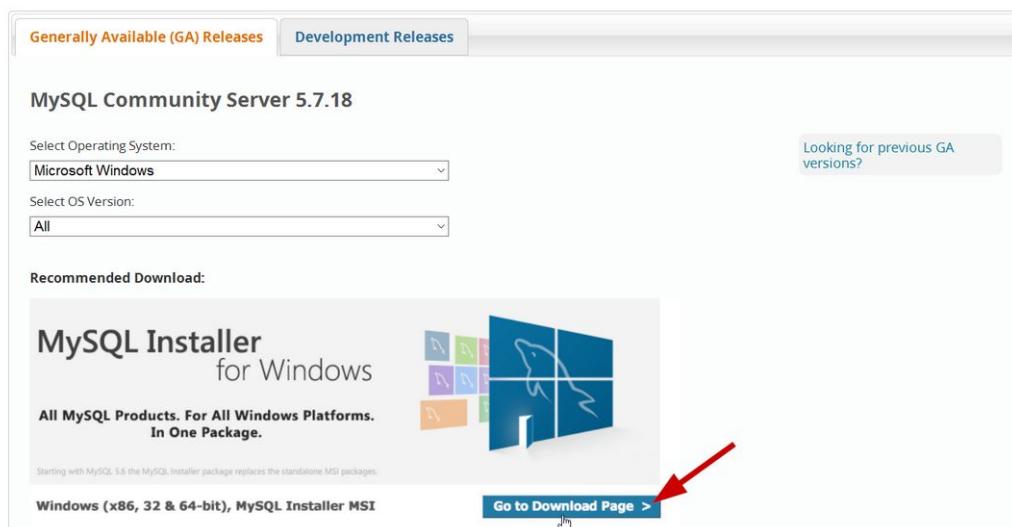
Di balik keunggulannya, menginstall MySQL secara terpisah butuh file instalasi yang cukup besar. **MySQL 5.7.18** yang akan saya download berukuran sekitar **415 MB**.

Jika dibandingkan dengan XAMPP, hal ini terasa agak aneh. File instalasi XAMPP 7.1.7 hanya berukuran 123 MB. Di dalamnya tidak hanya tersedia MariaDB, tapi juga berbagai aplikasi lain seperti Apache, PHP, Perl, dll. Kenapa file instalasi MySQL bisa sebesar ini?

Alasannya karena XAMPP sudah “menyurat” beberapa fitur yang tidak terlalu dibutuhkan. Sedangkan jika kita mendownload file MySQL langsung dari situs resminya, di dalam paket instalasi juga terdapat berbagai fitur tambahan, seperti *MySQL Workbench*, *MySQL Documentation*, *MySQL Connectors*, dll.

Baik, jika anda ingin menginstall MySQL secara terpisah, silahkan download file installer melalui situs resminya di dev.mysql.com/downloads/mysql/⁵. Scroll halaman hingga bagian bawah, lalu klik gambar “MySQL Installer for Windows”.

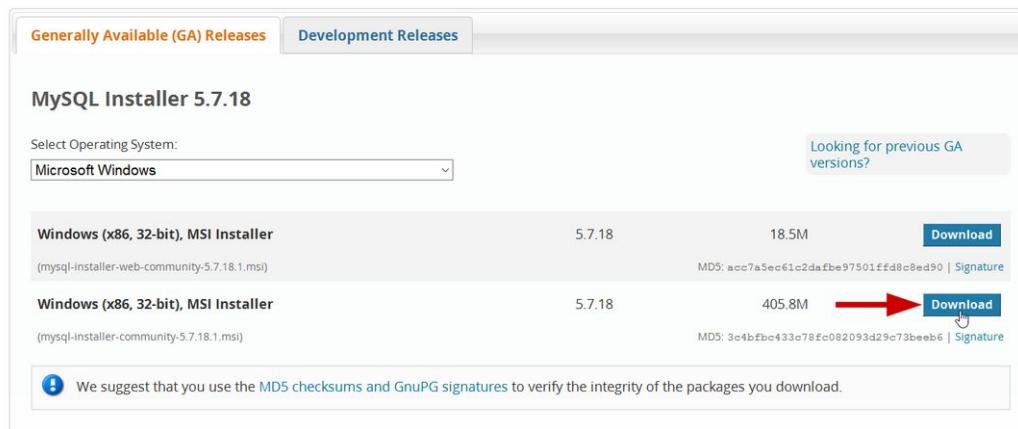
⁵ <https://dev.mysql.com/downloads/mysql/>



Gambar: Klik gambar untuk download file instalasi

Dibawah gambar juga terdapat beberapa link, tapi itu adalah installer dalam bentuk zip. Yang kita butuhkan adalah file berbentuk **msi**, jadi abaikan link tersebut.

Please report any bugs or inconsistencies you observe to our [Bugs Database](#).
Thank you for your support!



Gambar: Pilih Windows (x86, 32-bit), MSI Installer

Kembali, scroll halaman hingga paling ke bawah. Disini terdapat 2 pilihan link yang sama-sama bernama “Windows (x86, 32-bit), MSI Installer”. Bedanya, satu file berukuran 18.5M, sedangkan yang satu lagi berukuran 405.8M.

File dengan ukuran kecil adalah *online installer*, dimana proses download akan berjalan pada saat instalasi nanti. Jika anda seperti saya (koneksi internet naik-turun), sebaiknya pilih installer dengan ukuran yang besar (405.8 MB). Ini adalah *offline installer*, yakni kita akan mendownload seluruh file instalasi di awal. Selain itu kita juga bisa menyimpan master file MySQL jika nanti ingin menginstall ulang. Silahkan klik link “Download” .

Begin Your Download

mysql-installer-community-5.7.18.1.msi

Login Now or Sign Up for a free account.

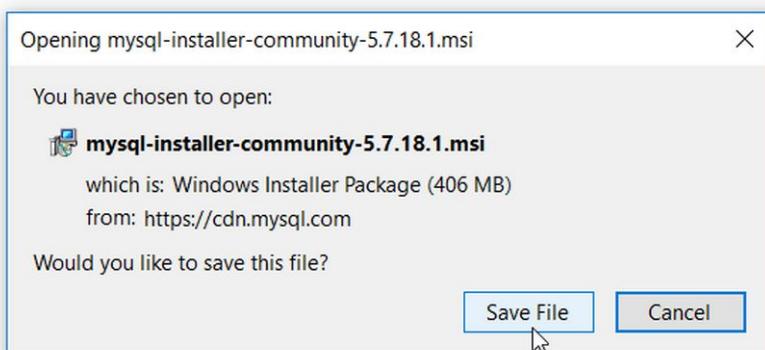
An Oracle Web Account provides you with the following advantages:

- Fast access to MySQL software downloads
- Download technical White Papers and Presentations
- Post messages in the MySQL Discussion Forums
- Report and track bugs in the MySQL bug system
- Comment in the MySQL Documentation



Gambar: Klik link “No thanks, just start my download”

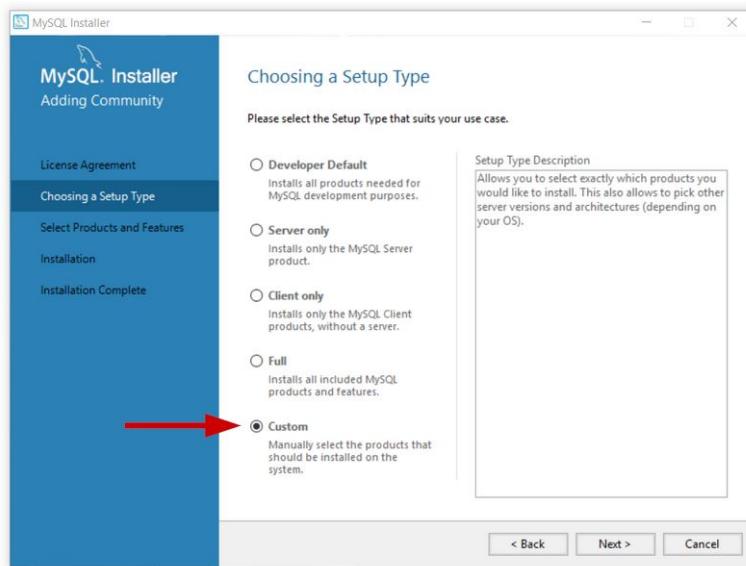
Di halaman selanjutnya kita disarankan untuk membuat akun di situs MySQL. Anda bisa mengikutinya atau langsung saja klik link “No thanks, just start my download”. Proses download akan berjalan.



Gambar: Proses download file installer MySQL

Tergantung koneksi, proses download ini bisa memakan waktu yang cukup lama. Terlebih ukuran file MySQL terhitung “jumbo”, hampir setengah gigabyte. File yang saya download bernama: **mysql-installer-community-5.7.18.1.msi**.

Setelah file MySQL sukses di download, klik dua kali untuk memulai proses instalasi. Jendela pertama terdapat “License Agreement”. Langsung saja checklist pilihan “*I accept the license terms*”, lalu klik tombol Next.

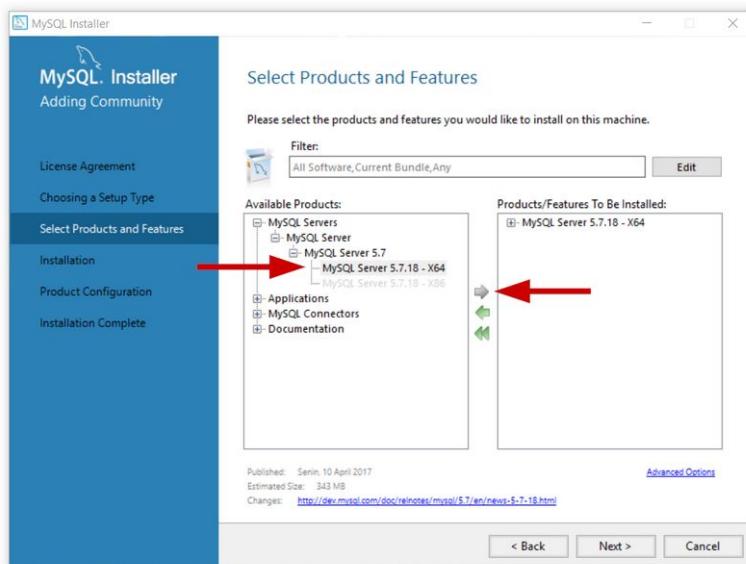


Gambar: Jendela “Choosing a Setup Type”

Pada jendela “Choosing a Setup Type”, terdapat beberapa pilihan tipe instalasi. Mulai dari untuk *developer default*, *server only*, *client only*, *full* hingga *custom*.

Karena kita akan menggunakan MySQL dalam tahap pengembangan program, mungkin pilihan “Developer Default” adalah yang paling pas. Namun opsi ini menginstall cukup banyak aplikasi, bisa sampai 10 komponen serta memakan ruang harddisk lebih dari 1GB. Yang kita perlukan saat ini hanyalah **MySQL Server**.

Akan tetapi, jika memilih **Server Only** kita tidak bisa mengubah folder instalasi MySQL. Oleh karena itu pilih **Custom** kemudian klik tombol **Next**.



Gambar: Memilih komponen MySQL Server

Pada jendela “Select Products and Features” kita bisa memilih aplikasi dan fitur apa saja yang ingin diinstall. Dapat dilihat bahwa file instalasi MySQL membundel banyak komponen selain

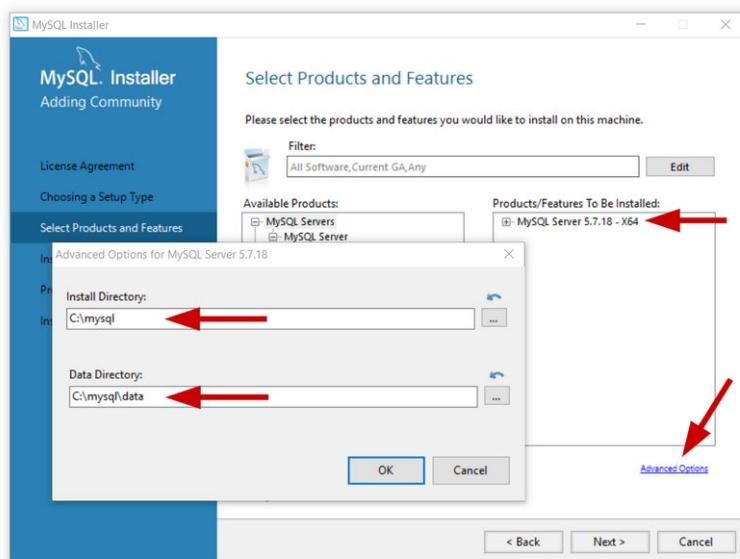
MySQL Server, seperti: Aplikasi admin untuk excel, visual studio, connector MySQL, dan juga file dokumentasi. Inilah yang membuat file installer MySQL begitu besar.

Silahkan klik tombol tanda tambah (+) paling atas (MySQL Server), sampai anda menemukan 2 pilihan: **MySQL Server 5.7.18-x64** dan **MySQL Server 5.7.18-x86**. Kita hanya butuh memilih salah satu saja dari pilihan ini.

Karena saya menggunakan Windows 10 64-bit, maka akan memilih **MySQL Server 5.7.18-x64**. Kemudian klik tombol panah hijau ditengah-tengah jendela untuk memindahkan dari jendela “Available Product” ke “Product/Features To Be Installed”.

Jika anda langsung klik tombol **Next**, maka program MySQL akan diinstall pada folder: C:\Program Files\MySQL\MySQL Server 5.7, dan folder data di C:\ProgramData\MySQL\MySQL Server 5.7. Meskipun ini tidak masalah, tapi saya ingin mengubahnya ke folder lain supaya mudah diakses.

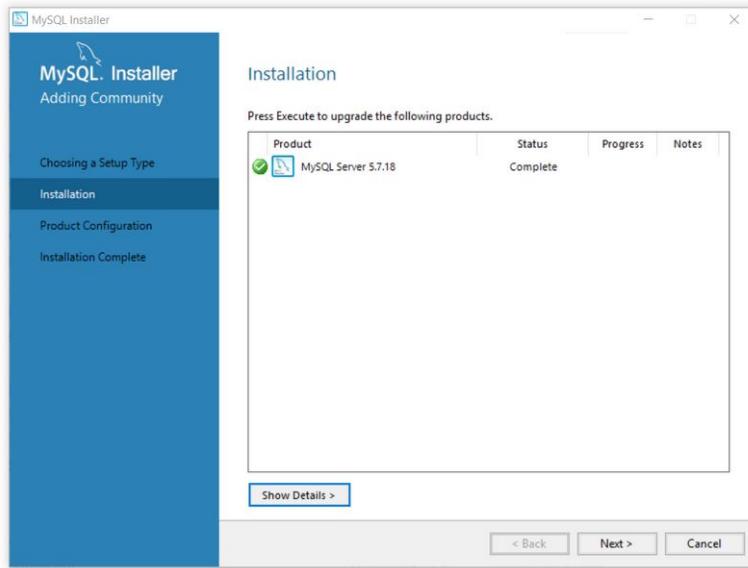
Caranya, klik pilihan **MySQL Server 5.7.18-x64** pada jendela sebelah kanan, kemudian di pojok kanan bawah akan muncul pilihan “Advanced Option”. Klik pilihan ini, dan akan keluar jendela baru untuk mengubah lokasi instalasi MySQL.



Gambar: Mengubah folder instalasi MySQL

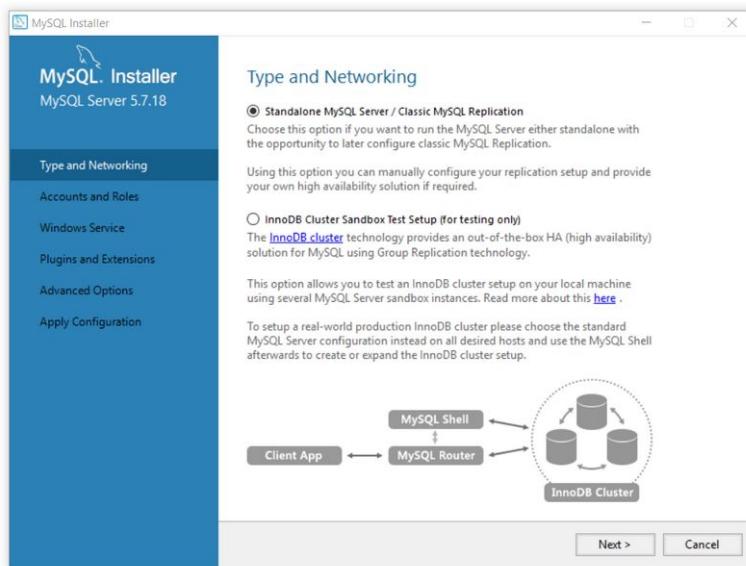
Terdapat 2 isian: **Install Directory** yang akan menjadi folder tempat file program MySQL. Dan **Data Directory** dimana file data seperti database dan tabel berada. Saya akan mengubah *Install Directory* ke C:\mysql dan *Data Directory* ke C:\mysql\data. Kedua folder ini tidak harus dibuat dahulu karena MySQL akan membuatnya secara otomatis. Akhiri dengan men-klik tombol **OK**, kemudian klik tombol **Next**.

Di jendela yang tampil klik tombol **Execute**. Proses instalasi akan berlangsung beberapa saat, kemudian klik tombol **Next** jika sudah selesai.



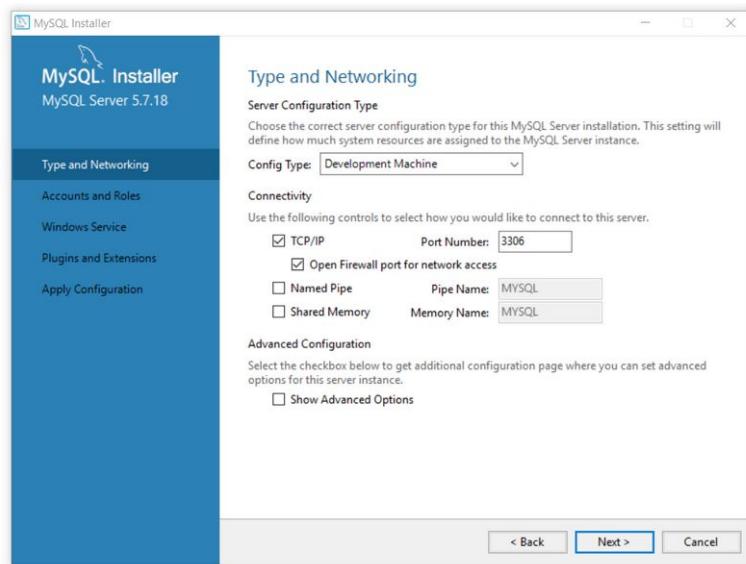
Gambar: MySQL Server berhasil diinstall

Jendela berikutnya adalah untuk konfigurasi MySQL Server. Klik tombol Next.



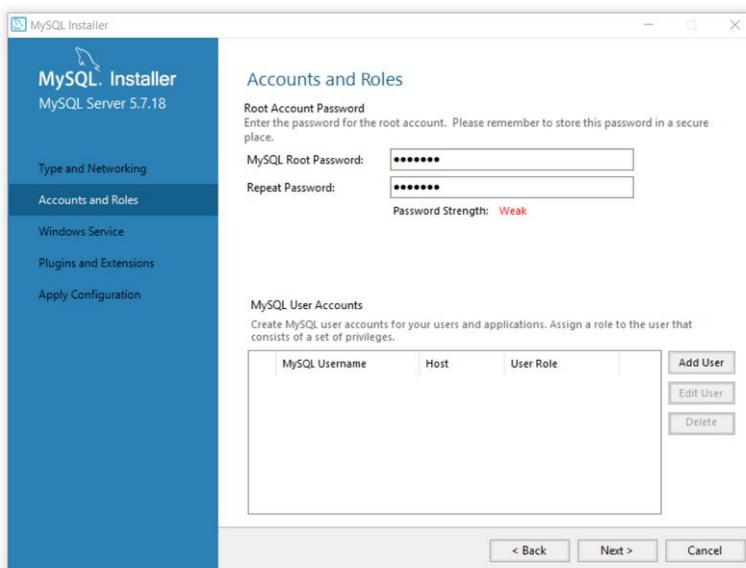
Gambar: Jendela konfigurasi “Type and Networking”

Di jendela konfigurasi “Type and Networking” pilih “*Standalone MySQL Server / Classic MySQL Replication*”, karena kita hanya akan menjalankan MySQL di satu komputer saja. Klik tombol Next.



Gambar: Jendela konfigurasi “Type and Networking” untuk port

Jendela berikutnya mengatur *config type* dan *port*. Biarkan settingan default **Development Machine** dan port MySQL di 3306. Klik tombol **Next**.

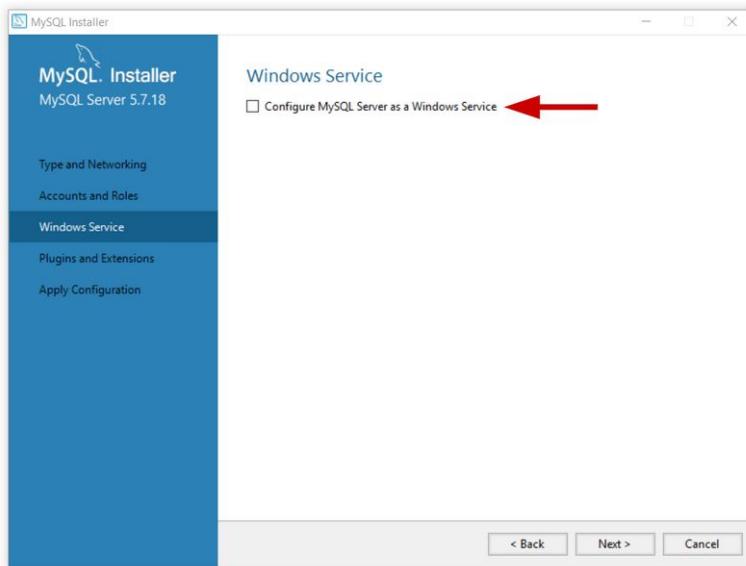


Gambar: Jendela konfigurasi “Accounts and Role”

Pada jendela konfigurasi “Accounts and Role” kita akan memberikan password untuk user **root**. **Root** adalah user tertinggi di dalam MySQL atau sering disebut juga dengan **superuser**. User root memiliki hak akses kepada semua tabel dan database, termasuk menghapus seluruh database yang ada.

Karena kita menggunakan MySQL server ini di komputer sendiri dan hanya untuk proses belajar, silahkan input password yang mudah diingat, saya akan menggunakan password: “**rahasia**”.

Di bagian bawah juga terdapat tombol untuk menambahkan user baru. Tapi kita tidak akan menggunakan karnya karena user MySQL juga bisa ditambahkan kemudian. Klik tombol **Next**.

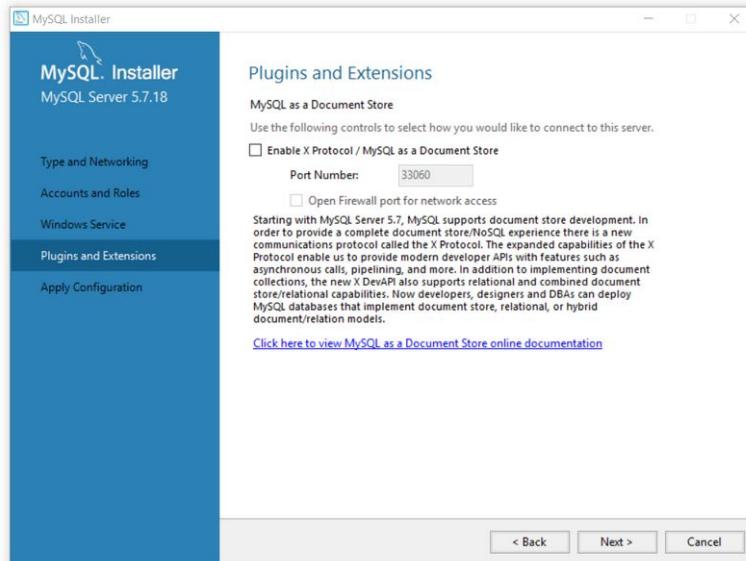


Gambar: Hapus pilihan *Configure MySQL Server as Windows Service*

Jendela berikutnya **sangat penting**, terutama jika anda berencana menginstall lebih dari 1 MySQL di dalam komputer (misalnya satu lagi dari XAMPP).

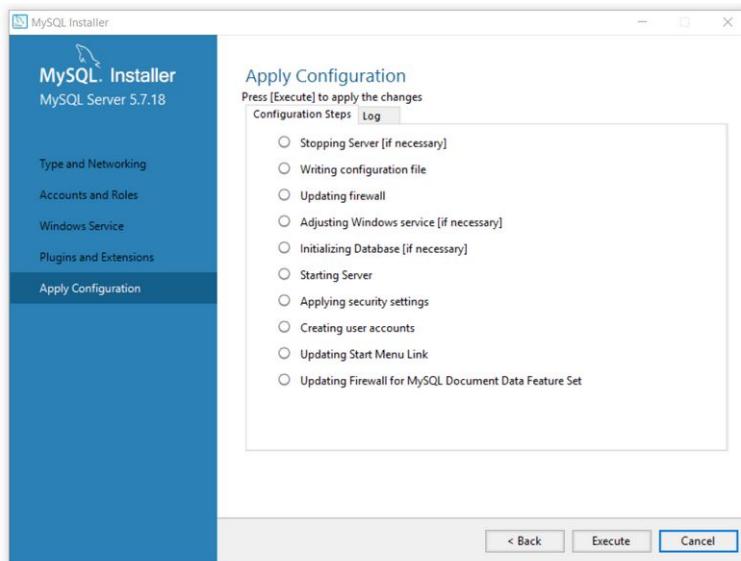
Jika checkbox “*Configure MySQL Server as Windows Service*” dipilih, MySQL akan diinstall sebagai “**Windows Service**”. Artinya, MySQL akan langsung aktif setiap Windows berjalan.

Ini bisa menjadi masalah jika anda ingin menjalankan MySQL dari sumber lain (seperti XAMPP). Karena MySQL tidak bisa berjalan bersamaan di satu komputer (jika menggunakan port yang sama). Oleh karena itu hapus pilihan “*Configure MySQL Server as Windows Service*”, lalu klik **Next**.



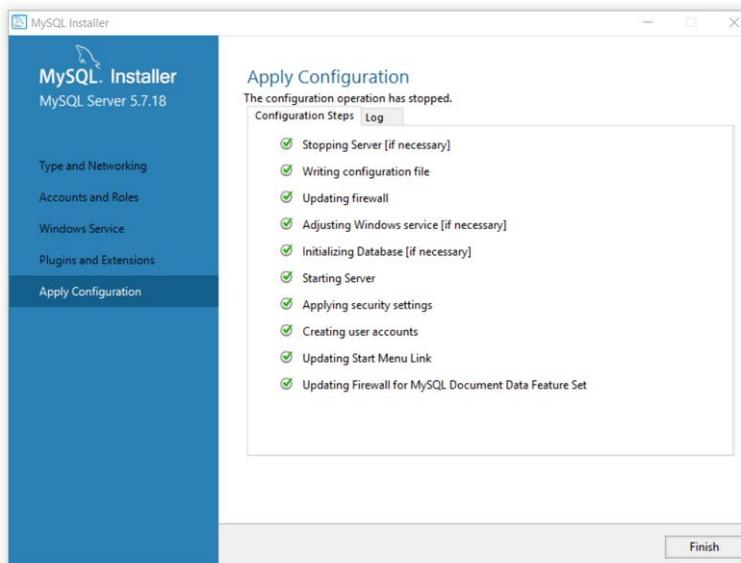
Gambar: Jendela *Plugins and Extensions*

Di halaman jendela konfigurasi “**Plugins and Extensions**” biarkan pilihan default (tidak dicentang) lalu klik **Next**.



Gambar: Jendela *Apply Configuration*

Jendela terakhir adalah “**Apply Configuration**”. Disini seluruh settingan yang sudah kita pilih akan diterapkan. Klik tombol **Execute**.



Gambar: Konfigurasi sudah selesai diterapkan

Setelah beberapa saat, seluruh konfigurasi sudah diterapkan. Klik tombol **Finish**, **Next**, dan **Finish**. Proses instalasi MySQL Server sudah selesai.

4.3 Instalasi MariaDB

Sebagai alternatif dari MySQL, anda juga bisa menginstall MariaDB. Untuk mendownload file installer MariaDB, bisa langsung ke downloads.mariadb.org⁶.

⁶<https://downloads.mariadb.org>

Di halaman ini terdapat beberapa versi MariaDB. Versi pertama adalah versi **stable release**. Ini merupakan versi yang dianggap paling stabil dan layak dijadikan server untuk web sebenarnya. Pada saat buku ini saya tulis, versi stabil terakhir dari MariaDB adalah 10.1.23.

Terdapat juga versi yang lebih baru dari 10.1, yakni versi 10.2 dan 10.3. Namun versi ini masih versi **development release**. Versi development release artinya masih dalam pengembangan dan belum bisa dikatakan stabil (mungkin ada beberapa bug/error).

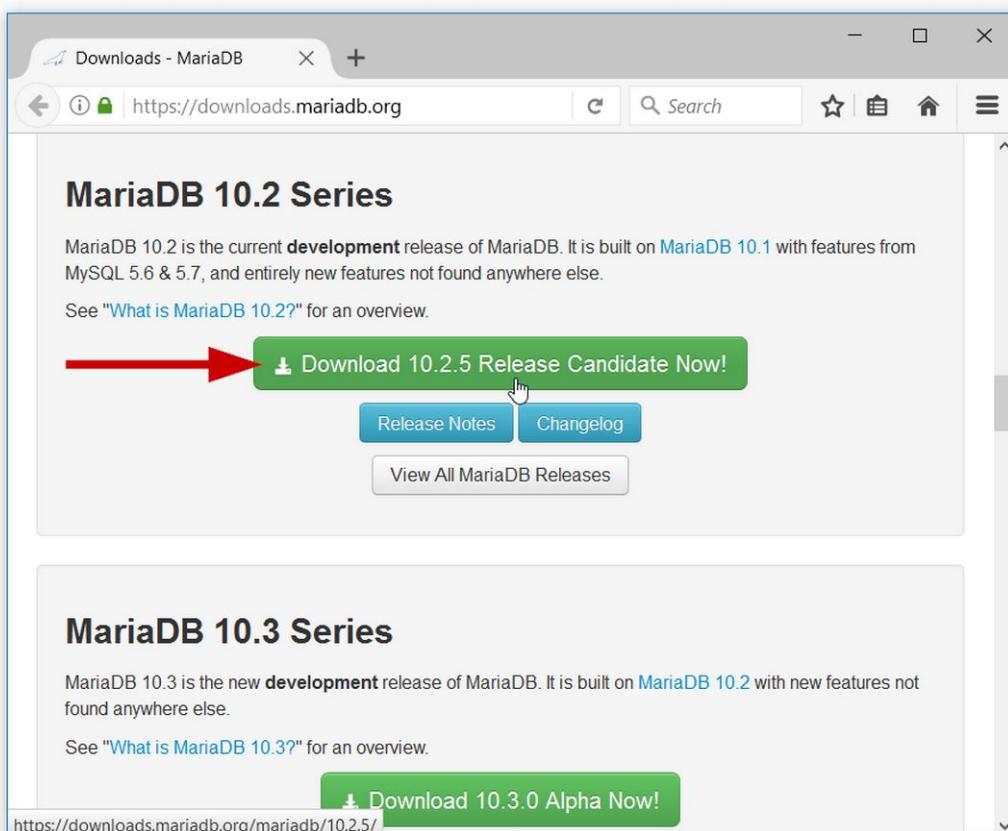
Development release ini juga memiliki beberapa tahap: **alpha**, **beta**, dan **release candidate**. Ini semua termasuk ke dalam siklus pengembangan sebuah software.

Versi **alpha** adalah tahap awal pengembangan, dimana potensi bug/error masih banyak. Fitur-fitur yang ada juga bisa dihapus atau ditambah tergantung kebutuhan.

Setelah mayoritas error di versi alpha di perbaiki, akan masuk ke tahap **beta**. Di tahap beta aplikasi kembali di uji dan memperbaiki error (jika masih ditemukan).

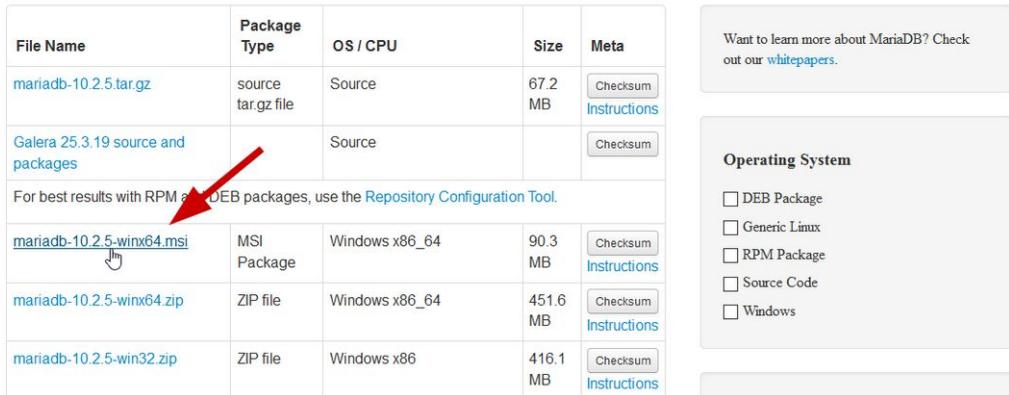
Tahap terakhir dalam development release adalah **release candidate**. Di tahap ini aplikasi dianggap sudah cukup “bersih” dan stabil, tapi masih butuh beberapa ujicoba. Setelah benar-benar yakin, baru kemudian naik tingkat ke **stable release**.

Untuk kali ini saya akan mendownload MariaDB versi 10.2.5 (termasuk kategori *release candidate*). Karena MariaDB akan selalu diupdate, bisa jadi versi yang anda dapatkan akan lebih baru dari ini.



Gambar: Download aplikasi MariaDB

Di halaman berikutnya terdapat link untuk mendownload berbagai versi MariaDB. Sesuaikan dengan sistem operasi yang anda gunakan. Kali ini saya akan mendownload **mariadb-10.2.5-winx64.msi**, yakni MariaDB untuk sistem Operasi Windows 64-bit (berukuran sekitar 86MB). Klik link tersebut:



File Name	Package Type	OS / CPU	Size	Meta
mariadb-10.2.5.tar.gz	source tar.gz file	Source	67.2 MB	Checksum Instructions
Galera 25.3.19 source and packages		Source		Checksum
For best results with RPM and DEB packages, use the Repository Configuration Tool .				
mariadb-10.2.5-winx64.msi	MSI Package	Windows x86_64	90.3 MB	Checksum Instructions
mariadb-10.2.5-winx64.zip	ZIP file	Windows x86_64	451.6 MB	Checksum Instructions
mariadb-10.2.5-win32.zip	ZIP file	Windows x86	416.1 MB	Checksum Instructions

Want to learn more about MariaDB? Check out our [whitepapers](#).

Operating System

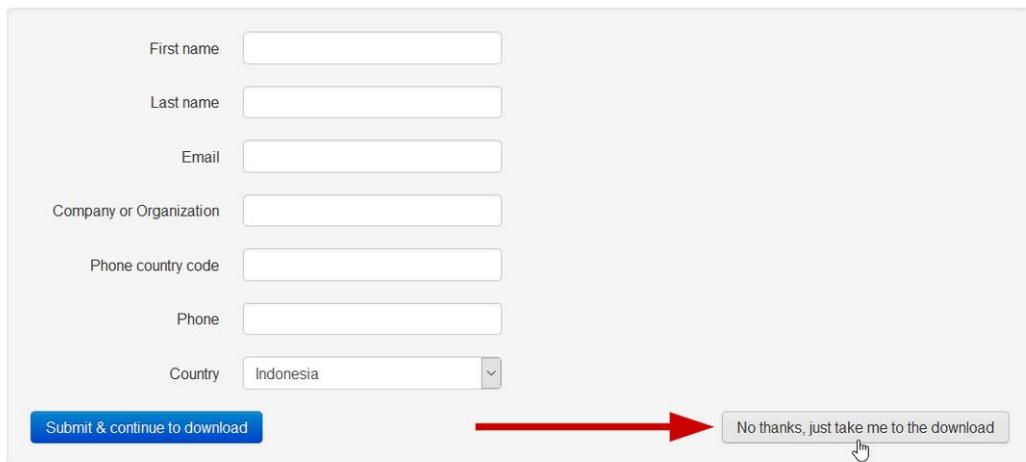
- DEB Package
- Generic Linux
- RPM Package
- Source Code
- Windows

Gambar: Download aplikasi MariaDB untuk Windows 64-bit

Halaman berikutnya kita diminta untuk mengisi form identitas diri. Data ini akan digunakan oleh MariaDB untuk mengetahui siapa saja yang mendownload aplikasi MariaDB. Anda boleh mengisinya atau langsung klik link tombol “*No thanks, just take me to the download*”.

MariaDB

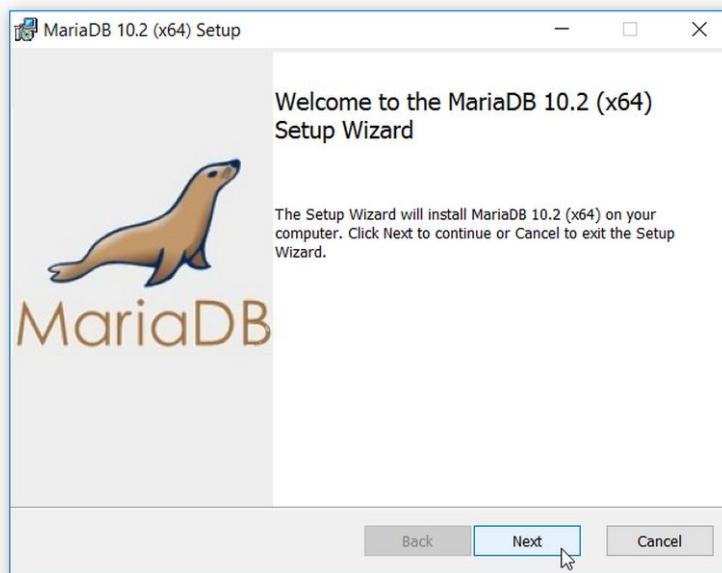
Thank you for your interest in downloading MariaDB. Please take a moment to tell us a little about yourself. Or you can go directly to the [download](#).



The form contains fields for First name, Last name, Email, Company or Organization, Phone country code, Phone, and Country (set to Indonesia). At the bottom, there are two buttons: "Submit & continue to download" and "No thanks, just take me to the download". A red arrow points to the second button.

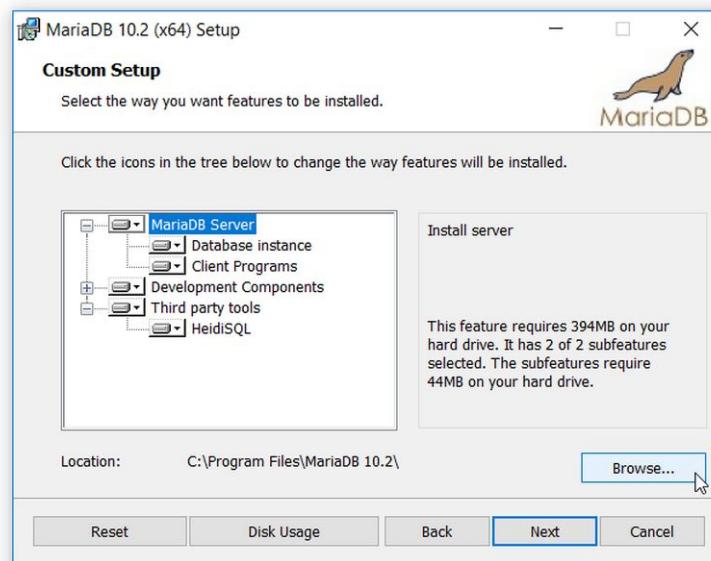
Gambar: Klik tombol “No thanks, just take me to the download”

Proses download akan berlangsung beberapa saat. Setelah berhasil di download, langsung double klik file installer tersebut:



Gambar: Jendela awal proses instalasi MariaDB

Klik tombol **Next** untuk melanjutkan. Di halaman “License Agreement”, klik checkbox “*I accept the terms in the License Agreement*”, lalu klik tombol **Next**.

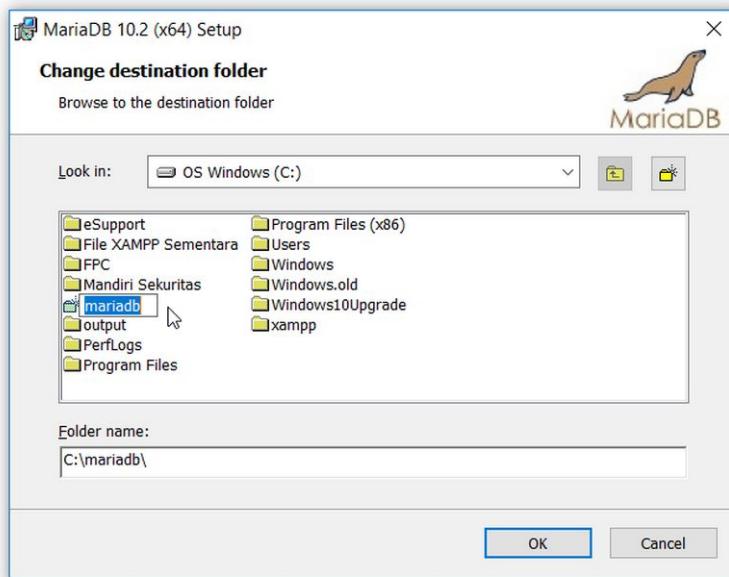


Gambar: Jendela Custom Setup MariaDB

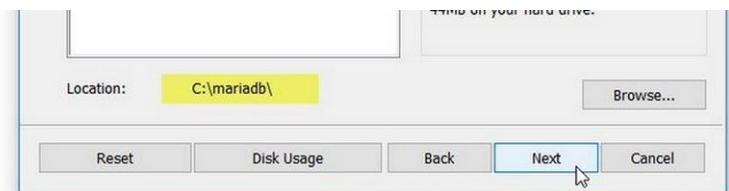
Pada jendela “Custom Setup”, kita bisa mengatur komponen apa saja yang akan diinstall. Di bagian bawah terdapat keterangan Location **C:\Program Files\MariaDB 10.2**. Artinya, hasil instalasi MariaDB akan ditempatkan di folder ini.

Supaya lebih mudah diakses, saya akan memindahkannya ke folder baru. Klik tombol “**Browse...**”, pindah ke drive C, kemudian buat folder “**mariadb**”. Dengan demikian, alamat folder instalasi MariaDB akan berada di **C:\mariadb**.

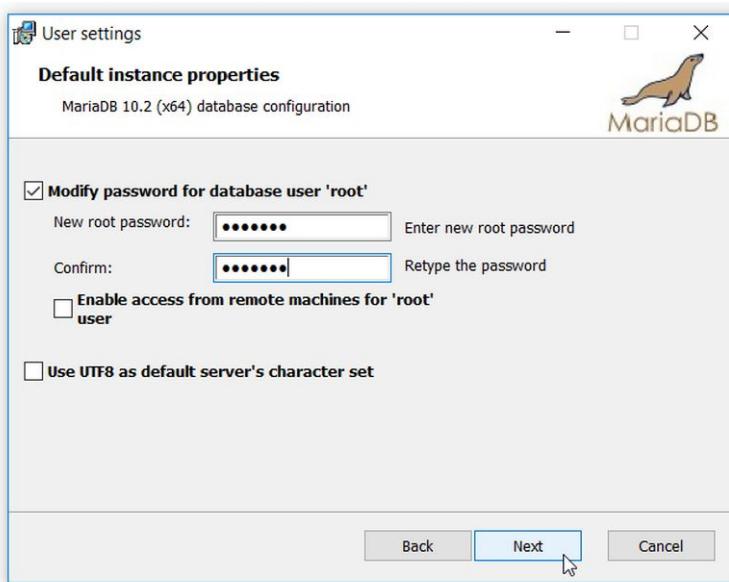
Klik tombol **OK**, kemudian klik **Next** untuk melanjutkan.



Gambar: Memindahkan folder instalasi MariaDB

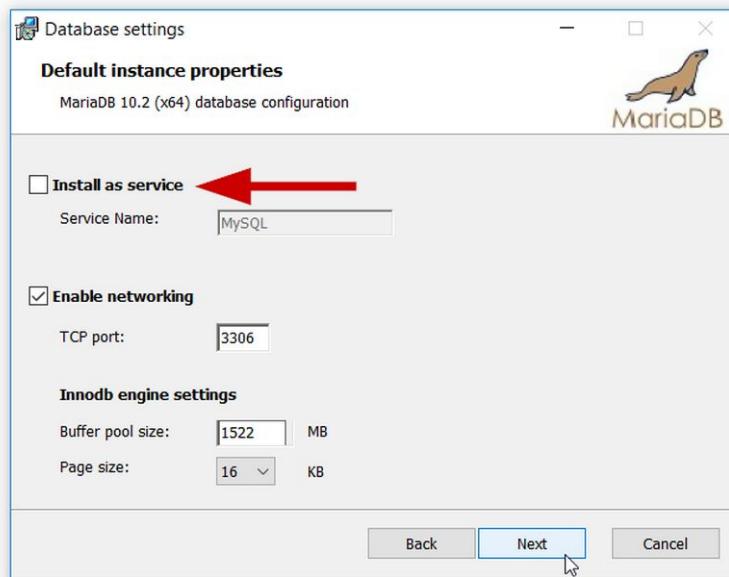


Gambar: MariaDB akan diinstall di folder C:\mariadb\



Gambar: Menginput password root MariaDB

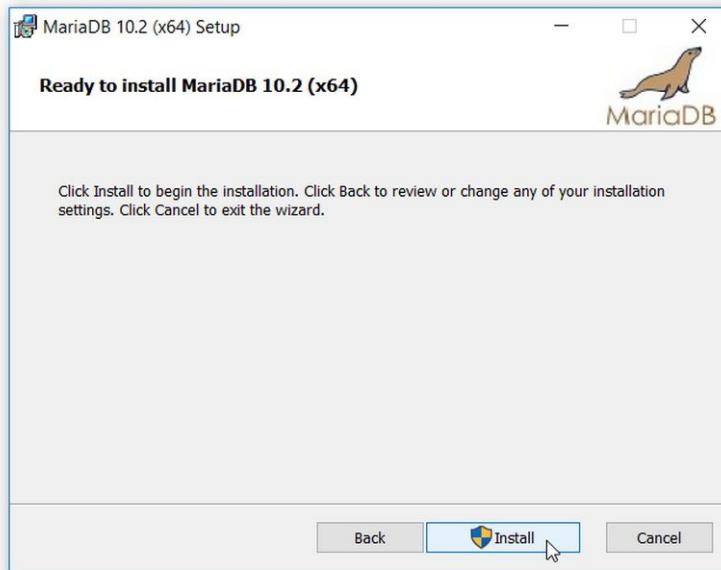
Pada jendela berikutnya, kita diminta untuk menginput password untuk user root MariaDB. Saya akan menginput password “**rahasia**”. Klik tombol Next.



Gambar: Pengaturan “Install as Service” MariaDB

Di jendela ini matikan checklist pilihan “**Install as Service**”. Jika ini dipilih, MariaDB akan diinstall sebagai “**Windows Service**”. Artinya, MariaDB akan langsung aktif saat Windows berjalan. Ini bisa menjadi masalah karena kita tidak bisa menjalankan MariaDB bawaan XAMPP (akan bentrok dengan MariaDB Service).

Pastikan menghapus checkbox “*Install as Service*”, kemudian klik tombol **Next**. Di jendela selanjutnya ada pilihan untuk mengirimkan data anonim ke MariaDB. Anda boleh berpartisipasi atau langsung saja klik tombol **Next**.



Gambar: Mulai proses instalasi MariaDB

Seluruh pengaturan untuk instalasi MariaDB sudah selesai. Klik tombol “**Install**” dan proses instalasi akan berlangsung. Klik tombol **Finish** jika sudah selesai.

Dalam bab ini kita sudah menginstall 3 aplikasi, yakni **XAMPP**, **MySQL** dan **MariaDB**. Anda tidak harus menginstall ketiga aplikasi ini. Yang saya rekomendasikan adalah XAMPP, karena proses instalasinya lebih singkat dan sudah satu paket dengan aplikasi pengembangan web server lain seperti **Apache**, **PHP**, serta **PHPmyAdmin**.

Menginstall MySQL atau MariaDB secara manual hanya cocok bagi anda yang tidak ingin menggunakan MySQL untuk keperluan web programming.

Berikut, kita akan mempelajari cara menjalankan MySQL Server dan MySQL Client.

5. MySQL Server dan MySQL Client

Setelah aplikasi XAMPP / MySQL / MariaDB berhasil diinstall, kita sudah bisa praktik. Dalam bab ini saya akan membahas apa saja isi dari folder instalasi MySQL, file pengaturan MySQL (`my.ini`), serta cara menjalankan MySQL server dan MySQL client.

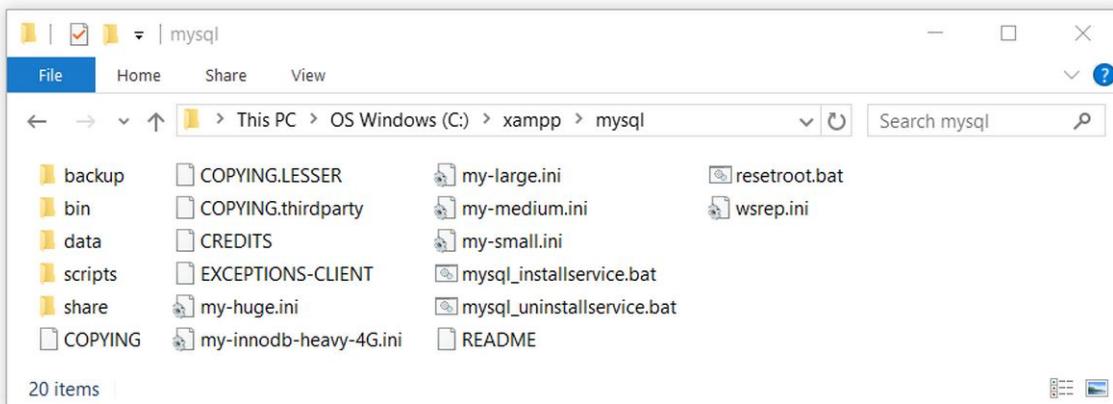
5.1 Folder Instalasi MySQL

Mari kita lihat apa saja isi dari folder instalasi MySQL / MariaDB. Jika anda mengikuti bab sebelumnya, file instalasi ini akan berada di:

- C:\xampp\mysql\ (jika menginstall XAMPP).
- C:\mysql\ (jika menginstall MySQL secara terpisah).
- C:\mariadb\ (jika menginstall MariaDB secara terpisah).

i Khusus untuk MySQL dan MariaDB yang diinstall secara terpisah (bukan bagian dari XAMPP), bisa juga berada di C:\Program Files\MySQL\MySQL Server 5.6 dan C:\Program Files\MariaDB 10.2. Ini jika anda tidak mengubah lokasi folder instalasi pada saat menginstall program tersebut.

Saya akan menggunakan contoh yang ada di C:\xampp\mysql, berikut tampilan folder tersebut:



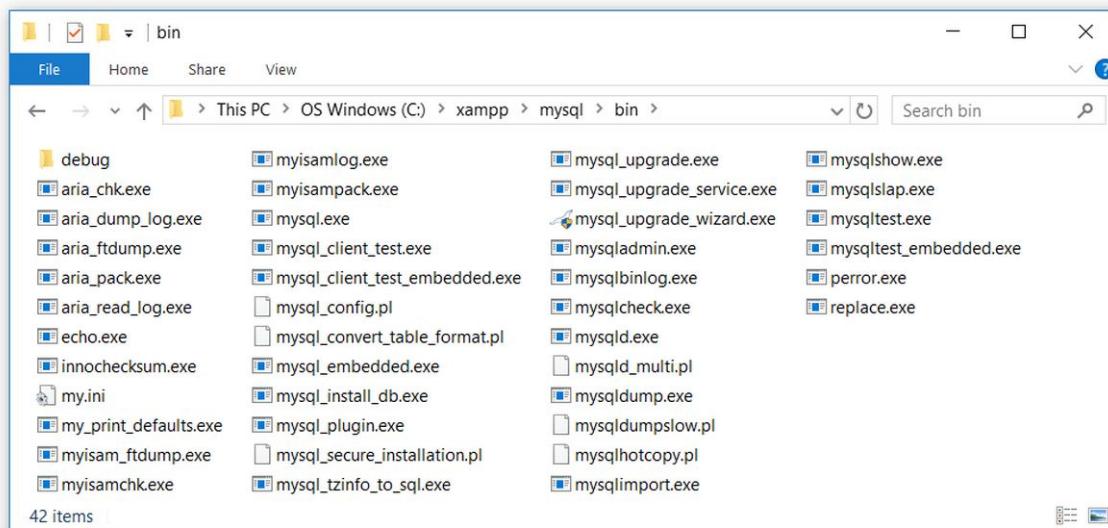
Gambar: Isi folder C:\xampp\mysql

Folder bin

Folder **bin** adalah folder utama yang berisi semua aplikasi MySQL dalam bentuk .exe. Di-antaranya:

- **mysqld.exe**: untuk menjalankan MySQL server.
- **mysql.exe**: untuk menjalankan MySQL client.
- **mysqldump.exe**: untuk export file database.
- **mysqladmin.exe**: untuk mengkonfigurasi MySQL.

Beberapa aplikasi di dalam folder bin ini akan kita pelajari secara bertahap. Salah satunya adalah **mysqld.exe** yang digunakan untuk menjalankan MySQL server, serta file **mysql.exe** untuk menjalankan MySQL client.

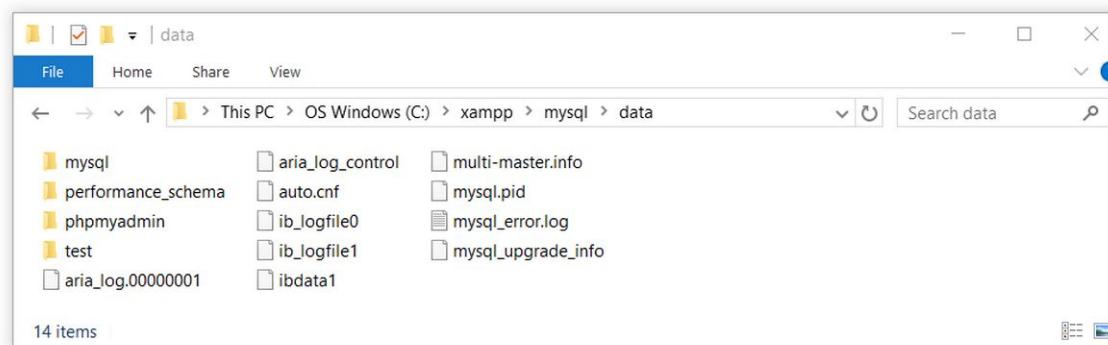


Gambar: Isi folder bin

Folder data

Folder **data** berisi seluruh file database dan tabel yang ada di dalam MySQL. Satu database akan memiliki satu folder. Di dalam MySQL bawaan XAMPP, sudah terdapat 4 folder: **mysql**, **performance_schema**, **phpmyadmin**, dan **test**. Artinya sudah ada 4 database bawaan.

Seandainya saya membuat database “**universitas**” dari dalam MySQL, akan terbentuk folder “**universitas**” di folder data.

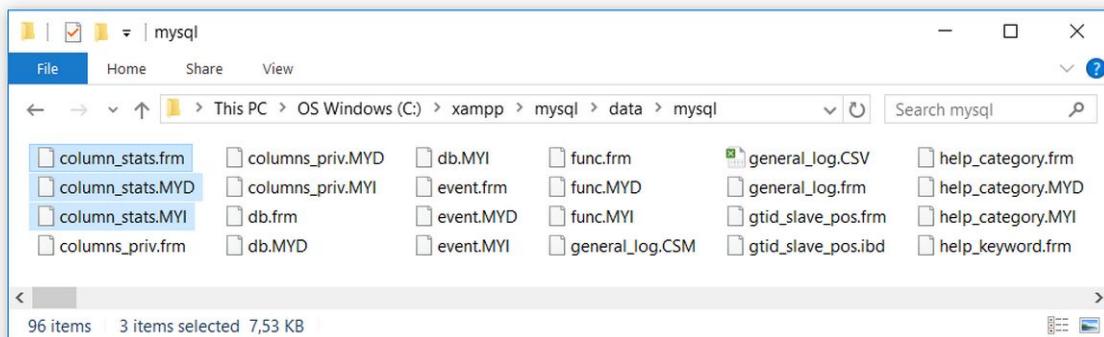


Gambar: Isi folder data

Mari kita lihat isi salah satu folder: **mysql**. Folder ini berisi cukup banyak file. File paling atas , yakni `column_stats`, adalah nama salah satu tabel dalam database **mysql**.

Setiap tabel nantinya memiliki 3 file, yaitu : `.frm`, `.MYD`, dan `.MYI` (dalam contoh kita : `column_stats.frm`, `column_stats.MYD`, dan `column_stats.MYI`). Aturan penamaan file ini bisa berbeda-beda tergantung tipe database yang digunakan (*database engine*).

Database **mysql**, adalah database khusus yang berisi berbagai settingan terkait mysql. Sebagai contoh, nama username yang berhak mengakses MySQL akan disimpan ke dalam database ini.



Gambar: Isi folder data\mysql

Karena setiap folder mewakili satu database, kita juga bisa memindahkan database dengan cara meng-copy folder tersebut. Ini adalah cara tercepat untuk copy database tanpa perlu masuk ke aplikasi MySQL. Namun cara ini hanya bisa dilakukan untuk versi MySQL yang sama (dan juga tidak selalu berhasil). Seandainya ada file yang terhapus, tabel di dalamnya juga akan ikut terhapus / error.

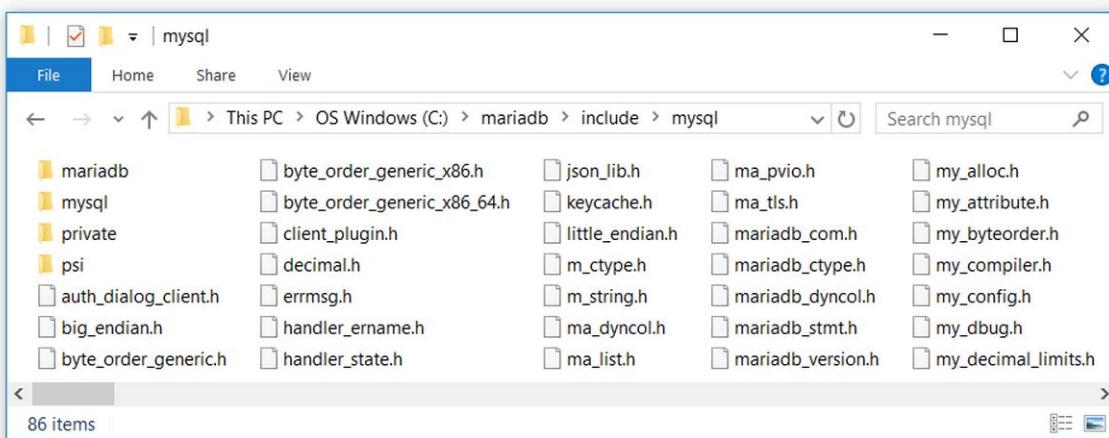


Di dalam MariaDB bawaan XAMPP, terdapat folder *backup*. Folder backup ini berisi cadangan file dari folder **data**.

Folder include

Folder **include** hanya bisa ditemukan pada instalasi MySQL atau MariaDB secara manual (bukan bawaan XAMPP). Folder **lib** juga demikian. MySQL bawaan XAMPP memindahkan kedua folder ini ke tempat lain.

Folder **include** berisi berbagai file library pendukung untuk MySQL (dikenal juga dengan sebutan *header file*). Kita dapat melihat isi dari file yang ada dengan aplikasi text editor seperti Notepad++. Sebagian besar library MySQL dibuat dengan bahasa C dan C++.



Gambar: Isi folder include

Folder lib

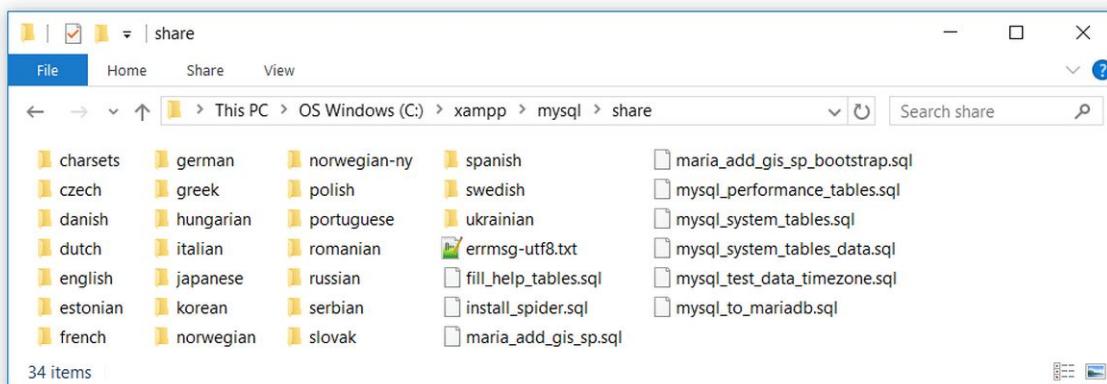
Folder **lib** juga berisi file library MySQL (mirip seperti folder include). Di dalamnya terdapat folder **plugin** untuk menambahkan fungsi tambahan ke dalam MySQL.



Gambar: Isi folder lib

Folder share

Dalam folder **share**, terdapat berbagai folder dengan nama-nama negara. Folder ini berisi file untuk mengatur *character set*, *collation* dan zona waktu terkait perbedaan negara.



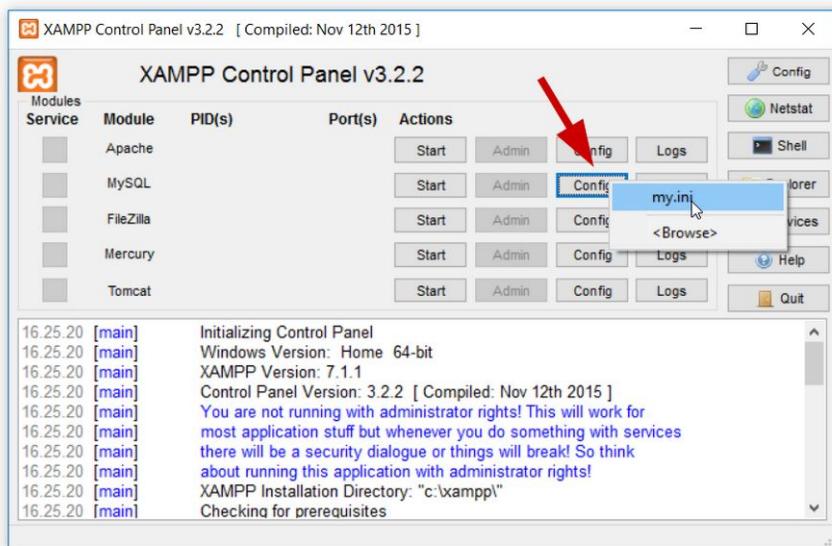
Gambar: Isi folder share

Dari berbagai folder di dalam MySQL / MariaDB, yang paling sering kita akses hanyalah folder bin. Di dalam folder bin inilah terdapat file mysqld.exe dan mysql.exe. Keduanya digunakan untuk menjalankan MySQL server dan MySQL client.

5.2 File Setingan MySQL: my.ini

MySQL dan MariaDB memiliki sebuah file khusus yang bernama *my.ini*. File *my.ini* berisi berbagai pengaturan (setting) terkait MySQL, seperti nomor port yang digunakan, lokasi folder database, jumlah maksimal memory yang dipakai, dll.

Jika menggunakan MariaDB bawaan XAMPP, cara mengakses file *my.ini* sangat praktis. Cukup buka XAMPP Control Panel, klik tombol config dibaris MySQL, kemudian pilih *my.ini*:



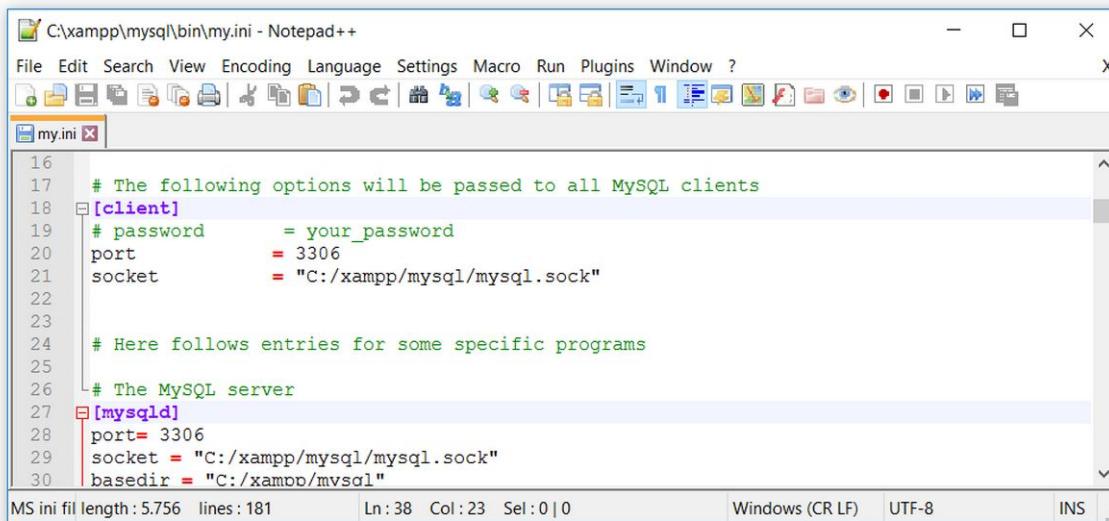
Gambar: Cara membuka file my.ini dari XAMPP Control Panel

Jika anda ingin mengakses manual, lokasi file *my.ini* berada di C:\xampp\mysql\bin\my.ini.



Untuk instalasi MySQL atau MariaDB secara terpisah, file *my.ini* berada di C:\mysql\data\my.ini dan C:\mariadb\data\my.ini.

File **my.ini** berupa file teks biasa. Anda bisa membukanya menggunakan teks editor seperti **Notepad++** maupun Notepad bawaan Windows. Jika ada pengaturan yang ingin diubah, tinggal edit, kemudian save. Setingan tersebut akan aktif pada saat MySQL server di restart.



Gambar: Membuka file my.ini menggunakan Notepad++

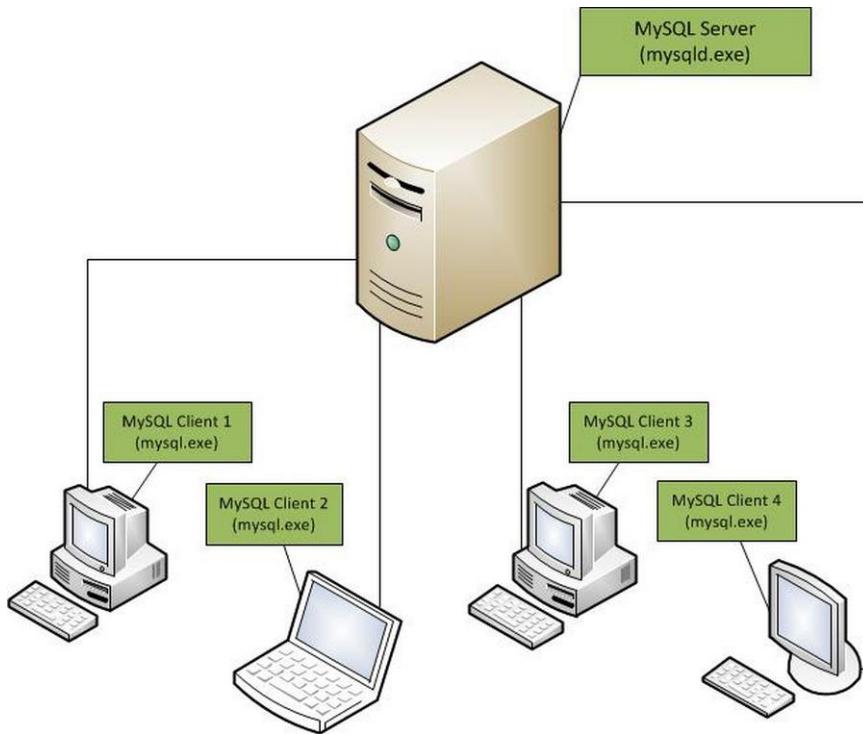
Pembahasan mengenai pengaturan **my.ini** akan kita pelajari secara bertahap.

5.3 Arsitektur Client-Server MySQL

Dalam operasionalnya, MySQL menggunakan konsep **arsitektur client-server**. Arsitektur ini mirip seperti yang ada pada web browser dan web server.

Salah satu komputer bertindak sebagai **server**. Di dalam server inilah seluruh database MySQL disimpan. Kemudian komputer lain berperan sebagai **client**. Client berinteraksi dengan server agar bisa mengakses database. Jumlah client yang mengakses server bisa 1 komputer, 100, atau bahkan 1000 komputer pada saat yang bersamaan.

Sepanjang praktik dalam buku ini, kita akan menjalankan kedua proses ini dalam satu komputer saja. Sehingga di dalam komputer yang sama akan terdapat **MySQL server**, sekaligus **MySQL client** yang jalan bersamaan.

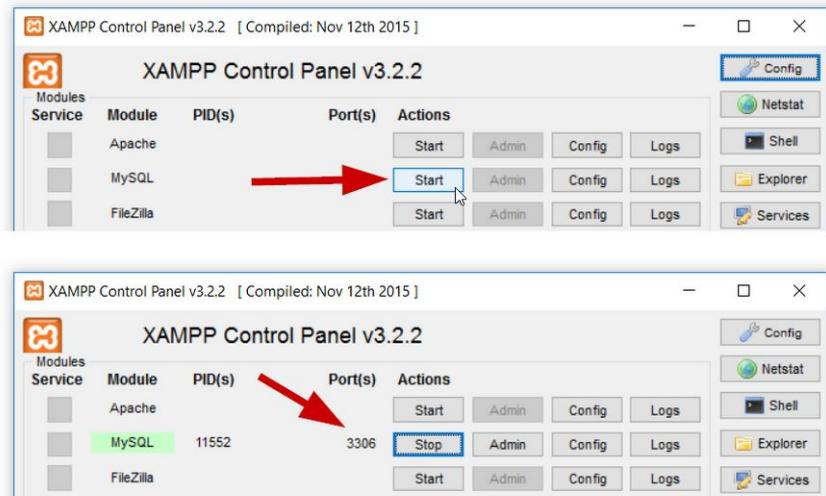


Gambar: Ilustrasi konsep Client-Server MySQL

5.4 Menjalankan MySQL Server (mysqld.exe)

Untuk menjalankan MySQL server, kita harus mengakses file **mysqld.exe** yang berada di folder **bin**. Akhiran “d” dari **mysqld.exe** berarti “*daemon*”, suatu istilah dalam sistem UNIX untuk aplikasi yang terus berjalan di background.

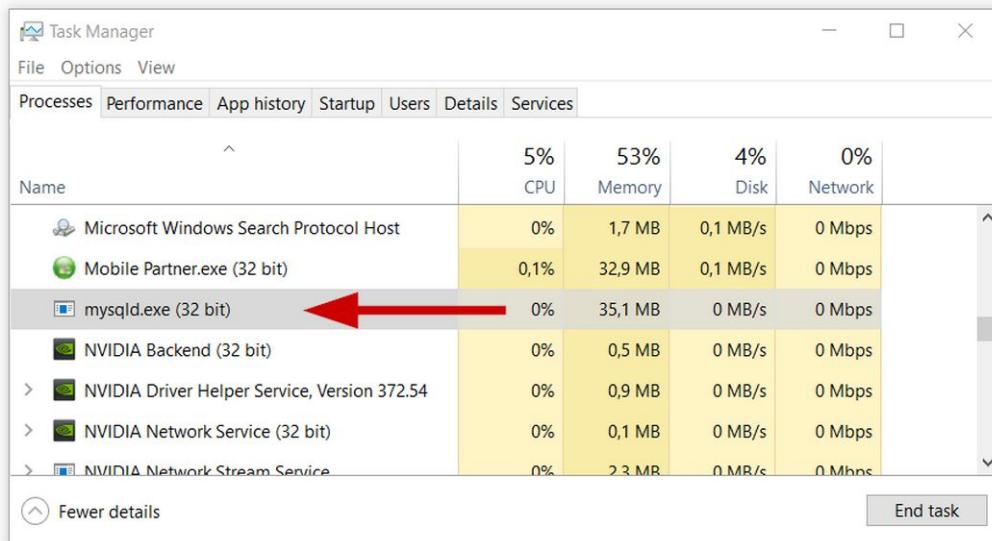
Terdapat beberapa cara menjalankan mysqld.exe. Jika anda menggunakan XAMPP, tinggal klik tombol **Start** pada baris **MySQL** dari XAMPP Control Panel.



Gambar: Menjalankan MySQL Server dari XAMPP Control Panel

Warna background dari teks “MySQL” akan berwarna hijau, kemudian tampil nomor port MySQL: 3306, serta tombol Start berubah menjadi Stop.

Untuk memastikan, buka Task Manager, kemudian cari program **mysqld.exe**. Jika ada, artinya MySQL server sudah berjalan.



Gambar: Proses mysqld.exe sudah berjalan

Menghentikan MySQL server bawaan XAMMP juga cukup mudah, tinggal klik tombol Stop dari XAMPP Control Panel.

Cara kedua adalah dengan mengakses file **mysqld.exe** langsung dari aplikasi command prompt (cmd) Windows. Cara ini bisa digunakan untuk XAMPP, maupun aplikasi MySQL / MariaDB yang diinstall terpisah.

Buka aplikasi cmd dengan cara mengetik “**cmd**” di kolom search Windows, atau akses dari menu **Start** → **All programs** → **Accessories** → **Command Prompt**.

Di aplikasi cmd, ketik alamat file mysqld.exe, yakni di C:\xampp\mysql\bin\mysqld.exe. Berikut hasil yang saya dapat di Windows 10:

```
cmd Command Prompt - C:\xampp\mysql\bin\mysqld.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Andre>C:\xampp\mysql\bin\mysqld.exe
2017-05-10 21:42:30 8360 [Note] C:\xampp\mysql\bin\mysqld.exe (mysqld 10.1.21-MariaDB) starting
as process 7408 ...
```

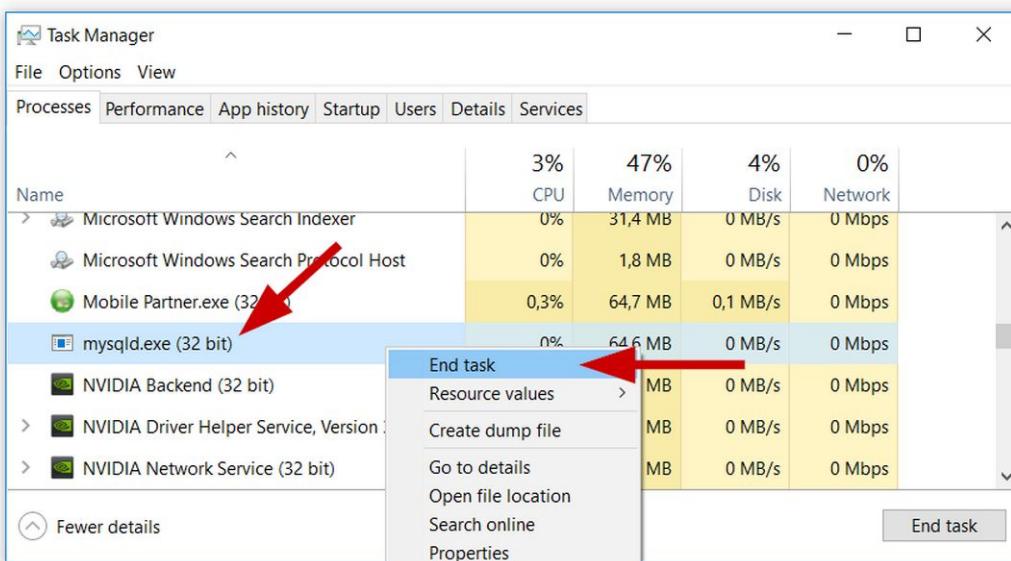
Gambar: Memanggil file mysqld.exe dari cmd Windows

Setelah memanggil **mysqld.exe**, akan tampil teks berikut:

```
2017-05-10 21:42:30 8360 [Note] c:\xampp\mysql\bin\mysqld.exe (mysqld 10.1.2\ 
1-MariaDB) starting as process 7408...
```

Cursor di cmd windows akan tertahan (kita tidak bisa mengetik perintah lain), namun hal ini tidak masalah. Jika pun jendela cmd ditutup, aplikasi **mysqld.exe** tetap berjalan. Ini bisa diperiksa dari Task Manager.

Untuk menghentikan mysqld.exe, cara yang paling praktis adalah dari Task Manager. Di dalam tab Processes cari **mysqld.exe**, klik kanan, kemudian pilih **End task**.



Gambar: menghentikan mysqld.exe dari Task Manager

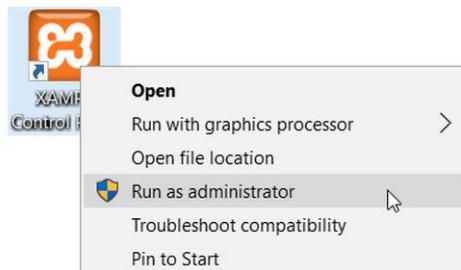
- i Untuk menjalankan MySQL server dari instalasi MySQL atau MariaDB yang bukan bawaan XAMPP, tinggal akses file mysqld.exe di alamat C:\mysql\bin\mysqld.exe dan C:\mariadb\bin\mysqld.exe.

5.5 Menjalankan MySQL Server sebagai Windows Service

MySQL server yang dijalankan sebelum ini hanya bertahan sampai komputer di shutdown atau MySQL server di hentikan. Jika anda ingin MySQL server langsung berjalan begitu Windows aktif (tanpa perlu menjalankan secara manual), bisa menginstallnya sebagai **windows service**, atau sering juga disebut sebagai **MySQL Service**.

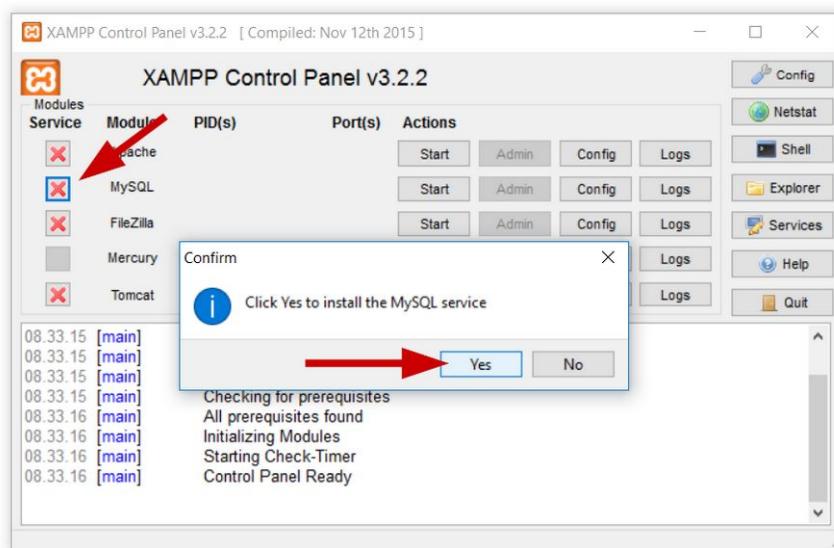
Yang harus menjadi perhatian, jika MySQL server dijalankan sebagai windows service, **MySQL server akan selalu aktif**. Ini bisa menjadi masalah ketika kita menjalankan kembali mysqld.exe karena akan bentrok dengan MySQL service. Selain itu MySQL service juga akan terus menggunakan memory meskipun tidak dibutuhkan lagi.

Untuk menginstall MySQL Service juga terdapat beberapa cara. Jika menggunakan XAMPP, bisa mengaksesnya dari XAMPP Control Panel. Namun XAMPP Control Panel ini harus dijalankan dengan user **Administrator**. Caranya, klik kanan icon XAMPP di desktop, kemudian pilih “Run as Administrator”.



Gambar: Menjalankan XAMPP Control Panel sebagai Administrator

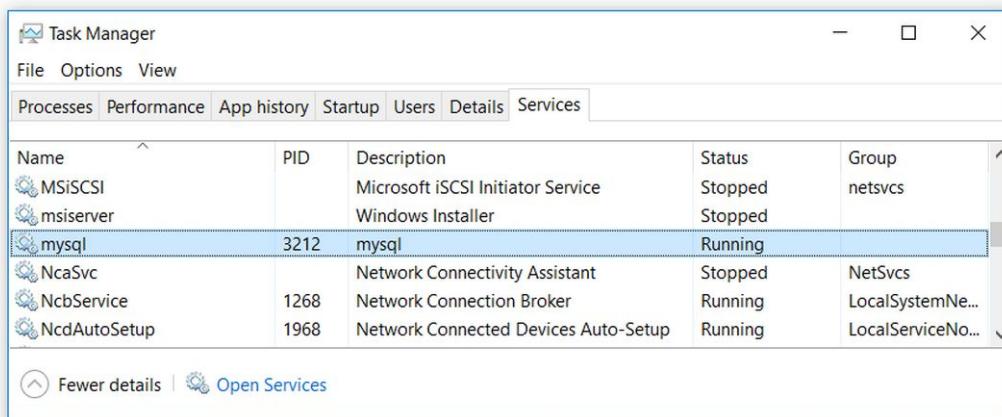
Klik icon silang di sebelah kiri tulisan MySQL, klik “Yes” di jendela konfirmasi “Click Yes to install the MySQL Service”.



Gambar: Install MySQL Service

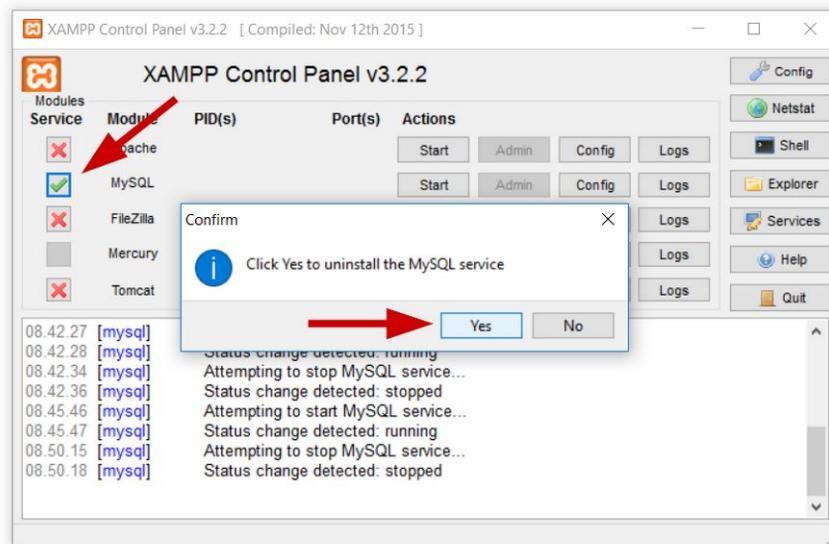
Sekarang MySQL server sudah terinstall sebagai service, namun tidak langsung aktif. Klik tombol Start untuk menjalankan MySQL server.

Untuk memeriksanya, bisa dari Task Manager. Pilih tab service, cari “mysql”. Jika ditemukan artinya MySQL service sudah berjalan.



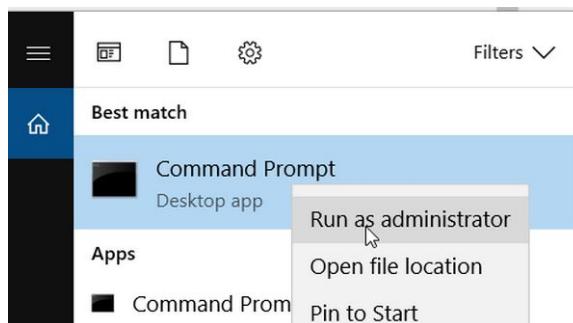
Gambar: MySQL Service sudah berjalan

Cara menghentikan MySQL service juga sama. Buka XAMPP Control Panel, Stop MySQL server terlebih dahulu, kemudian klik tombol checklist di sebelah kiri MySQL.



Gambar: Menghentikan MySQL Service dari XAMPP Control Panel

Cara kedua untuk menginstall MySQL server sebagai Windows Service adalah dari cmd. Cmd ini juga harus dijalankan sebagai **Administrator**. Caranya, klik kanan icon cmd kemudian pilih “Run as Administrator”.



Gambar: Menjalankan cmd sebagai Administrator

Untuk menginstall MySQL server sebagai Windows Service, ketik perintah:

```
c:\xampp\mysql\bin\mysqld --install
```

Jika penulisannya benar akan tampil teks “**Service successfully installed.**” Namun jika yang tampil adalah “*Install/Remove of the Service Denied!*” artinya cmd belum dijalankan sebagai administrator.

Meskipun MySQL service sudah terinstall, tapi belum langsung aktif. Jalankan dengan perintah:

```
net start mysql
```

Jika tidak ada masalah, akan tampil teks:

```
The MySQL service is starting..
The MySQL service was started successfully.
```

A screenshot of a Command Prompt window titled "Administrator: Command Prompt". The window shows the following text output:

```
Administrator: Command Prompt
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>c:\xampp\mysql\bin\mysqld --install
Service successfully installed.

C:\WINDOWS\system32>net start mysql
The MySQL service is starting..
The MySQL service was started successfully.

C:\WINDOWS\system32>
```

Gambar: MySQL service sudah berhasil diinstall

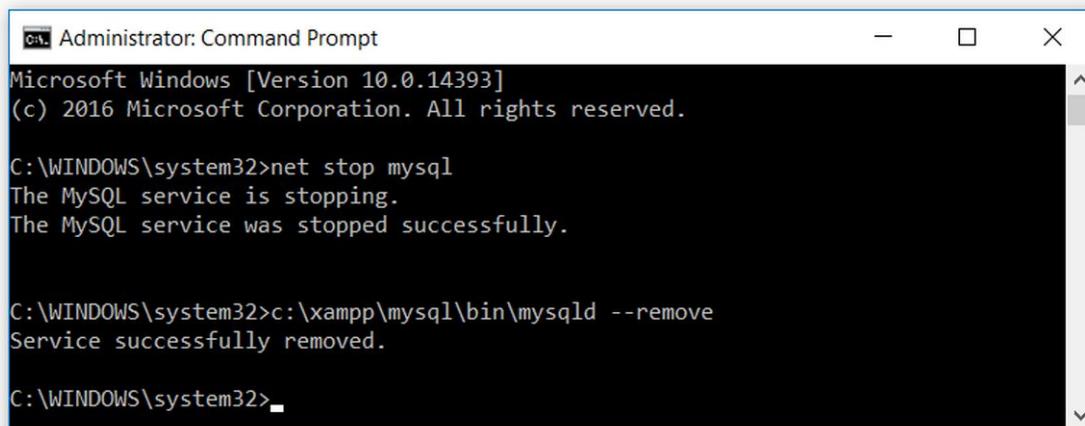
Anda bisa periksa dari Task Manager dan seharusnya sudah ada service: MySQL.

Untuk menghapus MySQL Service menggunakan cmd, pertama kita hentikan dulu MySQL server dengan perintah:

```
net stop mysql
```

Kemudian uninstall MySQL Service dengan perintah

```
c:\xampp\mysql\bin\mysqld --remove
```



The screenshot shows an 'Administrator: Command Prompt' window on Windows 10. The command prompt is black with white text. The output is as follows:

```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>net stop mysql
The MySQL service is stopping.
The MySQL service was stopped successfully.

C:\WINDOWS\system32>c:\xampp\mysql\bin\mysqld --remove
Service successfully removed.

C:\WINDOWS\system32>
```

Gambar: Men-uninstall MySQL Service dari Task Manager



Cara menghentikan MySQL service ini juga berguna jika terdapat error saat menjalankan MySQL server. Kemungkinan besar bentrok dengan MySQL service yang sedang berjalan di background.

5.6 Menjalankan MySQL Client

MySQL client digunakan untuk berkomunikasi dengan MySQL server. MySQL client hadir dengan berbagai bentuk. Ada yang diakses dari cmd, berbasis web, maupun yang berbasis aplikasi desktop.

XAMPP menyediakan halaman **phpMyAdmin** sebagai MySQL client. phpMyAdmin adalah bentuk MySQL client yang berbasis web. Jika anda menginstall MySQL secara terpisah, ada pilihan untuk menginstall aplikasi **MySQL Workbench**, ini adalah MySQL client yang berbentuk aplikasi desktop.

MySQL client yang paling sederhana (dan yang akan saya pakai sepanjang buku ini) adalah berbasis cmd (command prompt). Dimana kita mengetik perintah-perintah SQL secara manual untuk berkomunikasi dengan MySQL server.

Jika sebelumnya anda sudah pernah menggunakan phpMyAdmin, tentu merasa sangat mudah dalam mengelola database MySQL menggunakan aplikasi tersebut. Kita tinggal klik, pilih menu yang ada, dan dalam beberapa menit database sudah selesai dibuat tanpa perlu menulis perintah SQL apapun.

Di balik kepraktisannya, aplikasi phpMyAdmin kurang cocok untuk proses belajar. Inti dari memahami MySQL ada di penulisan perintah SQL atau “query”. Di PHP nanti, perintah SQL inilah yang harus ditulis agar bisa membuat aplikasi database berbasis web. Aplikasi phpMyAdmin baru cocok dipakai jika kita sudah paham sebagian besar query MySQL.

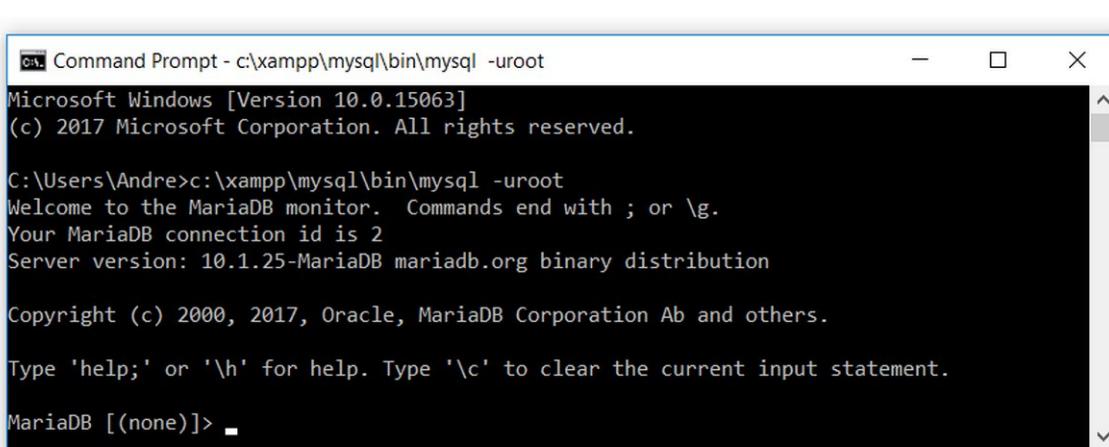
Baik mari kita masuk ke praktik menjalankan MySQL client dari cmd. Caranya mirip seperti menjalankan MySQL server, hanya saja kali ini yang diakses adalah file **bin\mysql.exe**.

Sebelum mulai, pastikan MySQL server sudah berjalan, boleh dari XAMPP Control Panel atau secara manual dari cmd. Kemudian, buka kembali cmd dan ketik perintah berikut:

```
c:\xampp\mysql\bin\mysql -uroot
```

Jika penulisannya benar, akan tampil teks “welcome” dari MariaDB:

```
Welcome to the MariaDB monitor. Commands end with ; or \g.  
Your MariaDB connection id is 2  
Server version: 10.1.25-MariaDB mariadb.org binary distribution  
  
Copyright (c) 2000, 2016, Oracle, MariaDB Corporation Ab and others.  
  
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```



The screenshot shows a Microsoft Windows Command Prompt window titled "Command Prompt - c:\xampp\mysql\bin\mysql -uroot". The window displays the following text:
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\Andre>c:\xampp\mysql\bin\mysql -uroot
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 2
Server version: 10.1.25-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> ■

Gambar: Login sebagai user root

Perintah **c:\xampp\mysql\bin\mysql -uroot** artinya, saya mencari file **mysql.exe** yang berada di folder **c:\xampp\mysql\bin**, kemudian masuk sebagai user **root**.

Untuk keluar dari MySQL client, ketik perintah: **exit**.

```
C:\Users\Andre>c:\xampp\mysql\bin\mysql -uroot
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 2
Server version: 10.1.25-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> exit
Bye

C:\Users\Andre>
```

Gambar: Logout dari MySQL client

Perintah `mysql -uroot` hanya bisa dipakai untuk mengakses MariaDB bawaan XAMPP yang tidak memiliki password. Kalau dipakai untuk MySQL / MariaDB yang diinstall secara terpisah, akan keluar pesan error. Kenapa? Karena user root untuk MySQL / MariaDB tersebut menggunakan password.

Saya akan coba mengakses MariaDB yang diinstall terpisah. Jika anda masih ingat, saya membuat password “**rahasia**” untuk user **root** pada saat proses instalasi. Berikut perintah untuk masuk sebagai user root ke dalam MariaDB:

```
c:\mariadb\bin\mysql -uroot -prahasia
```

```
C:\Users\Andre>c:\mariadb\bin\mysqld.exe
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Andre>c:\mariadb\bin\mysqld.exe
2017-05-12 19:33:31 10724 [Note] options --log-slow-admin-statements, --log-queries-not-using-indexes and --log-slow-slave-statements have no effect if --log-slow-queries is not set
2017-05-12 19:33:31 10724 [Note] c:\mariadb\bin\mysqld.exe (mysqld 10.2.5-MariaDB) starting as process 4988 ...

C:\Users\Andre>c:\mariadb\bin\mysql -uroot -prahasia
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Andre>c:\mariadb\bin\mysql -uroot -prahasia
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 8
Server version: 10.2.5-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2016, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

Gambar: Login sebagai user root menggunakan password

Mari kita bahas perintah yang digunakan untuk masuk ke MySQL client ini dengan lebih detail. Berikut format dasar perintah login MySQL:

```
mysql -h host -u user -p_password
```

- **mysql** adalah program MySQL client yang kita gunakan untuk mengakses server (file aslinya bernama: **mysql.exe**).
- **-h** adalah perintah untuk menginput **host**. Host disini adalah alamat IP komputer dimana MySQL server berada. Karena kita menjalankan MySQL server pada komputer yang sama dengan MySQL client, alamat IP tersebut adalah **127.0.0.1** atau **localhost**. Localhost merupakan nilai default, sehingga jika nilai host tidak ditulis, akan dianggap sebagai “localhost”.
- **-u** adalah perintah untuk menginput nama **user**.
- **-p** adalah perintah untuk menginput **password**. Penulisan inputan password harus langsung digabungkan dengan **-p**. Jika saya menggunakan password “**rahasia**”, maka penulisannya menjadi **-prahasia**, tidak boleh **-p rahasia** (terdapat spasi antara “**-p**” dan “**rahasia**”)

Untuk masuk ke MySQL server di **localhost** sebagai user **root** dengan password **rahasia**, perintah nya adalah:

```
c:\mariadb\bin\mysql -h localhost -u root -prahasia
```

Karena localhost adalah nilai default dari host, perintah diatas boleh disingkat:

```
c:\mariadb\bin\mysql -u root -prahasia
```

Apabila MySQL server berada di komputer lain yang beralamat IP **10.12.254.14**, maka perintahnya menjadi:

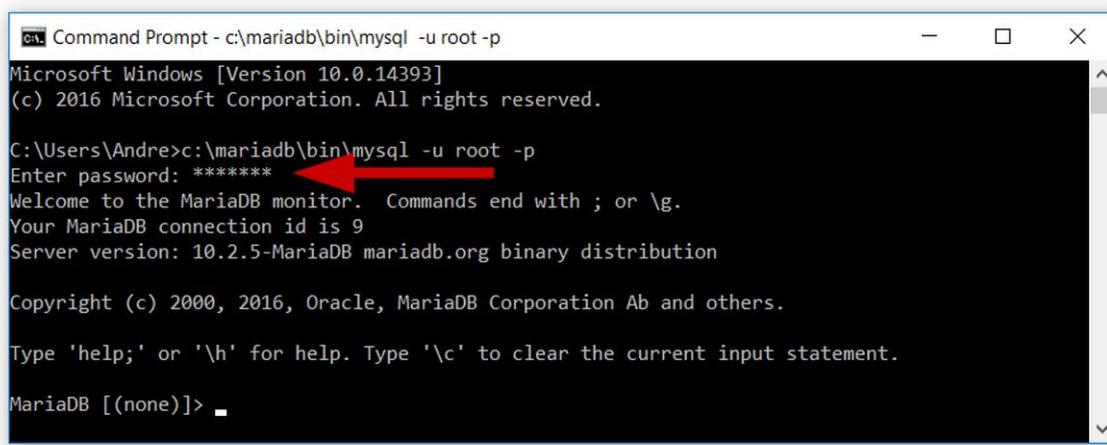
```
c:\mariadb\bin\mysql -h 10.12.254.14 -u root -prahasia
```

Dalam perintah diatas, password untuk user langsung tertulis dan mudah untuk dilihat. Agar lebih aman (dari orang yang mengintip password di belakang anda), bisa menggunakan perintah alternatif:

```
c:\mariadb\bin\mysql -u root -p
```

Tampilan cmd akan berhenti sesaat untuk menunggu kita menginputkan password:

```
Enter password:*****
```



```
Command Prompt - c:\mariadb\bin\mysql -u root -p
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\Andre>c:\mariadb\bin\mysql -u root -p
Enter password: *****  

Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 9
Server version: 10.2.5-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2016, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> -
```

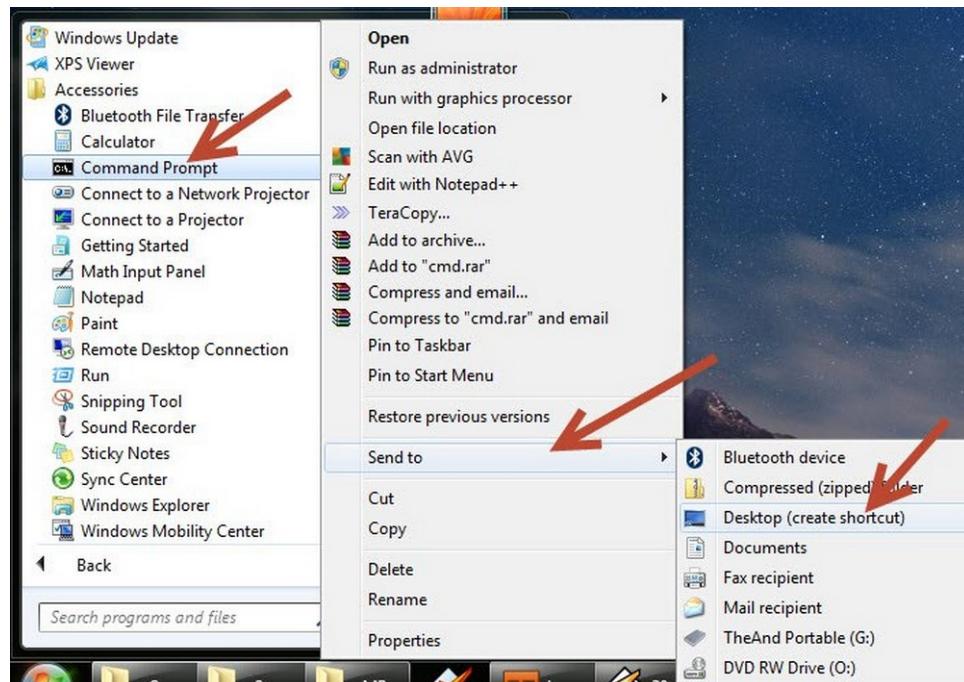
Gambar: Menginput password secara terpisah

5.7 Membuat Shorcute untuk Folder mysql/bin

Sepanjang bab ini, saya mengakses file **mysqld.exe** (MySQL server) dan **mysql.exe** (MySQL client) dengan cara mengetik lengkap alamat file tersebut, seperti **c:\xampp\mysql\bin\mysqld** dan **c:\xampp\mysql\bin\mysql**.

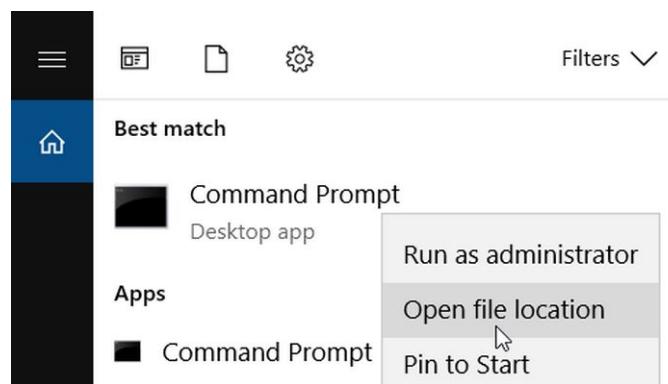
Karena kita akan sering mengakses alamat ini dari cmd, akan lebih praktis mempersiapkan sebuah *shortcut* yang ketika di klik langsung terbuka di folder tersebut.

Cari aplikasi **cmd** dari menu Start → All programs → Accessories → Command Prompt. Kemudian klik kanan, pilih Send to, lalu pilih Desktop (create shortcut).

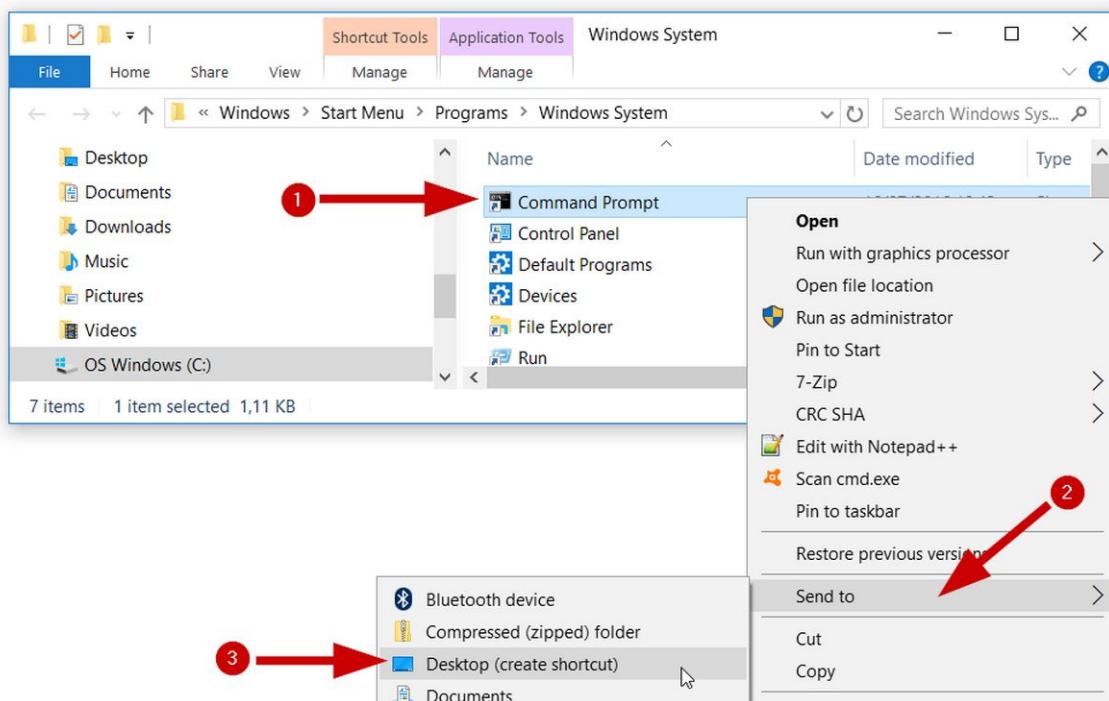


Gambar: Membuat shortcut cmd

Untuk Windows 10, akses dari Search -> cmd, klik kanan kemudian pilih “Open file location”. Klik kanan icon cmd, kemudian Send to, lalu pilih Desktop (create shortcut).

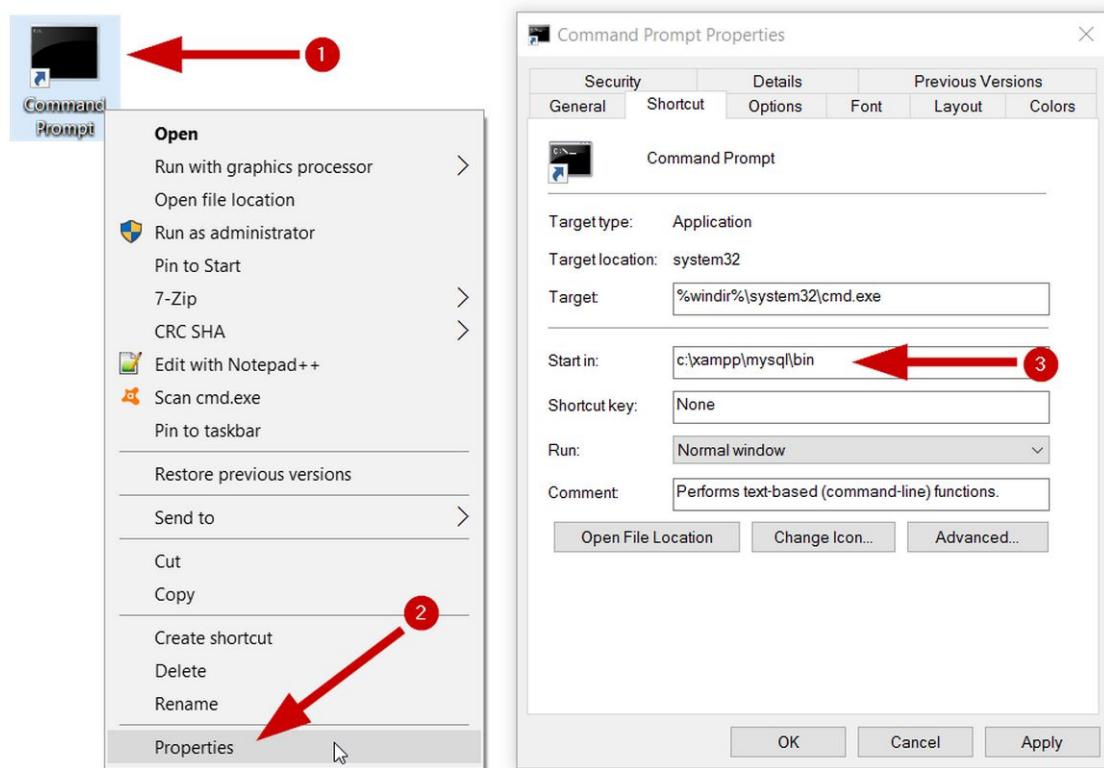


Gambar: Pilih “Open file location” Windows 10



Gambar: Membuat shortcut cmd Windows 10

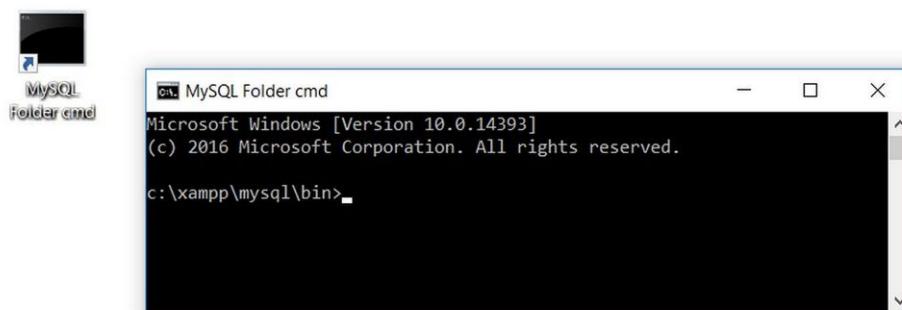
Shortcut cmd akan tampil di desktop. Sekarang kita akan ubah lokasi default dari cmd. Klik kanan shortcut cmd ini, lalu pilih properties. Dari tab Shortcut, cari input box “Start in”. Ubah isian kotak input ini menjadi “c:\xampp\mysql\bin”. Lalu klik OK.



Gambar: Mengubah lokasi start ini cmd

Terakhir, silahkan me-rename shortcut ini menjadi lebih spesifik, misalnya: “MySQL Folder cmd”.

Untuk uji coba, double klik shorcut ini. Seharusnya folder aktif dari cmd akan langsung terbuka di “c:\xampp\mysql\bin”.



Gambar: Hasil akhir shortcut MySQL

Dalam bab ini kita telah membahas cara konsep client - server MySQL, menjalankan MySQL server, menjalankan MySQL Client dan membuat shortcut cmd untuk folder c:\xampp\mysql\bin. Selanjutnya kita akan masuk ke praktik penggunaan bahasa SQL (Structured Query Language).

6. Dasar Bahasa Query MySQL

Dalam bab ini kita akan membahas dasar penggunaan bahasa SQL atau sering juga disingkat sebagai “query”. Query inilah yang nantinya diketik ke dalam MySQL agar bisa memproses data.

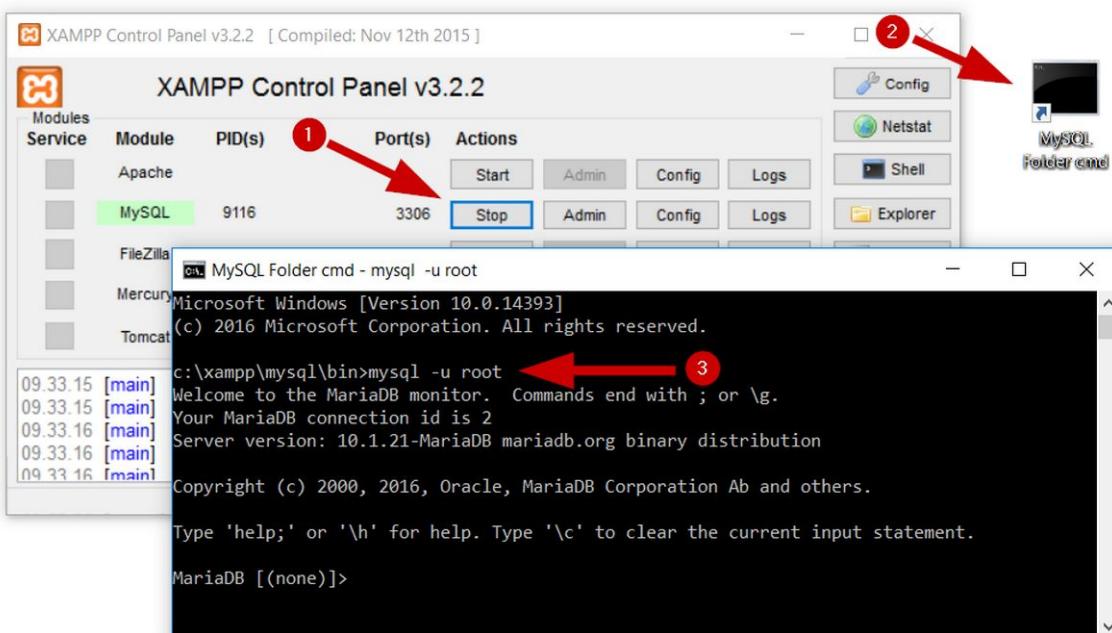
Materi tentang query akan saling berhubungan. Sebagai contoh, ketika kita membahas cara membuat tabel (query CREATE), hasilnya baru kelihatan saat tabel ditampilkan (query SELECT).

Oleh karena itu di dalam bab ini saya akan membahas sekilas berbagai query dasar MySQL, seperti CREATE, INSERT, UPDATE, dan SELECT. Materi yang lebih detail akan dibahas dalam bab tersendiri.

Mulai dari bab ini dan seterusnya, saya juga akan menggunakan MariaDB bawaan XAMPP. Jika anda ingin memakai MySQL / MariaDB yang diinstall secara terpisah juga tidak masalah.

Untuk belajar bahasa SQL, saya menggunakan metode yang paling sederhana, yakni dari cmd Windows. Tujuannya, agar perintah query lebih mudah dipahami. Menggunakan phpMyAdmin memang lebih praktis, tapi kurang pas untuk proses belajar. Di cmd Windows, setiap perintah SQL harus ditulis manual, tidak sekedar klik-klik seperti di phpMyAdmin.

Sebelum memulai, pastikan MySQL server dan MySQL client sudah berjalan. Kemudian login sebagai user **root**. Jika tidak ada masalah, cursor cmd akan diawali dengan teks MariaDB [(none)]>



Gambar: Menjalankan MySQL Client dan login sebagai root

6.1 MySQL Manual

Perintah query di dalam MySQL sangat sangat banyak. Mustahil anda (dan juga saya) bisa menghapal seluruh query ini. Jangan khawatir jika tidak bisa mengingat perintah SQL yang paling sederhana sekalipun. Perintah SQL untuk pembuatan tabel misalnya, hanya perlu sekali di awal pembuatan aplikasi sehingga jarang dipakai dan memang akan sering lupa.

Buku seperti **MySQL Uncover** ini bisa menjadi pegangan sebagai referensi perintah-perintah query MySQL. Semua query ini tidak perlu di hapal, cukup dipahami saja. Jika pun nanti lupa, tinggal buka buku sebentar dan langsung cari materi yang membahas query tersebut.

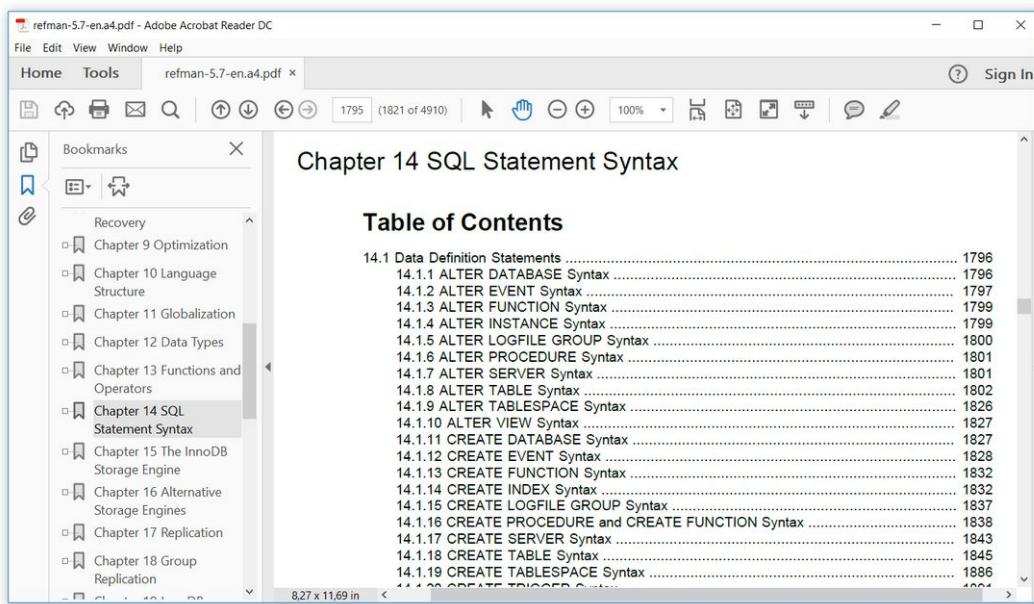
Jika anda butuh referensi yang lebih lengkap terkait query MySQL, tidak ada yang selengkap *MySQL Manual*. **MySQL Manual** adalah dokumentasi resmi dari MySQL yang berisi seluruh perintah query serta penjelasan tambahan lain.

MySQL manual bisa diakses dari <https://dev.mysql.com/doc/refman/5.7/en/>¹. Untuk MariaDB dokumentasinya juga bisa diakses di <https://mariadb.com/kb/en/mariadb/documentation/>².

MySQL Manual juga tersedia dalam versi dokumen pdf yang bisa di download dan dibuka secara offline: [MySQL 5.7 Manual](#)³ (37MB).

Dengan total 4910 halaman A4, MySQL Manual menjadi salah satu dokumen pdf paling tebal yang pernah saya lihat. Ingin menguasai penuh MySQL? Inilah materi yang perlu anda baca.

Namun bahasa yang digunakan memang sangat teknis dan cukup susah dipahami bagi yang bisa bahasa inggris sekalipun. MySQL Manual baru pas bagi anda yang sudah berpengalaman dan butuh referensi lengkap terkait MySQL.



Gambar: MySQL Manual versi pdf

¹<https://dev.mysql.com/doc/refman/5.7/en/>

²<https://mariadb.com/kb/en/mariadb/documentation/>

³<https://downloads.mysql.com/docs/refman-5.7-en.a4.pdf>

6.2 Aturan Penulisan Query MySQL

Dasar teori tentang apa itu bahasa SQL sudah kita bahas dalam bab “**Pengantar Teori Database dan SQL**”. Disana saya menulis bahwa SQL (Structured Query Language) adalah *bahasa pemrograman khusus untuk berkomunikasi dengan aplikasi RDBMS seperti MySQL*.

Bahasa **SQL** (atau sering disingkat dengan “query” saja), terlihat seperti perintah sederhana dalam bahasa inggris, seperti “SELECT nama_provinsi FROM provinsi”. Mari kita coba menulis beberapa query MySQL.

Dengan cmd MySQL client yang terbuka, silahkan ketik perintah berikut. Akhiri dengan tombol **Enter**:

```
MariaDB [(none)]> SELECT NOW();
+-----+
| NOW()           |
+-----+
| 2017-05-23 09:56:47 |
+-----+
1 row in set (0.00 sec)
```

Perintah **SELECT NOW()** berfungsi untuk menampilkan tanggal dan waktu saat ini yang diambil dari MySQL server. Bagian MariaDB [(none)]> tidak perlu ditulis karena ini hanya sebagai penanda cursor. Jika menggunakan MySQL, cursornya berupa mysql>.



Agar lebih praktis, hasil penulisan query akan saya tampilkan berupa teks seperti diatas (tidak lagi menggunakan gambar).

Anda bisa perhatikan bahwa setiap perintah SQL harus diakhiri dengan tanda titik koma (;). Selain menggunakan tanda titik koma, query MySQL juga bisa diakhiri dengan tanda (\g).

```
MariaDB [(none)]> SELECT NOW() \g
+-----+
| NOW()           |
+-----+
| 2017-05-23 09:58:13 |
+-----+
1 row in set (0.00 sec)
```

Perintah SQL baru akan dieksekusi setelah tombol Enter ditekan. Selama query MySQL belum diakhiri dengan tanda ; atau \g maka tetap dianggap satu perintah.

Query diatas juga bisa ditulis menjadi seperti ini:

```
MariaDB [(none)]> SELECT
    -> NOW()
    -> ;
+-----+
| NOW()           |
+-----+
| 2017-05-23 09:59:06 |
+-----+
1 row in set (0.00 sec)
```

Ketika mengetik `SELECT` dan menekan tombol `Enter`, cursor cmd hanya menampilkan tanda ->. Ini artinya perintah tersebut belum selesai. Saat saya menginput `NOW()`, cursor juga masih dengan tanda ->. Barulah di baris ketiga tanda ‘ ; ’ akan mengakhiri penulisan query.

Disini juga terlihat bahwa *whitespace* (karakter spasi, tab, maupun enter) tidak berpengaruh apa-apa ke dalam SQL. Kita bisa menjalankan perintah `SELECT NOW()` dalam 1 baris atau memecahkan menjadi 3 baris. Memisahkan penulisan query menjadi beberapa baris umum dipakai untuk perintah SQL yang panjang.

Hampir untuk semua perintah query, MySQL akan menampilkan jumlah baris yang dipengaruhi dan lama waktu eksekusi.

Pada contoh diatas, terdapat keterangan: `1 row in set (0.00 sec)`. Keterangan ini berarti query tersebut diproses selama 0 detik (0 second), dan mempengaruhi 1 baris (1 row). Ini bukan berarti query tersebut tampil seketika (tanpa proses). Hanya saja karena perintah itu cukup sederhana, MySQL (mungkin) hanya perlu waktu 0,001 sekian detik untuk memprosesnya (dibulatkan menjadi 0,00).



Untuk menghemat tempat, baris keterangan seperti `1 row in set (0.00 sec)` tidak lagi saya tampilkan, kecuali jika keterangan ini perlu kita bahas.

Mari kita coba query lain. Perintah berikut digunakan untuk menampilkan nama user yang sedang aktif dan versi MySQL server yang digunakan pada saat ini. Fungsi yang dipakai adalah `USER()` dan `VERSION()`:

```
MariaDB [(none)]> SELECT NOW(),USER(),VERSION();
+-----+-----+-----+
| NOW()          | USER()        | VERSION()      |
+-----+-----+-----+
| 2017-05-23 10:01:07 | root@localhost | 10.1.21-MariaDB |
+-----+-----+-----+
```

Sama seperti sebelumnya, perintah diatas juga bisa dipecah menjadi beberapa baris:

```
MariaDB [(none)]> SELECT NOW(),
-> USER(),
-> VERSION();
+-----+-----+-----+
| NOW() | USER() | VERSION() |
+-----+-----+-----+
| 2017-05-23 10:01:49 | root@localhost | 10.1.21-MariaDB |
+-----+-----+-----+
```

Untuk menghapus perintah SQL yang sudah ditulis sebagian, bisa menggunakan karakter (\c):

```
MariaDB [(none)]> SELECT USER(),
-> NOW
-> \c
MariaDB [(none)]>
```

Perhatikan bahwa tanda -> akan kembali menjadi MariaDB [(none)]> yang menandakan MariaDB client sudah siap menerima perintah baru (perintah lama akan dibatalkan).

Untuk query yang hasilnya melebihi lebar jendela cmd, akan menjadi sulit dibaca. MySQL menyediakan cara untuk merubah tampilan tabel menjadi baris. Yakni dengan mengganti tanda titik koma (;) menjadi (\G), sebuah karakter forward slash dan huruf G besar:

```
MariaDB [(none)]> SELECT NOW(),USER(),VERSION() \G
***** 1. row *****
NOW(): 2017-05-23 10:03:59
USER(): root@localhost
VERSION(): 10.1.21-MariaDB
1 row in set (0.00 sec)
```

Dalam MySQL dan MariaDB, huruf besar dan kecil umumnya tidak dibedakan (**case insensitive**). Sebagai contoh, 3 perintah ini akan menghasilkan tampilan yang sama:

```
MariaDB [(none)]> SELECT version();
+-----+
| version() |
+-----+
| 10.1.21-MariaDB |
+-----+
```

```
MariaDB [(none)]> SELECT VERSION();
+-----+
| VERSION() |
+-----+
| 10.1.21-MariaDB |
```

```
+-----+
MariaDB [(none)]> sE1EcT veRsIOn();
+-----+
| veRsIOn()      |
+-----+
| 10.1.21-MariaDB |
+-----+
```

Namun untuk penulisan **nama database** dan **nama tabel**, MySQL akan mengikuti sistem operasi dimana MySQL server berjalan. Untuk OS Windows, nama database `mahasiswa` dianggap sama dengan `MaHaSiSwA` maupun `MAHASISWA`. Tetapi di OS Linux, ketiga database ini dianggap berbeda (**case sensitive**).

Karena alasan ini, sebaiknya kita konsisten dalam penulisan huruf besar dan kecil. Terlebih mayoritas web hosting menggunakan OS Linux (saat website ingin di-online-kan nanti).

Disarankan untuk selalu menggunakan huruf kecil dalam penulisan database, tabel dan nama variabel. Khusus untuk *keyword* dan fungsi bawaan MySQL, ditulis dengan HURUF BESAR, seperti `SELECT`, `CREATE`, `WHERE`, dll. Aturan penulisan ini memang tidak wajib, tapi sudah menjadi sebuah standar tidak resmi dan banyak dipakai oleh programmer serta buku-buku panduan MySQL.

Dalam beberapa kasus, sebuah query bisa saja menggunakan tanda baca selain angka dan huruf seperti: , (koma), \ (forward slash/garis miring depan), ' (tanda petik), dan " " (spasi). Misalnya untuk 123 dan '123' atau 12e + 14 dan 12e+14 akan memiliki arti berbeda. Jika anda menemui error, mungkin penggunaan "tanda" ini letak masalahnya.

Mayoritas error penulisan SQL ada di salah ketik perintah, entah kurang tanda atau salah huruf:

```
MariaDB [(none)]> SELCT version();
```

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual
that corresponds to your MariaDB server version for the right syntax to use
near 'SELCT version()' at line 1
```

Pesan error yang tampil adalah “*You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near ‘SELCT version()’ at line 1*”. Artinya, ada perintah yang error di baris 1 dekat ‘SELCT version()’.

Jika anda menemukan pesan error seperti ini, coba diteliti lagi perintah query yang ditulis. Disini saya menulis “SELCT version()”, dimana kurang satu huruf “E” di perintah “SELCT”.



Sedikit tips untuk menggunakan cmd: Anda bisa menekan tombol panah atas di keyboard untuk menampilkan perintah yang sudah diketik sebelumnya. Jika ingin mengcopy perintah SQL dari eBook ini atau sumber lain, di cmd cukup klik kanan (Windows 10). Atau klik kanan dan pilih paste.

6.3 Membuat dan Menghapus Database

Pada sistem RDBMS (seperti MySQL dan MariaDB), database adalah kumpulan tabel yang saling terhubung. Sebuah aplikasi biasanya hanya butuh satu database. Namun satu database bisa berisi ratusan hingga ribuan tabel (tergantung kompleksitas dari aplikasi yang ingin dibuat). Di dalam database inilah tabel-tabel akan disimpan.

Untuk membuat database, format penulisan querynya adalah sebagai berikut:

```
CREATE DATABASE [IF NOT EXISTS] nama_database;
```



Perintah yang ditandai dengan kurung siku, seperti “[IF NOT EXISTS]” adalah perintah **optional** (boleh tidak ditulis). Ini adalah cara penulisan yang digunakan di panduan resmi MySQL: [MySQL Manual⁴](#).

Mari kita buat sebuah database “indonesia”:

```
MariaDB [(none)]> CREATE DATABASE indonesia;
Query OK, 1 row affected (0.04 sec)
```

Tambahan query [IF NOT EXISTS] berfungsi agar MySQL tidak menampilkan pesan error jika database sudah ada. Misalnya jika saya mencoba membuat ulang database “indonesia”, akan terjadi error:

```
MariaDB [(none)]> CREATE DATABASE indonesia;
ERROR 1007 (HY000): Can't create database 'indonesia'; database exists
```

Pesan error ini berguna untuk mengidentifikasi kesalahan. Apabila kita membuat kode query yang panjang dan dieksekusi secara keseluruhan (yang sering berasal dari PHP), pesan error ini bisa membuat seluruh query gagal berjalan.

Perintah [IF NOT EXISTS] akan membuat database jika database itu belum ada sebelumnya. Jika sudah ada, query CREATE DATABASE tidak akan melakukan apa-apa (database yang lama juga tidak akan tertimpas).

```
MariaDB [(none)]> CREATE DATABASE IF NOT EXISTS indonesia;
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

Bedanya, sekarang ada tambahan **1 warning**. Di dalam MySQL (dan kebanyakan bahasa pemrograman), **error** berbeda dengan **warning**. **Error** terjadi karena kesalahan fatal dan membuat proses berhenti. Sedangkan warning hanya kesalahan kecil (minor) dimana proses masih bisa dilanjutkan. Misalnya menggunakan sebuah fungsi yang *deprecated* (usang) di PHP akan menampilkan sebuah warning.

Untuk melihat pesan warning dari perintah SQL yang telah dieksekusi, gunakan perintah:

⁴<https://dev.mysql.com/doc/refman/5.7/en/>

```
SHOW WARNINGS;
```

Berikut contohnya:

```
MariaDB [(none)]> CREATE DATABASE IF NOT EXISTS indonesia;
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

```
MariaDB [(none)]> SHOW WARNINGS;
```

Level	Code	Message
Note	1007	Can't create database 'indonesia'; database exists

```
1 row in set (0.00 sec)
```

Untuk melihat apa saja database yang tersedia di MySQL server, bisa menggunakan perintah SHOW DATABASES. Berikut contohnya:

```
MariaDB [(none)]> SHOW DATABASES;
```

Database
indonesia
information_schema
mysql
performance_schema
phpmyadmin
test

```
6 rows in set (0.08 sec)
```

Terlihat bahwa selain database `indonesia`, MariaDB dari XAMPP juga memiliki beberapa database bawaan (sedapat mungkin agar tidak menghapus atau mengubah database-database ini)

Untuk menghapus sebuah database, bisa menggunakan perintah berikut:

```
DROP DATABASE [IF EXISTS] database_name;
```

Yang perlu diperhatikan, query `DROP DATABASE` akan menghapus database, termasuk seluruh tabel dan isi dari tabel tersebut. Harap selalu berhati-hati dengan perintah seperti ini, karena di MySQL / MariaDB tidak ada tombol “*undo*”. Jika database telah dihapus, tidak bisa dikembalikan lagi.

Sama seperti query pada pembuatan database, pilihan `[IF EXISTS]` digunakan untuk menghindarkan pesan error jika seandainya database tersebut memang tidak ada.

Untuk menghapus database “`indonesia`”, querynya adalah sebagai berikut:

```
MariaDB [(none)]> DROP DATABASE indonesia;
Query OK, 0 rows affected (0.09 sec)
```

Sekarang, database “indonesia” sudah berhasil dihapus. Mari coba sekali lagi:

```
MariaDB [(none)]> DROP DATABASE indonesia;
ERROR 1008 (HY000): Can't drop database 'indonesia'; database doesn't exist
```

Akan tampil pesan error karena database ‘indonesia’ memang sudah tidak ada. Solusinya, bisa menambahkan perintah IF EXISTS:

```
MariaDB [(none)]> DROP DATABASE IF EXISTS indonesia;
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

```
MariaDB [(none)]> SHOW WARNINGS;
+-----+-----+
| Level | Code | Message |
+-----+-----+
| Note  | 1008 | Can't drop database 'indonesia'; database doesn't exist |
+-----+-----+
1 row in set (0.00 sec)
```

Selain menggunakan keyword DATABASE, MySQL/MariaDB juga mendukung keyword SCHEMA. Yang juga akan membuat sebuah database.

```
MariaDB [(none)]> CREATE SCHEMA indonesia;
Query OK, 1 row affected (0.00 sec)
```

```
MariaDB [(none)]> SHOW DATABASES;
+-----+
| Database |
+-----+
| indonesia |
| information_schema |
| mysql |
| performance_schema |
| phpmyadmin |
| test |
+-----+
```

```
MariaDB [(none)]> DROP SCHEMA indonesia;
```

Terlihat bahwa di MySQL, **schema** sama artinya dengan **database**. Tapi di sistem RBDMS lain seperti ORACLE dan SQL Server, **schema** merupakan sebuah bagian yang terpisah dari database.



Karena query SQL menggunakan bahasa inggris, kita harus hati-hati dalam mengetik kata *singular* (tunggal) dan *plural* (jamak). Misalnya untuk melihat database, querynya adalah SHOW DATABASES, tetapi jika tidak sengaja tertulis SHOW DATABASE, ini akan menjadi error. Perhatikan bahwa perintahnya adalah DATABASES, bukan DATABASE. Begitu juga dengan IF EXISTS, bukan IF EXIST (ada huruf S diakhir kata).

Query terakhir yang berkaitan dengan database adalah untuk memilih *default* database. Format perintahnya adalah:

```
USE nama_database;
```

Jika ingin memilih database `indonesia`, maka querynya adalah:

```
MariaDB [(none)]> USE indonesia;  
Database changed
```

```
MariaDB [indonesia]>
```

Perhatikan ada perubahan cursor dari `MariaDB [(none)]>` menjadi `MariaDB [indonesia]>`. Artinya, setiap perintah query yang kita ketik, akan menggunakan database “`indonesia`” secara default.

6.4 Membuat dan Menghapus Tabel

Database pada dasarnya merupakan sebuah *container* atau penampung dari kumpulan tabel. Di dalam tabel inilah data akan disimpan.

Pembahasan mengenai cara pembuatan tabel cukup panjang, karena kita juga harus mempelajari apa saja tipe data yang bisa disimpan ke dalam suatu tabel. Terkait hal ini, akan saya bahas dengan lebih detail dalam bab tersendiri. Disini kita hanya melihat sekilas cara pembuatan tabel MySQL.

Sebelum membuat tabel, harus dipilih sebuah database default. Mari buat kembali database `indonesia` (jika belum ada) dan pilih sebagai database default:

```
MariaDB [(none)]> CREATE DATABASE IF NOT EXISTS indonesia;  
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

```
MariaDB [(none)]> USE indonesia;  
Database changed  
MariaDB [indonesia]>
```

Format dasar query untuk membuat tabel adalah sebagai berikut:

```
CREATE TABLE [IF NOT EXISTS] nama_tabel
(nama_kolom tipe_kolom [, nama_kolom tipe_kolom] )
```

- Query opsional [**IF NOT EXISTS**] berfungsi sama seperti pada pembuatan database. Perintah ini digunakan agar tidak tampil pesan error jika tabel tersebut telah ada sebelumnya.
- **nama_tabel** diisi dengan nama tabel yang ingin dibuat.
- **nama_kolom** diisi dengan nama kolom.
- **tipe_kolom** diisi dengan tipe data kolom.

Untuk **nama_kolom** dan **tipe_kolom** bisa diulang sesuai dengan jumlah kolom yang dibutuhkan. Jika sebuah tabel terdiri dari 5 kolom, maka perlu 5 kali pendefinisian kolom (**nama_kolom** dan **tipe_kolom**).

Sebagai contoh, saya ingin membuat tabel yang berisi data provinsi di Indonesia. Berikut struktur tabel tersebut:

id_prov	nama_prov	ibu_kota	populasi	tgl_diresmikan
1	Jakarta	Jakarta	10.012.271	28 Agustus 1961
2	Banten	Serang	11.704.877	4 Oktober 2000
3	Kalimantan Selatan	Banjarmasin	3.922.790	7 Desember 1956
4	Kepulauan Riau	Tanjung Pinang	1.917.415	25 Oktober 2002
5	Papua Barat	Manokwari	849.809	4 Oktober 1999
6	Maluku Utara	Sofifi	1.138.667	4 Oktober 1999

Gambar: Tabel provinsi

Tabel diatas terdiri dari 5 kolom:

- Kolom **id_prov**: digunakan untuk menyimpan nomor urut provinsi yang terdiri dari angka bulat. Di dalam MySQL, angka bulat bisa ditampung oleh tipe data INT. Pendefinisian kolomnya: **id_prov** INT.
- Kolom **nama_prov**: digunakan untuk menyimpan nama provinsi yang berupa teks. Di dalam MySQL, teks bisa ditampung oleh tipe data VARCHAR. Sebagai tambahan, saya ingin membatasi nama provinsi maksimal 100 karakter. Pendefinisian kolomnya: **nama_prov** VARCHAR(100).
- Kolom **ibu_kota**: digunakan untuk menyimpan nama ibu kota provinsi. Sama seperti kolom **nama_prov**, isinya juga berupa teks dengan maksimal 100 karakter. Pendefinisian kolomnya: **ibu_kota** VARCHAR(100).
- Kolom **populasi**: digunakan untuk menyimpan jumlah populasi dari suatu provinsi. Data ini disimpan berbentuk angka bulat, sehingga bisa ditampung oleh tipe data INT. Pendefinisian kolomnya: **populasi** INT.
- Kolom **tgl_diresmikan**: digunakan untuk menyimpan tanggal kapan provinsi diresmikan. Karena berupa tanggal, akan saya simpan ke tipe date MySQL. Pendefinisian kolomnya: **tgl_diresmikan** DATE.

Merangkai ke-5 pendefinisian kolom ini, maka query pembuatan tabel provinsi adalah sebagai berikut:

```
MariaDB [indonesia]> CREATE TABLE provinsi
(id_prov INT, nama_prov VARCHAR(100), ibu_kota VARCHAR(100),
populasi INT, tgl_diresmikan DATE);
```

Karena querynya cukup panjang, saya pecah menjadi 3 baris. Anda juga bisa menulisnya dalam 1 baris sekaligus.

Query diatas bertujuan untuk membuat tabel provinsi dengan 5 kolom. Cara pendefinisian kolom adalah dengan format: nama_kolom1 tipe_kolom1, nama_kolom2 tipe_kolom2, dst.

Penjelasan yang lebih detail terkait tipe data akan saya bahas dalam bab khusus. Tapi setidaknya anda bisa melihat bahwa saya menggunakan 3 jenis tipe data di dalam tabel provinsi: INT untuk angka bulat, VARCHAR untuk teks, dan date untuk data tanggal.

Penamaan kolom sebaiknya tidak mengandung spasi dan dalam huruf kecil semua. Dalam tabel provinsi, saya memilih nama kolom: id_prov, bukan ID Prov. Ini untuk menghindari masalah saat membuat query maupun ketika memprosesnya di PHP. Nantinya, kita bisa dengan mudah mengubah judul kolom pada saat akan ditampilkan.

Untuk melihat daftar tabel dari suatu database, bisa menggunakan query: SHOW TABLES:

```
MariaDB [indonesia]> SHOW TABLES;
+-----+
| Tables_in_indonesia |
+-----+
| provinsi           |
+-----+
```

Hasilnya, hanya terdapat satu database di dalam database indonesia.

Untuk melihat struktur tabel, bisa menggunakan query DESCRIBE nama_tabel:

```
MariaDB [indonesia]> DESCRIBE provinsi;
+-----+-----+-----+-----+-----+-----+
| Field      | Type       | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id_prov    | int(11)    | YES  |     | NULL    |       |
| nama_prov  | varchar(100)| YES  |     | NULL    |       |
| ibu_kota   | varchar(100)| YES  |     | NULL    |       |
| populasi   | int(11)    | YES  |     | NULL    |       |
| tgl_diresmikan | date     | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
```

Disini terlihat semua tipe data dari setiap kolom yang ada di tabel provinsi. Untuk keterangan kolom seperti Null, Key, Default, dan Extra akan kita bahas lebih detail nantinya. Query DESCRIBE, juga bisa ditulis singkat, DESC:

```
MariaDB [indonesia]> DESC provinsi;
```

Untuk menghapus tabel, format dasarnya adalah:

```
DROP TABLE [ IF EXISTS] table_name;
```

Mari kita coba menghapus tabel provinsi:

```
MariaDB [indonesia]> DROP TABLE provinsi;
Query OK, 0 rows affected (0.36 sec)
```

```
MariaDB [indonesia]> SHOW TABLES;
Empty set (0.00 sec)
```

Sekarang, tabel provinsi sudah dihapus. Kembali saya ingatkan bahwa tidak ada tombol undo di MySQL. Ketika sebuah tabel dihapus, kita tidak bisa mengembalikannya kembali. Jadi harus extra hati-hati dengan perintah ini.

Query opsional [IF EXISTS] digunakan untuk menghilangkan pesan error apabila tabel yang akan dihapus memang sudah tidak ada.

```
MariaDB [indonesia]> DROP TABLE provinsi;
ERROR 1051 (42S02): Unknown table 'indonesia.provinsi'
```

```
MariaDB [indonesia]> DROP TABLE IF EXISTS provinsi;
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

6.5 Menambahkan Data ke Tabel

Setelah tabel selesai dibuat, saatnya menambahkan data menggunakan query `INSERT`. Query `INSERT` memiliki beragam format penulisan. Yang paling sederhana adalah:

```
INSERT INTO nama_tabel VALUES (nilai_kolom1, nilai_kolom2, ...);
```

Jika menggunakan format ini, data yang diinput harus sesuai dengan jumlah dan urutan kolom. Untuk tabel provinsi, terdapat 5 kolom sehingga harus diinput semuanya secara berurutan.

Sebelum praktek, pastikan tabel provinsi sudah dibuat, atau jalankan query berikut:

```
MariaDB [indonesia]> CREATE TABLE IF NOT EXISTS provinsi
(id_prov INT, nama_prov VARCHAR(100), ibu_kota VARCHAR(100),
populasi INT, tgl_diresmikan DATE);
```

Query OK, 0 rows affected, 1 warning (0.00 sec)

Tambahan perintah IF NOT EXISTS berguna untuk menghindari pesan error apabila tabel provinsi ternyata sudah ada.

Berikut query untuk menginput data Jakarta ke tabel provinsi:

```
MariaDB [indonesia]> INSERT INTO provinsi VALUES
(1, "Jakarta", "Jakarta", 10012271, "1961-08-28");
```

Tabel provinsi memiliki 5 kolom, oleh karena itu nilai dari VALUE juga ada 5 data. Untuk data teks dan tanggal, harus ditulis dalam tanda kutip, boleh menggunakan tanda kutip satu (') maupun tanda kutip dua (").

Khusus inputan tanggal, formatnya berbentuk tahun-bulan-tanggal. Tanggal 1961-08-28 berarti 28 Agustus 1961. Ini adalah format yang digunakan oleh tipe data date dalam MySQL.

Untuk memastikan data sudah diinput, mari lihat isi tabel menggunakan query SELECT:

```
MariaDB [indonesia]> SELECT * FROM provinsi;
+-----+-----+-----+-----+
| id_prov | nama_prov | ibu_kota | populasi | tgl_diresmikan |
+-----+-----+-----+-----+
| 1 | Jakarta | Jakarta | 10012271 | 1961-08-28 |
+-----+-----+-----+-----+
```

Perintah SELECT * FROM provinsi berfungsi untuk menampilkan seluruh data yang ada di dalam tabel provinsi. Cara penggunaan query SELECT akan kita bahas sesaat lagi.

Selain menginput satu data, query INSERT juga bisa digunakan untuk menginput banyak data sekaligus. Berikut contohnya:

```
MariaDB [indonesia]> INSERT INTO provinsi VALUES
(2, "Banten", "Serang", 11704877, "2000-10-4"),
(3, "Kalimantan Selatan", "Banjarmasin", 3922790, "1956-12-7");
```

Query OK, 2 rows affected (0.19 sec)
Records: 2 Duplicates: 0 Warnings: 0

Disini saya menginput dua buah data provinsi dalam satu kali pemanggilan query INSERT. Tanda koma dipakai sebagai pemisah data dari satu baris ke baris lainnya. Format dasar penulisan query diatas adalah sebagai berikut:

```
INSERT INTO nama_tabel VALUES
(nilai_kolom1, nilai_kolom2, ...),
(nilai_kolom1, nilai_kolom2, ...),
(nilai_kolom1, nilai_kolom2, ...), dst... ;
```

Berikut isi dari tabel provinsi setelah penambahan dua data baru:

```
MariaDB [indonesia]> SELECT * FROM provinsi;
+-----+-----+-----+-----+
| id_prov | nama_prov | ibu_kota | populasi | tgl_diresmikan |
+-----+-----+-----+-----+
| 1 | Jakarta | Jakarta | 10012271 | 1961-08-28 |
| 2 | Banten | Serang | 11704877 | 2000-10-04 |
| 3 | Kalimantan Selatan | Banjarmasin | 3922790 | 1956-12-07 |
+-----+-----+-----+-----+
```

Penulisan query INSERT sebelum ini mengharuskan data ditulis secara berurutan dan untuk seluruh kolom. Bagaimana jika kita ingin mengisi kolom tertentu saja dan tidak secara berurutan? Untuk keperluan seperti ini, bisa menggunakan format kedua query INSERT:

```
INSERT INTO nama_tabel (kolom1, kolom2, ...)
VALUES (nilai_kolom1, nilai_kolom2, ...);
```

Setelah penulisan nama_tabel, di dalam tanda kurung ditulis kolom apa saja yang ingin diinput. Nama kolom ini harus bersesuaian dengan nilai yang ada setelah perintah VALUES.

Sebagai contoh, saya bisa menginput data provinsi Kepulauan Riau dengan query berikut:

```
MariaDB [indonesia]> INSERT INTO provinsi
(nama_prov, tgl_diresmikan, ibu_kota, populasi, id_prov)
VALUES ("Kepulauan Riau", "2002-10-25", "Tanjung Pinang", 1917415, 4);
```

Perhatikan bahwa data yang diinput tidak diawali oleh id_prov, tapi nama_prov. Ini tidak masalah karena data tersebut akan sesuai dengan nilai inputan yang saya tulis setelah query VALUES.

Menggunakan penulisan query INSERT seperti ini, kita juga bisa menginput sebagian data saja:

```
MariaDB [indonesia]> INSERT INTO provinsi (nama_prov, ibu_kota, id_prov)
VALUES ("Papua Barat", "Manokwari", 5);
```

Disini saya menambah data provinsi Papua Barat hanya untuk 3 kolom saja. Bagaimana dengan 2 kolom lainnya? Mari kita lihat:

```
MariaDB [indonesia]> SELECT * FROM provinsi;
+-----+-----+-----+-----+
| id_prov | nama_prov      | ibu_kota     | populasi | tgl_diresmikan |
+-----+-----+-----+-----+
| 1 | Jakarta        | Jakarta      | 10012271  | 1961-08-28   |
| 2 | Banten          | Serang       | 11704877  | 2000-10-04   |
| 3 | Kalimantan Selatan | Banjarmasin | 3922790   | 1956-12-07   |
| 4 | Kepulauan Riau    | Tanjung Pinang | 1917415  | 2002-10-25   |
| 5 | Papua Barat     | Manokwari    | NULL       | NULL        |
+-----+-----+-----+-----+
```

Secara default, data yang tidak diisi akan ditampilkan sebagai NULL. NULL adalah tipe data khusus yang menyatakan ‘tidak ada data’. Nantinya kita bisa mengganti nilai NULL ini dengan data lain, atau memaksa MySQL untuk menampilkan pesan error jika sebuah kolom tidak diisi nilainya (tidak boleh NULL).

6.6 Mengubah Data Tabel

Ada saatnya kita ingin mengkoreksi data yang salah atau ingin meng-update data lama. Untuk keperluan mengubah data yang sudah ada di tabel, tersedia query UPDATE.

Berikut format dasar query UPDATE MySQL:

```
UPDATE nama_tabel SET nama_kolom = data_baru WHERE kondisi
```

Kunci dari query UPDATE berada di bagian kondisi. Kondisi inilah yang akan membatasi data mana yang ingin diubah.

Sebagai contoh, saya ingin mengubah nama provinsi Jakarta menjadi DKI Jakarta. Artinya saya akan mengubah data pada kolom nama_prov menjadi DKI Jakarta dimana kolom tersebut memiliki id_prov = 1. Berikut querynya:

```
MariaDB [indonesia]> UPDATE provinsi SET nama_prov = "DKI Jakarta"
WHERE id_prov = 1;
```

```
Query OK, 1 row affected (0.11 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
MariaDB [indonesia]> SELECT * FROM provinsi;
+-----+-----+-----+-----+
| id_prov | nama_prov      | ibu_kota     | populasi | tgl_diresmikan |
+-----+-----+-----+-----+
| 1 | DKI Jakarta    | Jakarta      | 10012271  | 1961-08-28   |
| 2 | Banten          | Serang       | 11704877  | 2000-10-04   |
| 3 | Kalimantan Selatan | Banjarmasin | 3922790   | 1956-12-07   |
+-----+-----+-----+-----+
```

	4 Kepulauan Riau	Tanjung Pinang	1917415	2002-10-25	
	5 Papua Barat	Manokwari	NULL	NULL	

Lupa menulis kondisi WHERE untuk query UPDATE, bisa berakibat fatal. Karena seluruh kolom akan berubah.

6.7 Menghapus Data Tabel

Untuk menghapus data yang ada di dalam sebuah tabel, tersedia query DELETE. Berikut format dasarnya:

```
DELETE FROM nama_tabel WHERE kondisi
```

Sama seperti query UPDATE, kunci proses penghapusan ada di kondisi.

Sebagai contoh, saya ingin menghapus baris tabel yang memiliki kondisi ibu_kota = Banjarmasin. Berikut perintahnya:

```
MariaDB [indonesia]> DELETE FROM provinsi WHERE ibu_kota = "Banjarmasin";
Query OK, 1 row affected (0.05 sec)
```

```
MariaDB [indonesia]> SELECT * FROM provinsi;
+-----+-----+-----+-----+
| id_prov | nama_prov | ibu_kota | populasi | tgl_diresmikan |
+-----+-----+-----+-----+
| 1 | DKI Jakarta | Jakarta | 10012271 | 1961-08-28 |
| 2 | Banten | Serang | 11704877 | 2000-10-04 |
| 4 | Kepulauan Riau | Tanjung Pinang | 1917415 | 2002-10-25 |
| 5 | Papua Barat | Manokwari | NULL | NULL |
+-----+-----+-----+-----+
```

Terlihat, data provinsi Kalimantan Selatan dengan ibu_kota = "Banjarmasin" sudah terhapus dari tabel provinsi.

6.8 Menampilkan Data Table

Dari sekian banyak query di dalam MySQL, perintah untuk menampilkan data adalah yang paling sering dipakai, yakni query SELECT. Proses pembuatan tabel biasanya hanya perlu sekali di awal, akan tetapi proses menampilkan data terus menerus dipanggil selama aplikasi digunakan.

Untuk menampilkan data, setidaknya kita butuh 3 hal:

- Apa saja kolom yang ingin ditampilkan.
- Nama tabel yang akan ditampilkan.
- Kondisi untuk menampilkan data.

Query SELECT perlu ketiga hal ini. Berikut adalah format dasar penulisan query SELECT:

```
SELECT kolom1, kolom2 FROM nama_tabel WHERE kondisi;
```

Khusus untuk kondisi, apabila tidak ditulis maka seluruh data tabel akan ditampilkan.

Sebagai contoh, untuk menampilkan kolom `nama_prov` dan `ibu_kota` dari tabel `provinsi`, bisa menggunakan query berikut:

```
MariaDB [indonesia]> SELECT nama_prov, ibu_kota FROM provinsi;
+-----+-----+
| nama_prov | ibu_kota |
+-----+-----+
| DKI Jakarta | Jakarta |
| Banten | Serang |
| Kepulauan Riau | Tanjung Pinang |
| Papua Barat | Manokwari |
+-----+
```

Jika ingin menampilkan seluruh kolom, bagian nama kolom bisa diganti dengan karakter bintang (*):

```
MariaDB [indonesia]> SELECT * FROM provinsi;
+-----+-----+-----+-----+-----+
| id_prov | nama_prov | ibu_kota | populasi | tgl_diresmikan |
+-----+-----+-----+-----+-----+
| 1 | DKI Jakarta | Jakarta | 10012271 | 1961-08-28 |
| 2 | Banten | Serang | 11704877 | 2000-10-04 |
| 4 | Kepulauan Riau | Tanjung Pinang | 1917415 | 2002-10-25 |
| 5 | Papua Barat | Manokwari | NULL | NULL |
+-----+-----+-----+-----+
```

Untuk membatasi baris mana saja yang ingin ditampilkan, bisa menulis bagian kondisi. Sebagai contoh, saya ingin menampilkan data untuk `id_prov` 4 dan 5. Querynya adalah:

```
MariaDB [indonesia]> SELECT * FROM provinsi WHERE id_prov=1 OR id_prov=2;
+-----+-----+-----+-----+
| id_prov | nama_prov | ibu_kota | populasi | tgl_diresmikan |
+-----+-----+-----+-----+
| 1 | DKI Jakarta | Jakarta | 10012271 | 1961-08-28 |
| 2 | Banten | Serang | 11704877 | 2000-10-04 |
+-----+-----+-----+-----+
```

Disini saya menggunakan operator `OR` untuk mengambil data dengan `id_prov` 4 dan 5. Selain itu masih banyak operator lain yang bisa digunakan untuk membuat kondisi. Kita akan membahasnya pada bab khusus nanti.

Query `SELECT` juga menyediakan perintah opsional `ORDER BY` untuk mengurutkan data. Format dasar penulisannya adalah sebagai berikut:

```
SELECT kolom1, kolom2 FROM nama_tabel
WHERE kondisi ORDER BY nama_kolom_urut [ASC atau DESC]
```

Bagian `nama_kolom_urut` adalah kolom yang menjadi penentu urutan. Pilihan ini dapat diatur dengan penambahan instruksi `ASC` (singkatan dari *ascending*) untuk pengurutan dari kecil ke besar, dan `DESC` (singkatan dari *descending*) untuk urutan besar ke kecil. Jika tidak ditulis, secara default `ORDER BY` akan menggunakan `ASC`.

Sebagai contoh, saya ingin menampilkan data seluruh provinsi yang diurutkan berdasarkan kolom `nama_prov`:

```
MariaDB [indonesia]> SELECT * FROM provinsi ORDER BY nama_prov;
+-----+-----+-----+-----+
| id_prov | nama_prov      | ibu_kota      | populasi | tgl_diresmikan |
+-----+-----+-----+-----+
|      2 | Banten          | Serang         | 11704877 | 2000-10-04    |
|      1 | DKI Jakarta     | Jakarta        | 10012271 | 1961-08-28    |
|      4 | Kepulauan Riau | Tanjung Pinang | 1917415  | 2002-10-25    |
|      5 | Papua Barat    | Manokwari     |      NULL |      NULL     |
+-----+-----+-----+-----+
```

Bagaimana jika diurutkan berdasarkan populasi mulai dari yang terbesar? tambahkan perintah `DESC`:

```
MariaDB [indonesia]> SELECT * FROM provinsi ORDER BY populasi DESC;
+-----+-----+-----+-----+
| id_prov | nama_prov      | ibu_kota      | populasi | tgl_diresmikan |
+-----+-----+-----+-----+
|      2 | Banten          | Serang         | 11704877 | 2000-10-04    |
|      1 | DKI Jakarta     | Jakarta        | 10012271 | 1961-08-28    |
|      4 | Kepulauan Riau | Tanjung Pinang | 1917415  | 2002-10-25    |
|      5 | Papua Barat    | Manokwari     |      NULL |      NULL     |
+-----+-----+-----+-----+
```

Query `SELECT` juga memiliki instruksi tambahan untuk membatasi berapa jumlah baris yang akan ditampilkan, yakni dengan perintah `LIMIT`.

Berikut format dasarnya:

```
SELECT kolom1, kolom2 FROM nama_tabel
WHERE kondisi LIMIT baris_awal, jumlah_baris
```

Dimana `baris_awal` adalah awal baris yang ingin ditampilkan, dan `jumlah_baris` adalah jumlah baris yang diurutkan dari `baris_awal`. Nomor baris pada MySQL diawali dengan nomor 0.

Misalkan saya ingin menampilkan data 2 provinsi paling atas yang dirutukan berdasarkan `nama_prov`, maka querynya adalah:

```
MariaDB [indonesia]> SELECT * FROM provinsi ORDER BY nama_prov LIMIT 0,2;
+-----+-----+-----+-----+
| id_prov | nama_prov | ibu_kota | populasi | tgl_diresmikan |
+-----+-----+-----+-----+
|      2 | Banten     | Serang    | 11704877 | 2000-10-04   |
|      1 | DKI Jakarta | Jakarta   | 10012271 | 1961-08-28   |
+-----+-----+-----+-----+
```

Jika kita hanya memberikan 1 angka di belakang instruksi `LIMIT`, maka MySQL menganggap bahwa angka dihitung dari baris teratas (baris ke 0). Sehingga tampilan sebelumnya juga bisa didapat dengan perintah berikut:

```
MariaDB [indonesia]> SELECT * FROM provinsi ORDER BY nama_prov LIMIT 2;
```

Dalam bab ini kita telah melihat cara penulisan query MySQL serta beberapa perintah dasar seperti `CREATE`, `UPDATE`, `DELETE` dan `SELECT`. Apa yang saya tulis disini baru perkenalan dasar, sekedar pemanasan sebelum kita membahasnya dengan lebih detail.

Selanjutnya, kita akan pelajari query tentang **Database, Character set dan Collation**.

7. Database, Character set dan Collation

Dalam bab sebelumnya, kita telah membahas beberapa query terkait database:

- CREATE DATABASE / CREATE SCHEMA
- USE DATABASE
- SHOW DATABASES
- DROP DATABASE / DROP SCHEMA

Ini semua sudah mencakup perintah umum pembuatan dan penghapusan database. Schema merupakan istilah lain dari database sehingga bisa digunakan sebagai perintah alternatif.

Query tambahan yang belum dibahas untuk pembuatan database adalah mengenai **character set** dan **collation**. Sebelum masuk ke perintah querinya, kita akan membahas dulu apa pengertian kedua istilah ini.

7.1 Pengertian Character set dan Collation

Character set (biasa disingkat sebagai **charset**) adalah kumpulan karakter yang bisa dipahami oleh komputer. Sedangkan **collation** adalah aturan pengurutan data berdasarkan **charset**.

Charset mengatur bagaimana MySQL menyimpan setiap karakter ke dalam tabel. Jika anda sudah pernah mempelajari HTML (dari buku HTML Uncover misalnya), tentu pernah melihat tag `<meta charset="UTF-8">`. **UTF-8** adalah salah satu contoh dari character set.

Materi tentang charset dan collation ini sebenarnya baru diperlukan jika anda berencana menyimpan huruf non-latin ke dalam MySQL, seperti karakter jepang, korea, china atau arab. Untuk alphabet latin yang kita gunakan sehari-hari, charset dan collation tidak terlalu berpengaruh.

Penjelasan tentang character set sedikit teknis, tapi akan saya coba jabarkan secara perlahan.

Pernah mengenal *sandi morse*? Sandi atau kode morse digunakan ketika teknologi komunikasi baru mulai muncul. Pesan dikirim menggunakan media kabel atau radio seperti saat ini, namun perangkat penerima dan pengirim hanya bisa menerima 2 sinyal: **sinyal pendek** dan **sinyal panjang**. Teknologi saat itu belum bisa mengirim suara layaknya radio modern.

Bagaimana cara bertukar pesan dengan kondisi seperti ini? Ingat, yang bisa dikirim hanya sinyal pendek dan sinyal panjang: yakni bunyi *tut*, dan *tuuut*.

Solusinya, kita bisa membuat sebuah daftar kode karakter. Kode dengan pola kombinasi tertentu akan mewakili satu huruf.

Misalnya untuk huruf A, sinyalnya adalah *pendek, pendek, pendek*. Untuk huruf B sinyalnya *pendek, pendek, panjang*, dst. Dengan kombinasi 3 sinal untuk setiap huruf, sudah dapat menampung $2^3 = 8$ karakter.

Mari kita anggap sinal pendek dengan angka 0, dan sinal panjang dengan angka 1. Daftar kodenya adalah sebagai berikut:

- A: 000
- B: 001
- C: 010
- D: 011
- E: 100
- F: 101
- G: 110
- H: 111

Daftar diatas sudah bisa disebut sebagai sebuah character set sederhana. Saya sebut saja sebagai `sandi_8_karakter`.

Huruf abjad latin terdiri dari 26 karakter, untuk bisa membuat kode yang bisa menampung seluruh karakter alphabet, setidaknya harus disimpan ke dalam 5 digit kode, dimana $2^5 = 32$. Artinya jika kita membuat daftar dimana setiap karakter diwakili oleh 5 digit, tersedia ruang untuk 32 jenis karakter.

Berikut daftar karakter set-nya:

- A: 00000
- B: 00001
- C: 00010
- ...
- Z: 11010
- ...

Huruf abjad berjumlah 26 karakter. Jika menggunakan 5 digit masih ada sisa $32 - 26 = 6$ karakter lagi. Ini bisa dipakai untuk karakter khusus seperti titik, koma, tanda seru, tanda tanya, titik dua dan tanda kutip.

Daftar ini pun sudah bisa menjadi sebuah character set yang saya sebut sebagai `sandi_32_karakter`.

Namun `sandi_32_karakter` juga masih belum bisa menampung angka 0 - 9 (karena sudah tidak muat di 5 digit kode). Kita harus naikkan lagi menjadi 6 digit kode yang sanggup menampung $2^6 = 64$ karakter. Ini saya sebut sebagai `sandi_64_karakter`.

Sampai disini, kita sudah memiliki 3 character set: `sandi_8_karakter`, `sandi_32_karakter` dan `sandi_64_karakter`. Setiap character set bisa digunakan tergantung keperluan. Jika saya hanya butuh mengirim sinal huruf, maka `sandi_32_karakter` sudah mencukupi. Jika butuh mengirim huruf dan angka, maka harus menggunakan `sandi_64_karakter`.

Di sistem komputer modern, cara kerjanya sangat mirip seperti contoh diatas. Sinyal panjang dan pendek dari sandi morse digantikan dengan digit 1 dan 0 dari sistem bilangan **biner**. Inilah basis dari cara komputer bekerja: ada sinyal listrik (1), dan tidak ada sinyal listrik (0).

Angka 0 dan angka 1 ini dikenal dengan istilah **bit**. Agar terdengar lebih keren, `sandi_64_-karakter` yang saya rancang sebelumnya bisa juga disebut sebagai `sandi_6_bit`. Karena setiap karakter butuh 6 bit.

Jadi, komputer tidak paham apa itu huruf ‘A’ atau angka 9. Ia hanya bisa memproses suatu data dalam kondisi biner (kondisi 0 atau 1). Agar bisa menampilkan huruf ‘A’, kita harus memberikan instruksi kepada komputer untuk menterjemahkan menggunakan character set tertentu. Tanpa character set, huruf tidak bisa ditampilkan.



Sandi morse sendiri sebenarnya tidak menggunakan panjang kode yang tetap, tapi berlainan untuk setiap karakter. Huruf ‘E’ sebagai huruf yang paling sering muncul di text bahasa inggris hanya butuh 1 *sinyal pendek*, sedangkan huruf ‘C’ dikodekan sebagai gabungan dari 4 sinyal: *panjang, pendek, panjang, dan pendek*. Sumber: [Morse code¹](#).

7.2 ASCII Character Set

Setelah era sandi morse, perkembangan teknologi telegram butuh karakter set yang lebih baik.

Pada tahun 1960an, badan standarisasi Amerika Serikat, yakni *American Standards Association (ASA)* atau sekarang berubah nama menjadi **ANSI** (*American National Standards Institute*) mulai merancang character set baru sebagai pengganti sandi morse.

Character set ini diberi nama **ASCII** dan dirilis pada tahun 1963. ASCII sendiri merupakan singkatan dari **American Standard Code for Information Interchange**.

ASCII adalah character set 7 bit, yang artinya setiap karakter di dalam ASCII butuh 7 digit angka biner. Sebagai contoh, huruf ‘A’ dalam character ASCII memiliki kode biner 1000001. Dengan 7 bit, total karakter yang bisa ditampung sebanyak $2^7 = 128$.

Pada era awal kemunculan komputer, ASCII ini adalah character set yang paling banyak dipakai.

Prinsip kerja character set ASCII

Prinsip kerja character set ASCII mirip dengan ilustrasi sandi morse yang kita pelajari sebelumnya. Hanya saja terdapat urutan khusus:

- 32 karakter awal dengan kode biner 0000000 - 0011111 berisi karakter “control”, yang dikenal dengan istilah **ASCII control characters**. Karakter khusus ini bukan untuk ditampilkan tapi mengontrol bagaimana alur teks disusun, contohnya “line feed” sebagai penanda baris baru dan karakter “tab”.

¹https://en.wikipedia.org/wiki/Morse_code

- 96 karakter berikutnya dengan kode biner 00100000 - 0011111 adalah karakter yang bisa terlihat, dikenal sebagai **ASCII printable characters**. Bagian ini terdiri dari seluruh huruf alphabet (A-Z dan a-z), angka (0 - 9), serta berbagai karakter penanda seperti \$, +, = dan &. Hampir seluruh karakter ini ada di keyboard komputer yang kita gunakan sekarang.

Agar lebih memahami tentang character set, kita akan latihan menerjemahkan kode biner ASCII ke dalam bentuk teks.

Perhatikan rangkaian kode biner berikut:

0100110101100101011100100110010001100101011000010010001

Digit kode biner diatas merupakan “makanan” sehari-hari yang dihadapi oleh komputer. Namun komputer sendiri belum tentu paham cara membacanya. Kenapa? Ia butuh petunjuk lain, yakni character set apa yang dipakai?

Setiap character set punya aturan sendiri-sendiri. Menerjemahkan kode biner menggunakan character set yang salah, hasilnya juga tidak akan sesuai. Kode biner diatas saya buat dari kode ASCII 7 bit, karena itu juga harus menggunakan character set yang sama untuk menterjemahkannya.

Pertama, pisahkan rangkaian bit biner setiap 7 bit:

1001101 1100101 1110010 1100100 1100101 1101011 1100001 0100001

Selanjutnya, cari di tabel character set ASCII digit-digit yang bersesuaian. Tabel berikut berisi 96 **ASCII printable characters**:

Desimal	Binary	Simbol	Desimal	Binary	Simbol	Desimal	Binary	Simbol
32	0100000	(spasi)	64	1000000	@	96	1100000	'
33	0100001	!	65	1000001	A	97	1100001	a
34	0100010	"	66	1000010	B	98	1100010	b
35	0100011	#	67	1000011	C	99	1100011	c
36	0100100	\$	68	1000100	D	100	1100100	d
37	0100101	%	69	1000101	E	101	1100101	e
38	0100110	&	70	1000110	F	102	1100110	f
39	0100111	,	71	1000111	G	103	1100111	g
40	0101000	(72	1001000	H	104	1101000	h
41	0101001)	73	1001001	I	105	1101001	i
42	0101010	*	74	1001010	J	106	1101010	j
43	0101011	+	75	1001011	K	107	1101011	k
44	0101100	,	76	1001100	L	108	1101100	l
45	0101101	-	77	1001101	M	109	1101101	m
46	0101110	.	78	1001110	N	110	1101110	n
47	0101111	/	79	1001111	O	111	1101111	o
48	0110000	0	80	1010000	P	112	1110000	p
49	0110001	1	81	1010001	Q	113	1110001	q
50	0110010	2	82	1010010	R	114	1110010	r

Desimal	Binary	Simbol	Desimal	Binary	Simbol	Desimal	Binary	Simbol
51	0110011	3	83	1010011	S	115	1110011	s
52	0110100	4	84	1010100	T	116	1110100	t
53	0110101	5	85	1010101	U	117	1110101	u
54	0110110	6	86	1010110	V	118	1110110	v
55	0110111	7	87	1010111	W	119	1110111	w
56	0111000	8	88	1011000	X	120	1111000	x
57	0111001	9	89	1011001	Y	121	1111001	y
58	0111010	:	90	1011010	Z	122	1111010	z
59	0111011	;	91	1011011	[123	1111011	{
60	0111100	<	92	1011100	\	124	1111100	
61	0111101	=	93	1011101]	125	1111101	}
62	0111110	>	94	1011110	^	126	1111110	~
63	0111111	?	95	1011111	_	127	1111111	(delete)

Angka di kolom **Desimal** berisi nomor urut dalam bilangan desimal (basis 10). Sedangkan kolom **Biner** berisi nomor urut karakter dalam bilangan biner (basis 2).

Sekarang, kita tinggal cari setiap 7 bit dari tabel ASCII diatas. Hasilnya adalah sebagai berikut:

Kode biner	Karakter ASCII
1001101	M
1100101	e
1110010	r
1100100	d
1100101	e
1101011	k
1100001	a
0100001	!

Yup, string biner sebelumnya dapat dibaca sebagai “Merdeka!” menggunakan ASCII 7 bit. Dalam proses penerjemahan ini, aspek yang terpenting adalah kita harus mengetahui character set apa yang mesti dipakai. Salah menggunakan character set, hurufnya juga tidak akan terbaca.

7.3 Extended-ASCII Character Set

128 character set yang terdapat di dalam ASCII sebenarnya sudah mencukupi untuk kebutuhan negara yang menggunakan aksara latin seperti Amerika Serikat. Bahkan pada praktiknya tidak semua perangkat menggunakan ke-128 karakter ASCII standar. Kadang ada yang hanya 95 dan ada juga yang hanya 64 karakter.

Keterbatasan karakter yang ada diakali dengan cara yang unik. Karakter ASCII tidak memiliki tanda “÷” sehingga mayoritas bahasa pemrograman menggantinya dengan tanda “/” untuk operator pembagian.

Dalam bahasa C, digunakan tanda “%” and “%>” sebagai pengganti kurung kurawal “{” dan “}” yang tidak selalu ada di keyboard komputer.

Di beberapa negara, karakter set ASCII ada yang diganti menjadi karakter lain, misalnya negara Inggris mengganti tanda “#” menjadi “£”, di negara Jepang tanda “\” diganti menjadi “¥”.

Masalah mulai muncul ketika ASCII digunakan pada negara yang butuh karakter non latin, seperti China, Korea, Jepang atau Arab.

Keadaan ini berlangsung cukup lama. Usaha untuk membuat character set baru yang bisa menampung lebih banyak karakter agak susah dilakukan karena terbentur masalah teknologi.

Ketika perkembangan komputer mulai beralih ke 8-bit (1 byte), terdapat peluang untuk mengembangkan character set ASCII. Dengan 8 digit biner akan sanggup menampung $2^8 = 256$ karakter. Artinya terdapat ruang untuk 128 karakter baru.

Character ASCII dengan 8 bit ini dikenal dengan sebutan **Extended-ASCII Character Set²**. Urutan 1 - 128 tetap ditempati karakter ASCII lama, dan karakter baru akan menempati urutan 129 - 256.

Masalahnya, apa saja karakter yang harus ditambahkan?

Disinilah awal kemunculan bermacam-macam character set, karena tentu saja setiap negara butuh karakter sendiri-sendiri. Untuk komputer di Jepang, mereka butuh menginput aksara jepang.

Agar bisa mengakomodasi kebutuhan ini, ISO (International Organization for Standardization) selaku pembuat standar teknologi menetapkan 16 jenis **Extended-ASCII character set**, atau dikenal juga sebagai **ISO/IEC 8859-1³**:

- Latin-1 Western European
- Latin-2 Central European
- Latin-3 South European
- Latin-4 North European
- Latin/Cyrillic
- Latin/Arabic
- Latin/Greek
- Latin/Hebrew
- Latin-5 Turkish
- Latin-6 Nordic
- Latin/Thai
- Latin/Devanagari
- Latin-7 Baltic Rim
- Latin-8 Celtic
- Latin-9
- Latin-10 South-Eastern European

Nama dari setiap character set bersesuaian dengan tujuannya. Character set *Latin/Arabic* ditujukan untuk aksara arab. Sedangkan *Latin-5 Turkish* dibuat untuk aksara turki.

²https://en.wikipedia.org/wiki/Extended_ASCII

³https://en.wikipedia.org/wiki/ISO/IEC_8859-1

Character set latin-1 Western European sebenarnya ditujukan untuk aksara eropa barat. Nomor urut 129 - 256 berisi berbagai karakter dialek seperti huruf â, å, í, î, ú, õ, juga termasuk karakter matematis seperti %, ±, dan μ.

Selain character set diatas, juga muncul berbagai character set non-standar. IBM membuat EBCDIC yang digunakan pada perangkat buatannya, Microsoft membuat Windows-1252, yang semakin menambah beragamnya jenis character set.

7.4 Unicode Character Set

Sebagaimana yang telah kita praktekkan, agar bisa saling berkomunikasi, 2 buah komputer harus sepaham terkait karakter set yang digunakan. Kalau tidak, kode tersebut tidak bisa terbaca.

Pernah melihat karakter kotak-kotak tampil di halaman web? Itu merupakan salah satu indikasi dimana web browser mencoba menerjemahkan teks dengan character set yang tidak sesuai. Masalah ini diperparah dengan banyaknya jenis character set yang beredar.

Di era sebelum internet, hal ini tidak menjadi masalah. Setiap komputer hanya perlu character set sesuai dengan daerah masing-masing. Jika ingin mengirim dokumen antar negara, dokumen itu di print terlebih dahulu, kemudian dikirim menggunakan fax.

Akan tetapi untuk era internet, hal tersebut menjadi masalah besar. Yang dikirim bukan lagi fisik dokumen, tetapi angka biner (sinyal data).

Ketika seseorang membuat dokumen di jepang, ia menggunakan character set [JIS encoding⁴](#). Agar dokumen yang sama bisa terbaca di Amerika, komputer tersebut harus bisa memahami JIS encoding. Selain itu, bagaimana jika di dalam sebuah dokumen terdapat aksara jepang, korea dan thailand sekaligus?

Solusi dari masalah ini, harus ada sebuah character set “universal” yang sanggup menampung mayoritas aksara yang ada di dunia. Tentu saja 256 karakter yang ada di Extended-ASCII tidak bisa menampung semuanya.

Unicode character set muncul sebagai solusi. Character set ini sanggup menampung puluhan ribu karakter. Dengan kata lain **Unicode** mencoba menggabungkan seluruh character set menjadi sebuah character set “universal”.

Daftar lengkap dari karakter Unicode ini bisa dilihat di [unicode-table.com⁵](#). Karakter yang tersedia sangat beragam, mulai dari huruf latin, jepang, korea, arab, hingga huruf jawa kuno dan emoji (icon).

UTF-8

Penggunaan **Unicode** character set tampak sebagai solusi ideal, akan tetapi memiliki masalah di sisi kinerja dan efisiensi.

Berapa bit yang dibutuhkan untuk setiap karakter? Agar bisa menampung setidaknya hingga 65.536 karakter, kita butuh character set dengan 16-bit ($2^{16} = 65.536$).

⁴https://en.wikipedia.org/wiki/JIS_encoding

⁵unicode-table.com

Oleh karena itu, dalam Unicode 16 bit, huruf ‘A’ perlu ruang penyimpanan 2 kali lipat lebih banyak dibandingkan huruf ‘A’ dari Extended ASCII, yakni 8 bit vs 16 bit (1 byte vs 2 byte). Lebih jauh lagi, juga dikembangkan Unicode 32 bit. Artinya, 1 huruf ‘A’ butuh dikodekan oleh 32 buah digit biner.

Ruang memory menjadi masalah disini. Jika kita beralih menggunakan Unicode, setiap dokumen akan berukuran 2 sampai 4 kali lipat ukuran asalnya.

Solusi dari masalah ini adalah merancang character set Unicode yang ‘fleksibel’. Character set ini diberi nama **UTF-8** ([Unicode Transformation Format 8 bit⁶](#)).

UTF-8 dirancang oleh **Ken Thompson** and **Rob Pike**. Ken Thompson sendiri merupakan ahli komputer yang mengembangkan bahasa pemrograman B (nantinya berkembang menjadi bahasa pemrograman C dan C++).

Dengan pengkodean khusus. UTF-8 menempatkan karakter dengan panjang bit yang berbeda-beda. Karakter alphabet latin (A-Z dan a-z) sebagai karakter yang paling sering digunakan hanya butuh 8 bit (1 byte), yakni sama seperti **Extended ASCII**.

Agar bisa kompatibel dengan **ASCII** dan **Extended ASCII**, 256 digit karakter awal dari UTF-8 sama persis dengan **Extended ASCII**.

Setelah 8 bit “habis”, karakter berikutnya akan menggunakan 2 byte (16 bit). Disinilah aksara arab, jepang, china, korea berada.

Ketika 2 byte “habis”, karakter berikutnya akan menggunakan 3 byte (24 bit). Disini disimpan karakter china kuno, dan berbagai karakter lain yang relatif jarang dipakai.

Terakhir terdapat karakter yang menggunakan 4 byte (32 bit) yang berisi seluruh karakter unicode lain dan di cadangkan untuk masa depan.

Berikut contoh kode biner untuk UTF-8:

- \$ = 00100100 (8 bit, dan merupakan bagian dari ASCII)
- € = 11000010 10100010 (16 bit)
- € = 11100010 10000010 10101100 (24 bit)

UTF-8 memiliki pengkodean khusus untuk mencegah bentrok antar kode biner.

Dengan membuat character set seperti ini, UTF-8 tetap efisien. Jika kita menyimpan huruf dalam aksara alphabet latin (seperti mayoritas teks dalam bahasa indonesia), setiap huruf hanya butuh 8 bit atau sama jika teks tersebut dibuat menggunakan **Extended ASCII**.

Saat ini UTF-8 sudah umum di gunakan terutama di bidang web programming. Pada juni 2017, sekitar [89% dari halaman web⁷](#) sudah menggunakan UTF-8 sebagai character set standar.

⁶<https://en.wikipedia.org/wiki/UTF-8>

⁷<https://en.wikipedia.org/wiki/UTF-8>

UTF-16 dan UTF-32

Unicode tidak hanya terdiri dari UTF-8 saja, tapi ada variasi character set lain seperti **UTF-16** dan **UTF-32**.

UTF-16 mirip dengan UTF-8, tetapi menggunakan basis 16bit. Jika karakter di 16 bit sudah habis, akan menggunakan 32 bit.

Apabila anda sering menulis karakter non-latin, bisa jadi UTF-16 akan lebih hemat tempat. Karena terdapat kemungkinan huruf tersebut disimpan sebagai 24 bit di UTF-8, tapi cukup menggunakan 16 bit di UTF-16.

UTF-32 menggunakan pengkodean 32 bit yang tetap untuk seluruh karakter (fixed). Artinya 1 huruf "A" butuh ruang memory 4 byte. Karena terlalu boros, UTF-32 tidak banyak dipakai.

Yang perlu diingat, baik **UTF-8**, **UTF-16** dan **UTF-32** sama-sama termasuk **Unicode** dan memiliki jumlah karakter yang sama. Perbedaannya hanya dari cara penyimpanan karakter.

7.5 Character Set di dalam MySQL

Dari pemaparan tentang character set, UTF-8 tampak sebagai character set yang ideal. Ini benar, terutama jika anda sering menginput kode teks dalam aksara non-latin. Namun sebenarnya character set **Extended-ASCII Latin-1** sudah memenuhi keperluan kita di Indonesia, karena sebagian besar text hanya akan menggunakan karakter alphabet latin.

MySQL mendukung hampir semua character set yang ada. Untuk melihat apa saja character set yang tersedia, bisa menjalankan query SHOW CHARACTER SET:

```
SHOW CHARACTER SET;
+-----+-----+-----+-----+
| Charset | Description          | Default collation | Maxlen |
+-----+-----+-----+-----+
| big5    | Big5 Traditional Chinese | big5_chinese_ci    |      2  |
| dec8    | DEC West European       | dec8_swedish_ci    |      1  |
| cp850   | DOS West European        | cp850_general_ci  |      1  |
| hp8     | HP West European         | hp8_english_ci    |      1  |
| koi8r   | KOI8-R Relcom Russian   | koi8r_general_ci  |      1  |
| latin1  | cp1252 West European    | latin1_swedish_ci |      1  |
| latin2  | ISO 8859-2 Central European | latin2_general_ci |      1  |
| swe7    | 7bit Swedish             | swe7_swedish_ci   |      1  |
| ascii   | US ASCII                 | ascii_general_ci  |      1  |
| ujis    | EUC-JP Japanese          | ujis_japanese_ci |      3  |
...
| geostd8 | GEOSTD8 Georgian         | geostd8_general_ci |      1  |
| cp932   | SJIS for Windows Japanese | cp932_japanese_ci |      2  |
| eucjpms | UJIS for Windows Japanese | eucjpms_japanese_ci |      3  |
+-----+-----+-----+-----+
40 rows in set (0.00 sec)
```

Terdapat 40 charset yang bisa dipilih. Di kolom `Description` kita bisa membaca penjelasannya. Misalnya `big5` charset yang ditujukan untuk karakter **Traditional Chinese**.

Di kolom paling kanan, `Maxlen` menunjukkan maksimal byte dari setiap charset. `latin1` hanya butuh 1 byte, sedangkan `utf32` butuh 4 byte.

Kolom ketiga dari tabel charset diatas berisi `Default collation`. Yakni sebuah aturan pengurutan karakter untuk suatu charset. Pengurutan ini perlu karena ada perbedaan di beberapa bahasa.

Dalam bahasa eropa seperti rusia, jerman dan yunani, huruf a bisa dibuat dari beberapa karakter: à, á, â, ã, ä dan å. Inilah yang diatur dari **collation**, yakni huruf a mana yang harus di dahulukan. Untuk bahasa indonesia yang terdiri dari karakter latin standar (alphabet biasa), tidak banyak perbedaan dari pilihan collation ini, bahkan tidak berpengaruh sama sekali.

Untuk melihat apa saja collation yang di dukung oleh MySQL bisa menjalankan query `SHOW COLLATION`:

```
SHOW COLLATION;
```

Collation	Charset	Id	Default	Compiled	Sortlen
big5_chinese_ci	big5	1	Yes	Yes	1
big5_bin	big5	84		Yes	1
dec8_swedish_ci	dec8	3	Yes	Yes	1
dec8_bin	dec8	69		Yes	1
koi8r_general_ci	koi8r	7	Yes	Yes	1
koi8r_bin	koi8r	74		Yes	1
latin1_german1_ci	latin1	5		Yes	1
latin1_swedish_ci	latin1	8	Yes	Yes	1
latin1_danish_ci	latin1	15		Yes	1
latin1_german2_ci	latin1	31		Yes	2
latin1_bin	latin1	47		Yes	1
latin1_general_ci	latin1	48		Yes	1
latin1_general_cs	latin1	49		Yes	1
latin1_spanish_ci	latin1	94		Yes	1
latin2_czech_cs	latin2	2		Yes	4
latin2_general_ci	latin2	9	Yes	Yes	1
latin2_hungarian_ci	latin2	21		Yes	1
latin2_croatian_ci	latin2	27		Yes	1
latin2_bin	latin2	77		Yes	1
swe7_swedish_ci	swe7	10	Yes	Yes	1
swe7_bin	swe7	82		Yes	1
ascii_general_ci	ascii	11	Yes	Yes	1
ascii_bin	ascii	65		Yes	1
...					
cp932_japanese_ci	cp932	95	Yes	Yes	1
cp932_bin	cp932	96		Yes	1
eucjpms_japanese_ci	eucjpms	97	Yes	Yes	1

```
+-----+-----+-----+-----+-----+
| eucjpms_bin | eucjpms | 98 | Yes | 1 |
+-----+-----+-----+-----+-----+
234 rows in set (0.02 sec)
```

Terdapat 234 pilihan collation!

Akan tetapi, kita tidak bisa sembarang memilih collation. Sebuah collation hanya bisa digunakan untuk character set tertentu saja. Misalnya collation `big5_chinese_ci` dan `big5_bin` hanya bisa untuk character set `big5`. Ini bisa dilihat di kolom Charset dari tabel diatas.

Untuk bisa “membaca” character set apa sebuah collation bisa dipakai, kita bisa pecah nama collation menjadi 3 pola:

```
"nama charset"_"nama bahasa / negara"_"case sensitifity".
```

Akhiran `_ci` berarti **case insensitive**, `_cs` berarti **case sensitive**, dan `_bin` berarti **binary**.

Sebagai contoh, `latin1_german1_ci` artinya collation ini digunakan untuk charset `latin1`, bahasa `jerman`, dan bersifat **case insensitive** (tidak membedakan huruf besar dan kecil).

Contoh lain, `latin1_general_cs` artinya collation ini untuk charset `latin1`, bahasa `general` (umum), dan bersifat **case sensitive** (huruf besar dan kecil dianggap berbeda).

Terakhir `latin1_bin` adalah collation khusus untuk charset `binary`.

Lebih lanjut tentang character set dan collation ini (dan juga prakteknya) akan kembali saya bahas di bab tentang membuat tabel.

7.6 Query Pembuatan Database

Dengan tambahan character set dan collation, format dasar pembuatan database di MySQL adalah sebagai berikut:

```
CREATE DATABASE [IF NOT EXISTS] nama_database
[CHARACTER SET charset] [COLLATE collation];
```

Sebagai contoh, untuk membuat database belajar dengan character set `utf8` dan collation `utf8_persian_ci` perintahnya adalah sebagai berikut:

```
CREATE DATABASE belajar CHARACTER SET utf8 COLLATE utf8_persian_ci;
```

Untuk memastikan character set dan collation dari sebuah database, bisa dilihat dengan menjalankan query `SHOW CREATE DATABASE`:

```
SHOW CREATE DATABASE belajar \G
```

```
***** 1. row *****
  Database: belajar
Create Database: CREATE DATABASE `belajar`
/*!40100 DEFAULT CHARACTER SET utf8 COLLATE utf8_persian_ci */
1 row in set (0.00 sec)
```

Sebagai alternatif penulisan, perintah CHARACTER SET bisa disingkat menjadi CHARSET. Database `belajar` sebelumnya, juga bisa dibuat menggunakan query berikut:

```
CREATE DATABASE belajar CHARSET utf8 COLLATE utf8_persian_ci;
```

Bagaimana dengan query pembuatan database yang tidak mencantumkan character set dan collation? MySQL akan menggunakan character set dan collation default. Mari kita cek:

```
CREATE DATABASE baru_belajar;
```

```
SHOW CREATE DATABASE baru_belajar \G
```

```
***** 1. row *****
  Database: baru_belajar
Create Database: CREATE DATABASE `baru_belajar`
/*!40100 DEFAULT CHARACTER SET latin1 */
1 row in set (0.00 sec)
```

Disini saya membuat database `baru_belajar` dengan query `CREATE DATABASE baru_belajar` tanpa mencantumkan character set dan collation. Hasilnya, database ini akan menggunakan character set `latin1`.

Bagaimana dengan collation-nya? Jika tidak dicantumkan, maka akan menggunakan collation default. Apa collation default dari `latin1`, bisa di cek di tabel hasil `SHOW CHARACTER SET`:

```
SHOW CHARACTER SET;
```

Charset	Description	Default collation	Maxlen
big5	Big5 Traditional Chinese	big5_chinese_ci	2
latin1	cp1252 West European	latin1_swedish_ci	1
latin2	ISO 8859-2 Central European	latin2_general_ci	1
swe7	7bit Swedish	swe7_swedish_ci	1

```
40 rows in set (0.00 sec)
```

Untuk baris latin1, Default collation bernilai latin1_swedish_ci. Inilah yang akan menjadi collation dari database baru_belajar, serta semua database yang dibuat dari MariaDB bawaan XAMPP.

Dari penjelasan panjang tentang character set dan collation ini, akhirnya kita sampai ke kesimpulan. Perlukah menambahkan perintah ini untuk pembuatan database?

Jika isi database nanti hanya berisi huruf dalam alphabet biasa dan dalam bahasa indonesia, kita tidak perlu memusingkan hal ini. Character set latin1 yang menjadi character default sudah sanggup menampung seluruh huruf dalam bahasa indonesia.

Tapi jika anda berencana ingin mengisi database dengan karakter non-latin seperti huruf arab, korea atau jepang, bisa menggunakan utf-8.

Sedikit tambahan jika anda ingin menggunakan UTF-8 character set. Di dalam MySQL terdapat 2 versi UTF-8: **utf8** dan **utf8mb4**.

Charset **utf8** dalam MySQL hanya mendukung hingga 3 byte karakter Unicode. Artinya terdapat kemungkinan karakter tidak tampil, terutama karakter tambahan yang butuh 4 byte seperti huruf china kuno dan emoji. Sedangkan charset **utf8mb4** sudah mendukung hingga 4 byte (seluruh karakter Unicode).

7.7 Mengubah Character set dan Collation

MySQL menyediakan perintah ALTER DATABASE untuk mengubah character set dan collation. Berikut format querynya:

```
ALTER DATABASE nama_database [CHARACTER SET charset] [COLLATE collation];
```

Berikut contoh penggunaan dari query ALTER DATABASE:

```
ALTER DATABASE baru_belajar CHARACTER SET ascii COLLATE ascii_bin;

SHOW CREATE DATABASE baru_belajar \G

***** 1. row *****
Database: baru_belajar
Create Database: CREATE DATABASE `baru_belajar`
/*!40100 DEFAULT CHARACTER SET ascii COLLATE ascii_bin */
1 row in set (0.00 sec)
```



Query ALTER digunakan untuk mengubah struktur database dan tabel. Untuk tabel, kita bisa mengubah nama tabel. Tapi MySQL tidak menyediakan perintah untuk mengubah nama database.

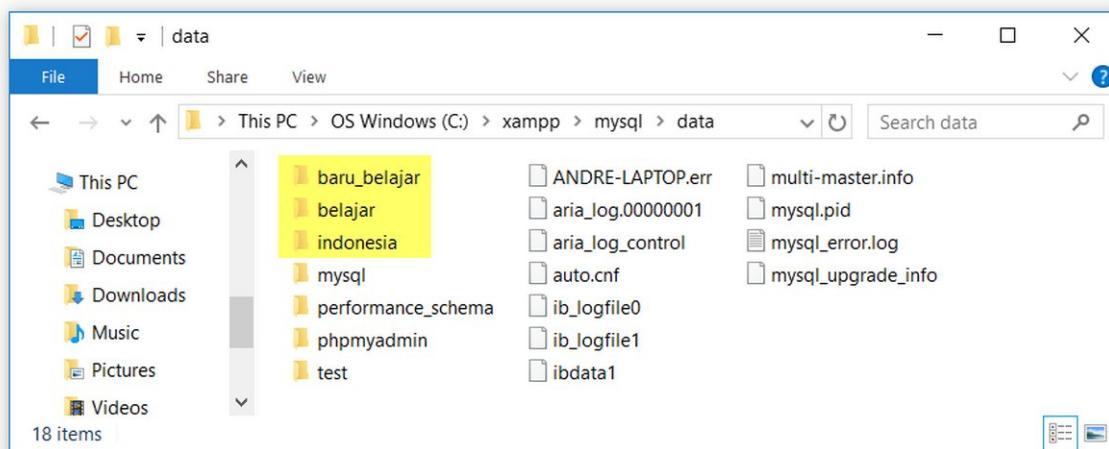
Jika anda ingin mengubah nama database, caranya cukup panjang yakni dengan men-export seluruh database lama (bisa menggunakan mysqldump), lalu diimport ke database baru.

7.8 Folder Penyimpanan Database

Di awal bab MySQL Server dan MySQL Client kita sempat membahas folder penyimpanan database MySQL, yakni di dalam folder **data**.

Jika anda mengikuti dan menjalankan langsung seluruh query yang ada, maka kita sudah membuat 3 buah database: `indonesia`, `belajar`, dan `baru_belajar`.

Silahkan buka folder data ini, dan anda akan menemukan 3 folder dengan nama yang sama.



Gambar: Isi folder data MySQL

Dengan jendela Windows Explorer yang masih terbuka, hapus database `baru_belajar` dari dalam MySQL Client, maka folder `baru_belajar` juga ikut terhapus.

```
DROP DATABASE baru_belajar;
```

Dalam bab ini kita telah membahas perintah tambahan untuk pembuatan database, yakni tentang **character set** dan **collation**. Untuk praktik penggunaannya akan saya bahas di materi tentang pembuatan tabel.

Berikutnya kita akan masuk ke pembahasan tentang tipe data dalam MySQL/MariaDB.

8. Tipe Data MySQL

Hampir semua hal yang kita lakukan di MySQL melibatkan data, karena database sendiri memang ditujukan untuk me-manajemen data.

Terdapat beragam tipe data yang disediakan oleh MySQL. Mulai dari tipe data angka (**numeric**), teks (**string**), hingga tanggal (**date**). Setiap tipe data ini juga memiliki beragam jenis yang dibedakan berdasarkan jangkauan, jenis data yang bisa disimpan, efisiensi serta cara penyimpanan.

Dalam bab ini kita akan membahas tipe data MySQL dengan lebih dalam. Query yang dibutuhkan diantaranya untuk pembuatan tabel (`CREATE TABLE`), menambah data tabel (`INSERT`) dan menampilkan data (`SELECT`). Konsep dasar dari ketiga perintah query ini telah di bahas pada bab: **Dasar Bahasa Query MySQL**.

8.1 Tipe Data Numeric

Tipe data **Numeric** adalah tipe data angka di dalam MySQL. Angka ini bisa berupa angka bulat maupun pecahan. Dalam penulisan query, tipe data angka ditulis tanpa tanda kutip. Berikut contohnya:

```
SELECT 123;
```

```
+----+
| 123 |
+----+
```

```
SELECT 456.77;
```

```
+-----+
| 456.77 |
+-----+
| 456.77 |
+-----+
```

Sama seperti bahasa pemrograman pada lain, MySQL menggunakan tanda titik sebagai penanda pecahan, bukan koma seperti yang kita gunakan sehari-hari. Angka “tiga koma empat belas” ditulis sebagai 3.14 dan bukan 3,14.

Penulisan angka juga bisa menggunakan notasi ilmiah (*scientific notation*), seperti $3e4$ dan $9.8E-4$. Huruf e atau E menandakan pangkat sepuluh. Angka $3e4$ sama artinya dengan 3×10^3 dan angka $9.8E-4$ sama dengan 9.8×10^{-4} :

```
SELECT 3e4;
```

```
+-----+
| 3e4    |
+-----+
| 30000  |
+-----+
```

```
SELECT 9.8E-4;
```

```
+-----+
| 9.8E-4   |
+-----+
| 0.00098  |
+-----+
```

Untuk penulisan notasi ilmiah, tidak boleh ada spasi sebelum dan sesudah huruf e atau E:

```
SELECT 3 e4;
```

```
+----+
| e4  |
+----+
| 3  |
+----+
```

```
SELECT 3e 4;
```

ERROR 1064 (42000): You have an error **in** your **SQL** syntax; **check** the manual
that corresponds **to** your MariaDB server **version for** the **right** syntax **to** use
near '**4**' **at** line 1

Pada contoh pertama, perintah SELECT 3 e4 akan dianggap sebagai angka 3 saja, sedangkan perintah SELECT 3e 4 akan menyebabkan error.

Jenis Tipe Data Numeric

Berdasarkan cara penyimpanannya, tipe data **numeric** MySQL dikelompokkan ke dalam 3 jenis:

- **Exact-value types:** Tipe data angka yang disimpan secara pasti dan presisi. Jenis ini terdiri dari tipe data INTEGER dan DECIMAL.
- **The floating-point types:** Tipe data angka yang disimpan dengan perkiraan (approximate-value). Jenis ini terdiri dari tipe data FLOAT dan DOUBLE.
- **The BIT type:** tipe data angka yang digunakan untuk menyimpan angka biner.

Kita akan bahas dari tipe data integer terlebih dahulu.

8.2 Tipe Data Integer

Tipe data INTEGER adalah tipe data angka yang terdiri dari angka bulat tanpa nilai pecahan, seperti 127 dan 10000.

Terdapat 5 jenis tipe data INTEGER yang dibedakan berdasarkan jangkauannya:

Tipe Data	Jangkauan SIGNED	Jangkauan UNSIGNED	Ukuran
TINYINT	-128 to 127	0 to 255	1 byte
SMALLINT	-32,768 to 32,767	0 to 65,535	2 bytes
MEDIUMINT	-8,388,608 to 8,388,607	0 to 16,777,215	3 bytes
INT	-2,147,483,648 to 2,147,483,647	0 to 4,294,967,295	4 bytes
BIGINT	-9,223,372,036,854,775,808 to -9,223,372,036,854,775,807	0 to 18,446,744,073,709,551,615	8 bytes

Dari kolom **jangkauan**, kita bisa melihat nilai minimum dan maksimum setiap tipe data. Jangkauan ini terdiri dari dua jenis: SIGNED dan UNSIGNED.

Secara default bawaan MySQL, tipe data INTEGER dianggap sebagai SIGNED yang artinya bisa menampung angka bulat negatif dan positif. Jika ditambahkan atribut UNSIGNED, angka negatif “dikorbankan” supaya bisa menampung nilai positif yang lebih besar (kita akan lihat prakteknya sesaat lagi).

Dari tabel diatas juga bisa disimpulkan bahwa semakin besar jangkauan, semakin besar pula ukuran memory yang dibutuhkan. Tipe data TINYINT hanya memerlukan 1 byte memory, sedangkan tipe data BIGINT membutuhkan ruang memory sebesar 8 byte.

Sedapat mungkin pilih tipe data terkecil sesuai kebutuhan. Tipe data yang kecil bisa diproses dengan lebih cepat dan lebih hemat tempat. Akan tetapi tetap pertimbangkan fleksibilitas jangka panjang, jangan sampai aplikasi kita tidak bisa dipakai karena angka yang akan disimpan sudah melebihi limit.

Mari masuk ke contoh praktek. Saya akan membuat sebuah database belajar. Database ini nantinya menampung berbagai contoh tabel sepanjang pembahasan dalam buku ini:

```
CREATE DATABASE belajar;
Query OK, 1 row affected (0.00 sec)
```

```
USE belajar;
Database changed
```

Jika anda masih ingat, query untuk pembuatan tabel adalah sebagai berikut:

```
CREATE TABLE [IF NOT EXISTS] nama_tabel
(nama_kolom tipe_kolom [, nama_kolom tipe_kolom] )
```

Pertama, saya ingin membuat tabel yang memiliki kelima jenis tipe data INTEGER dan menginput beberapa data:

```

CREATE TABLE contoh_int_1 (
    a TINYINT,
    b SMALLINT,
    c MEDIUMINT,
    d INT,
    e BIGINT
);

INSERT INTO contoh_int_1 VALUES ((100), (100), (100), (100), (100));

SELECT * FROM contoh_int_1;
+---+---+---+---+---+
| a | b | c | d | e |
+---+---+---+---+---+
| 100 | 100 | 100 | 100 | 100 |
+---+---+---+---+---+

```

Disini saya membuat tabel `contoh_int_1` dengan 5 kolom: `a`, `b`, `c`, `d` dan `e`. Setiap kolom di definisikan dengan tipe data INTEGER yang berbeda-beda. Setelah itu angka `100` di input ke setiap kolom.

Angka `100` cukup kecil dan bisa ditampung oleh semua kolom. Bagaimana jika angka yang diinput lebih besar dari nilai maksimal yang bisa didukung? Mari kita coba:

```

INSERT INTO contoh_int_1
VALUES ((200), (35000), (18000000), (2500000000), (1000000000000000000));
Query OK, 1 row affected, 5 warnings (0.07 sec)

```

```

SHOW WARNINGS;
+-----+-----+
| Level | Code | Message |
+-----+-----+
| Warning | 1264 | Out of range value for column 'a' at row 1 |
| Warning | 1264 | Out of range value for column 'b' at row 1 |
| Warning | 1264 | Out of range value for column 'c' at row 1 |
| Warning | 1264 | Out of range value for column 'd' at row 1 |
| Warning | 1264 | Out of range value for column 'e' at row 1 |
+-----+-----+

```

```

SELECT * FROM contoh_int_1;
+---+---+---+---+---+
| a | b | c | d | e |
+---+---+---+---+---+
| 100 | 100 | 100 | 100 | 100 |
| 127 | 32767 | 8388607 | 2147483647 | 9223372036854775807 |
+---+---+---+---+---+

```

Saya menginput angka-angka yang berada di luar jangkauan maksimal dari setiap kolom. Jika ini yang terjadi, nilai tersebut tetap bisa masuk ke tabel namun terdapat warning: *Out of range value*.

Hasilnya, angka akan dipotong ke nilai tertinggi dari setiap tipe data. Sebagai contoh, kolom a memiliki tipe data TINYINT. Kolom ini bisa menampung angka antara -128 hingga 127. Ketika diinput angka 200, yang tersimpan adalah 127 (sebagai nilai maksimum untuk kolom TINYINT).

Proses konversi seperti ini sebenarnya bisa di-setting menggunakan **SQL Mode**. Bisa jadi di dalam MySQL yang anda gunakan, menginput angka 200 ke kolom TINYINT akan menghasilkan error dan query gagal dijalankan (tidak terkonversi menjadi 127).

Mengenal SQL_MODE

MySQL memiliki pengaturan server yang dinamakan **SQL MODE**. Di dalam **SQL MODE**, kita bisa mengatur seberapa ketat ‘aturan’ yang diterapkan.

SQL MODE ini bisa berbeda-beda tergantung versi MySQL / MariaDB yang digunakan. Sebagai contoh, untuk MariaDB bawaan XAMPP yang saya pakai, jika ada data yang berada di luar jangkauan suatu kolom (seperti percobaan sebelum ini), MySQL tidak mengeluarkan error, tapi hanya warning. Data tetap bisa diinput namun dipotong sesuai jumlah maksimal karakter yang bisa ditampung.

Akan tetapi jika menggunakan versi MySQL/MariaDB yang berbeda (stand-alone yang diinstall terpisah), bisa saja operasi tersebut menghasilkan error dan data gagal diinput ke dalam tabel. Pengaturan yang lebih ketat ini dikenal juga sebagai “*strict mode*”.

Penjelasan mengenai **SQL MODE** dan cara mengubahnya, akan saya bahas dalam bab tersendiri. Untuk saat ini jika anda mendapati error ketika menginput angka 200 ke kolom TINYINT, artinya MySQL aktif dalam “*strict mode*”.

Percobaan selanjutnya, bagaimana dengan angka pecahan?

```
INSERT INTO contoh_int_1
VALUES ((10.34), (400.499),(12345.50), (143.655), (999.9999));

SELECT * FROM contoh_int_1;
+-----+-----+-----+-----+-----+
| a    | b    | c    | d    | e    |
+-----+-----+-----+-----+-----+
| 100 | 100 | 100 | 100 | 100 |
| 127 | 32767 | 8388607 | 2147483647 | 9223372036854775807 |
| 10  | 400 | 12346 | 144 | 1000 |
+-----+-----+-----+-----+-----+
```

Jika yang diinput berupa angka pecahan, angka tersebut akan dibulatkan ke nilai terdekat. Apabila kurang dari 0.5, dibulatkan ke bawah, jika lebih dari atau sama dengan 0.5, dibulatkan ke atas.

Sebagai contoh, angka 10.34 disimpan sebagai 10 (dibulatkan ke bawah) dan angka 12345.50 akan dikonversi menjadi 12346 (dibulatkan ke atas).

Tabel contoh_int_1 ini menggunakan pengaturan default dari tipe data integer. Untuk memodifikasi pengaturan (setting) sebuah kolom, bisa ditambahkan satu atau beberapa **atribut** pada saat pembuatan tabel.

Format dasar pembuatan tabel dengan **atribut** adalah sebagai berikut:

```
CREATE TABLE [IF NOT EXISTS] nama_tabel
(nama_kolom tipe_kolom atribut_kolom, ... )
```

Sebagai contoh, saya ingin membuat tabel yang memiliki kolom INTEGER dengan atribut UNSIGNED:

```
CREATE TABLE contoh_int_2 (
    a SMALLINT,
    b SMALLINT UNSIGNED
);
```

Tabel contoh_int_2 memiliki 2 buah kolom: a dan b. Kolom a menggunakan tipe data SMALLINT. Kolom b juga bertipe SMALLINT tapi kali ini ada tambahan atribut UNSIGNED.

Apa efeknya? Atribut UNSIGNED akan memperbesar jangkauan angka positif tipe data SMALLINT dengan cara mengalihkan “jatah” dari angka negatif. Secara default, kolom SMALLINT sanggup menampung angka dari -32768 hingga 32767. Dengan tambahan atribut UNSIGNED, jangkauannya menjadi 0 hingga 65535.

Seluruh tipe data integer bisa ditambahkan atribut UNSIGNED. Jangkauannya bisa dilihat dalam tabel tipe data integer sebelumnya.

Saya akan coba input beberapa nilai ke tabel contoh_int_2:

```
INSERT INTO contoh_int_2 VALUES (50000,50000), (-123,-123);
```

```
Query OK, 2 rows affected, 2 warnings (0.06 sec)
```

```
Records: 2  Duplicates: 0  Warnings: 2
```

```
SELECT * FROM contoh_int_2;
+-----+-----+
| a     | b     |
+-----+-----+
| 32767 | 50000 |
| -123  |      0 |
+-----+-----+
```

Nilai maksimum dari tipe data SMALLINT adalah 32767, sehingga ketika diinput nilai 50000 ke kolom a, akan dikonversi menjadi 32767 (nilai positif maksimum dari SMALLINT).

Untuk kolom b yang menggunakan atribut UNSIGNED, sanggup menampung hingga 65535, karena itu angka 5000 bisa disimpan ke dalam tabel. Sebaliknya, ketika diinput angka negatif, kolom b hanya bisa menyimpan angka 0 (nilai paling kecil dari tipe data SMALLINT UNSIGNED).

Pengaturan lain terkait tipe data INTEGER adalah untuk mengatur lebar tampilan (*display width*). Nilai ini ditulis dalam tanda kurung setelah tipe data, seperti INT(5) atau BIGINT(7).

Ketika sebuah kolom di definisikan sebagai INT(5), angka yang ada di dalamnya akan ditampilkan dengan lebar 5 digit. Namun ini tidak membatasi nilai maksimum kolom tersebut. Apabila angka yang diinput lebih dari 5 digit, tetap ditampilkan semua selama masih bisa ditampung oleh tipe data INT. Nilai maksimum dari INT(5) tetap 2147483647, bukan 99999.

Mari kita coba:

```
CREATE TABLE contoh_int_3 (
    a INT(5)
);

INSERT INTO contoh_int_3 VALUES (1), (13), (589), (9876);
Query OK, 4 rows affected (0.10 sec)
Records: 4  Duplicates: 0  Warnings: 0

SELECT * FROM contoh_int_3;
+---+
| a |
+---+
|   1 |
|  13 |
| 589 |
| 9876|
+---+
```

Dengan membuat kolom a sebagai INT(5), angka yang ada di dalam tabel contoh_int_3 ditampilkan dengan lebar 5 digit. Untuk angka 1, akan terdapat 4 buah spasi di sebelah kiri angka tersebut. Untuk angka 13 akan ada 3 buah spasi di sebelah kiri.

Efek *display width* akan jelas terlihat ketika kita menambahkan atribut ZEROFILL pada saat pembuatan kolom. Atribut ZEROFILL berfungsi untuk mengisi angka 0 di sebelah kiri selama belum sesuai dengan jumlah display width.

```
CREATE TABLE contoh_int_4 (
    a INT(7) ZEROFILL
);

INSERT INTO contoh_int_4 VALUES (1), (13), (589), (9876);
Query OK, 4 rows affected (0.07 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM contoh_int_4;
```

a
0000001
0000013
0000589
0009876

Saya membuat tabel contoh_int_4 dengan pendefinisian kolom a INT(7) ZEROFILL. Artinya, kolom a bisa menampung angka integer dengan aturan: jika angka tersebut kurang dari 7 digit, tambahkan angka 0 di sisi kiri. Ketika diinput angka 1, di dalam tabel akan menjadi 0000001.

Bagaimana jika angka yang diinput lebih dari 7 digit?

```
INSERT INTO contoh_int_4 VALUES (123456789);
```

```
SELECT * FROM contoh_int_4;
```

a
0000001
0000013
0000589
0009876
123456789

Saya menginput angka 123456789 ke dalam kolom a, hasilnya tetap bisa tersimpan meskipun sudah melebihi dari 7 digit.

Penambahan atribut ZEROFILL juga otomatis membuat kolom tersebut sebagai UNSIGNED. Artinya, kita tidak bisa menginput angka negatif:

```
INSERT INTO contoh_int_4 VALUES (-123), (4e9);
Query OK, 2 rows affected, 1 warning (0.05 sec)
Records: 2  Duplicates: 0  Warnings: 1
```

```
SELECT * FROM contoh_int_4;
+-----+
| a      |
+-----+
| 0000001 |
| 0000013 |
| 0000589 |
| 0009876 |
| 123456789 |
| 0000000 |
| 4000000000 |
+-----+
```

Saya menginput angka -123 ke dalam kolom a dari tabel contoh_int_4. Angka yang tersimpan adalah 0000000. Begitu juga ketika saya menginput angka 4e9 (4×10^9), angka ini bisa disimpan meskipun sudah melebihi nilai maksimum dari tipe INT standar. Ini semua bisa terjadi karena efek UNSIGNED dari atribut ZEROFILL.

Kesimpulannya, kolom a INT(7) ZEROFILL sama efeknya dengan a INT(7) ZEROFILL UNSIGNED. Dengan tambahan atribut UNSIGNED dan ZEROFILL, berikut format dasar penulisan tipe data INTEGER MySQL:

INT[(M)] [UNSIGNED] [ZEROFILL]

Huruf M menandakan lebar tampilan (*display width*), boleh ditulis atau tidak (opsional). Begitu juga dengan atribut UNSIGNED dan ZEROFILL yang bisa dipakai saat dibutuhkan (tidak harus selalu ditulis).

Tipe data INTEGER MySQL juga memiliki variasi nama lain agar konsisten dengan bahasa standar SQL:

- **TINYINT** = INT1.
- **SMALLINT** = INT2.
- **MEDIUMINT** = INT3 = MIDDLEINT.
- **INT** = INTEGER = INT4.
- **BIGINT** = INT8.
- Tipe khusus **BOOL** dan **BOOLEAN** yang artinya sama dengan TINYINT(1).

Berikut contoh pembuatan tabel dengan nama alternatif ini:

```
CREATE TABLE contoh_int_5 (
    a INT1,
    b INT2,
    c INT3,
    d INT4,
    e INT8,
    f BOOLEAN,
    g INTEGER
);

DESC contoh_int_5;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| a     | tinyint(4) | YES  |     | NULL    |       |
| b     | smallint(6) | YES  |     | NULL    |       |
| c     | mediumint(9) | YES  |     | NULL    |       |
| d     | int(11)    | YES  |     | NULL    |       |
| e     | bigint(20) | YES  |     | NULL    |       |
| f     | tinyint(1)  | YES  |     | NULL    |       |
| g     | int(11)    | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
```

Dari hasil query `DESC contoh_int_5;`, terlihat bahwa secara internal MySQL tetap menggunakan nama tipe data standar seperti TINYINT dan SMALLINT, dan tidak menyimpannya sebagai INT1 dan INT2.

8.3 Tipe Data Decimal

Tipe data DECIMAL digunakan untuk menampung angka pecahan, seperti 3.14 atau 999.9999.

MySQL sebenarnya memiliki 3 jenis tipe data untuk angka pecahan, yakni DECIMAL, FLOAT dan DOUBLE. Bedanya, tipe data DECIMAL disimpan menggunakan sistem **fixed point**.

Dalam sistem **fixed point**, angka akan disimpan persis seperti bentuk aslinya. Angka 0.0032 akan disimpan seperti itu. Oleh karena itu tipe data DECIMAL cocok untuk nilai yang memerlukan ketelitian, seperti jumlah uang.

Hal ini berbeda dengan tipe data FLOAT dan DOUBLE yang menggunakan sistem **floating point**. Dalam tipe data *floating point*, angka disimpan menggunakan “perkiraan” (*proximation*). Angka 0.0032 tidak persis disimpan sebagai 0.0032, tapi bisa saja 0.0032000001. Mengenai perbedaan ini akan kita bahas di materi tentang tipe data FLOAT dan DOUBLE setelah ini.

Tipe data DECIMAL memiliki 2 pengaturan yang berfungsi untuk membatasi berapa jumlah digit yang akan disimpan (jangkauan dari angka). Format dasarnya sebagai berikut:

```
DECIMAL [(M[,D])] [UNSIGNED] [ZEROFILL]
```

M adalah total jumlah digit keseluruhan, sedangkan **D** adalah jumlah digit di belakang koma (pecahan). Sebagai contoh, **DECIMAL [6,2]** digunakan untuk membuat kolom 6 digit angka dengan 2 digit di belakang koma.

Nilai **M** dan **D** juga akan membatasi jangkauan minimum dan maksimum dari kolom tersebut:

Deklarasi	Jangkauan
DECIMAL (4,1)	-999,9 hingga 999,9
DECIMAL (6,2)	-9999,99 hingga 9999,99
DECIMAL (3,2)	-9,99 hingga 9,99
DECIMAL (8,2)	-999999,99 hingga 999999,99

Apabila nilai **M** tidak ditulis, secara default **M = 10**, jika nilai **D** tidak ditulis akan dianggap **D = 0**. Sehingga apabila kedua nilai ini tidak ditulis, secara default MySQL akan menggunakan **DECIMAL (10,0)**. Dimana kolom tersebut hanya bisa menampung 10 digit angka bulat (tanpa nilai pecahan).

Nilai maksimal dari **M = 65**, dan nilai **D = 30**. Dengan syarat, nilai **D** tidak boleh lebih besar dari **M**.

Opsional query **UNSIGNED** jika disertakan akan membuang nilai negatif dari **DECIMAL** tanpa memperbesar jangkauannya (tidak seperti tipe data **INTEGER**). Opsional query **ZEROFILL** akan mengisi angka 0 di bagian kiri (angka bulat) dan kanan (angka pecahan).

Berikut contoh penggunaan dari tipe data **DECIMAL**:

```
CREATE TABLE contoh_dec_1 (
    a DECIMAL,
    b DECIMAL(5,2),
    c DECIMAL(6,3),
    d DECIMAL(9,4)
);

INSERT INTO contoh_dec_1 VALUES
(10, 10, 10, 10),
(13.4, 13.4, 13.4, 13.4),
(98123.45, 98123.45, 98123.45, 98123.45),
(0.0065, 0.0065, 0.0065, 0.0065),
(1e1, 2.3489e2, 7.11e-1, 9.87e-2);
```

```
Query OK, 4 rows affected, 7 warnings (0.07 sec)
Records: 4  Duplicates: 0  Warnings: 7
```

```
SELECT * FROM contoh_dec_1;
+-----+-----+-----+-----+
| a     | b     | c     | d     |
+-----+-----+-----+-----+
```

```

|   10 | 10.00 | 10.000 | 10.0000 |
|   13 | 13.40 | 13.400 | 13.4000 |
| 98123 | 999.99 | 999.999 | 98123.4500 |
|    0 | 0.01 | 0.007 | 0.0065 |
|   10 | 234.89 | 0.711 | 0.0987 |
+-----+-----+-----+

```

Saya membuat tabel contoh_dec_1 dengan 5 kolom, masing-masing kolom menggunakan definisi tipe data DECIMAL yang berbeda-beda. Setelah itu setiap kolom diinput berbagai angka.

Untuk kolom a yang di definisikan tanpa nilai M dan D, akan dianggap sebagai DECIMAL(10,0). Efeknya, nilai pecahan dibulatkan ke angka terdekat (sama seperti jika nilai pecahan diinput ke kolom INTEGER).

Kolom b didefinisikan sebagai DECIMAL(5,2). Artinya kolom ini bisa menyimpan total 5 digit angka, dimana 3 digit untuk angka bulat, dan 2 digit untuk angka pecahan. Jika angka yang diinput melebihi ketentuan ini, maka akan dibulatkan. Begitu juga dengan kolom c, d dan e yang masing-masingnya menggunakan jumlah digit yang berbeda-beda.

Jika kolom dengan tipe data DECIMAL ditambahkan atribut UNSIGNED, ini hanya menghapus nilai negatif dan tidak menambah jangkauan dari DECIMAL itu sendiri. Berikut percobaannya:

```

CREATE TABLE contoh_dec_2 (
  a DECIMAL(5,2) UNSIGNED,
  b DECIMAL(5,2) ZEROFILL
);

INSERT INTO contoh_dec_2 VALUES (100000, 100000),(-1.23, -1.23),(1.23, 1.23);

```

```

Query OK, 3 rows affected, 4 warnings (0.04 sec)
Records: 3  Duplicates: 0  Warnings: 4

```

```

SELECT * FROM contoh_dec_2;
+-----+-----+
| a      | b      |
+-----+-----+
| 999.99 | 999.99 |
| 0.00   | 000.00 |
| 1.23   | 001.23 |
+-----+-----+

```

Tabel contoh_dec_2 memiliki dua kolom: a dan b. Kolom a di definisikan sebagai DECIMAL(5,2) UNSIGNED, yang artinya kolom ini tidak bisa menyimpan angka negatif. Jika tetap diinput, hasilnya menjadi 0.

Kolom b didefinisikan sebagai DECIMAL(5,2) ZEROFILL. Kali ini, angka nol akan ditambahkan di sisi kiri dan kanan angka jika tidak sampai 5 digit. Selain itu, penggunaan atribut ZEROFILL juga

secara tidak langsung akan menghapus kemampuan kolom tersebut untuk menyimpan angka negatif (seolah-olah diberikan atribut UNSIGNED).

Tipe data DECIMAL juga bisa ditulis dengan keyword NUMERIC, DEC dan FIXED. Seperti contoh berikut:

```
CREATE TABLE contoh_dec_3 (
    a DEC(5,2),
    b NUMERIC(5,2),
    c FIXED(5,2)
);

DESC contoh_dec_3;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| a     | decimal(5,2) | YES  |     | NULL    |       |
| b     | decimal(5,2) | YES  |     | NULL    |       |
| c     | decimal(5,2) | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
```

Dari hasil query DESC contoh_dec_3, terlihat bahwa kolom NUMERIC, DEC dan FIXED secara internal MySQL tetap disimpan sebagai DECIMAL.

8.4 Tipe Data Float dan Double

Tipe data FLOAT dan DOUBLE digunakan untuk menyimpan angka pecahan (sama seperti tipe data DECIMAL). Kedua tipe data ini juga menggunakan pengaturan yang sama, yakni dengan format:

```
FLOAT [(M[,D])] [UNSIGNED] [ZEROFILL]
DOUBLE [(M[,D])] [UNSIGNED] [ZEROFILL]
```

M merupakan total jumlah digit keseluruhan, sedangkan D adalah jumlah digit di belakang koma (pecahan). Sebagai contoh, FLOAT [6,2] digunakan untuk membuat kolom 6 digit angka dengan 2 digit di belakang koma.

Perbedaan antara tipe data FLOAT dan DOUBLE dengan tipe data DECIMAL terletak pada teknik penyimpanan di memory komputer.

Tipe data FLOAT dan DOUBLE disimpan menggunakan teknik **floating point**. Dalam tipe data *floating point*, angka disimpan menggunakan “perkiraan” (*proximation*). Angka 0.0032 tidak persis disimpan sebagai 0.0032, tapi bisa saja 0.0032000001. Ini terjadi karena angka tersebut disimpan menggunakan kelipatan biner.

Keunggulan dari teknik seperti ini, jangkauan angka yang disimpan bisa lebih besar daripada tipe data DECIMAL. Selain itu juga bisa diproses dengan lebih cepat.

Tipe Data	Jangkauan	Ukuran
FLOAT	-3.402823466E+38 to 3.402823466E+38	4 bytes
DOUBLE	-1.7976931348623157E+308 to 1.7976931348623157E+308	8 bytes

Tipe data FLOAT disimpan menggunakan teknik **single precision**, sedangkan tipe data DOUBLE menggunakan **double precision**. Perbedaan keduanya sangat teknis dan tidak akan saya bahas. Namun seperti yang bisa dilihat dari tabel diatas, jangkauan tipe data DOUBLE jauh lebih besar daripada tipe data FLOAT.

Dalam tipe data DECIMAL, maksimal nilai **M = 65**, dan nilai **D = 30**. Untuk tipe data FLOAT dan DOUBLE, maksimal nilai **M** adalah 255, dan nilai **D** tetap 30. Artinya, angka yang bisa ditampung oleh tipe data FLOAT dan DOUBLE jauh lebih besar dibandingkan tipe data DECIMAL.

Jika nilai **M** dan **D** tidak ditulis, akan di set dengan nilai maksimal.

Efek penggunaan atribut **UNSIGNED** dan **ZEROFILL** pada tipe data FLOAT dan DOUBLE sama seperti pada tipe data DECIMAL. Yakni penggunaan **UNSIGNED** hanya akan menghapus kemampuan menyimpan angka negatif (tidak memperbesar jangkauan). Sedangkan ATRIBUT **ZEROFILL** akan menambah angka 0 di sisi kiri dan kanan.

Berikut contoh penggunaan tipe data FLOAT dan DOUBLE :

```
CREATE TABLE contoh_float_1 (
    a FLOAT(3,2),
    b FLOAT(4,2),
    c FLOAT(5,2),
    d DOUBLE ZEROFILL UNSIGNED,
    e DOUBLE(8,2) ZEROFILL
);

INSERT INTO contoh_float_1 VALUES
((1.3), (55.32), (523.77), (7832.3), (150000.45));

SELECT * FROM contoh_float_1;
+-----+-----+-----+-----+-----+
| a    | b    | c    | d          | e      |
+-----+-----+-----+-----+
| 1.30 | 55.32 | 523.77 | 0000000000000007832.3 | 150000.45 |
+-----+-----+-----+-----+
```

Saya merancang tabel contoh_float_1 dengan 5 kolom. Setiap kolom menggunakan pendefinisian tipe data FLOAT dan DOUBLE yang berbeda-beda. Kemudian saya menginput beberapa angka. Dari hasil query SELECT, tidak tampak ada masalah. Setiap angka pecahan sukses disimpan ke dalam tabel.

Mari kita coba contoh lain:

Kali ini saya membuat tabel contoh_float_2 dengan tipe kolom yang cukup ekstrim. Kolom a dengan FLOAT(200,10), sedangkan kolom b dengan DOUBLE(220,20). Artinya, kedua kolom ini dirancang untuk bisa menampung ratusan digit. Setelah itu diinput angka 123.456 ke masing-masing kolom.

Perhatikan hasil yang di dapat. Untuk kolom a, angka yang disimpan adalah 123.4560012817, tidak pas sejumlah 123.456. Inilah salah satu contoh kasus pendekatan atau “perkiraan” (*proximation*) untuk tipe data FLOAT. Meskipun saya menginput angka 123.456, yang disimpan adalah 123.4560012817.

Mari kita lihat contoh lain:

```
CREATE TABLE contoh_float_3 (
    a FLOAT(5,2),
    b DECIMAL(5,2)
);

INSERT INTO contoh_float_3 VALUES ((5.33), (5.33));

SELECT * FROM contoh_float_3;
+-----+-----+
| a    | b    |
+-----+-----+
| 5.33 | 5.33 |
+-----+-----+
```

Untuk tabel contoh_float_3, berisi 2 kolom. Kolom a didefinisikan dengan FLOAT(5,2) dan kolom b di definisikan sebagai DECIMAL(5,2). Kedua kolom ini kemudian saya input dengan angka 5.33. Keduanya menampilkan angka 5.33 dengan sukses.

Selanjutnya, menggunakan query SELECT, saya ingin menampilkan kolom a dan kolom b dimana masing-masing kolom dikali 100000. Hasil yang didapat seharusnya $5.33 * 100000 = 533000.00$. Mari kita coba:

```
SELECT a*100000, b*100000 FROM contoh_float_3;
+-----+-----+
| a*100000 | b*100000 |
+-----+-----+
| 532999.99 | 533000.00 |
+-----+-----+
```

Perhatikan nilai kolom a, hasilnya tidak pas 533000.00, tetapi 532999.99. Kembali, kita bisa melihat “kesalahan” pembulatan untuk tipe data FLOAT.

Bagaimana jika dikali dengan angka 1000000000?

```
SELECT a*1000000000, b*1000000000 FROM contoh_float_3;
+-----+-----+
| a*1000000000 | b*1000000000 |
+-----+-----+
| 5329999923.71 | 5330000000.00 |
+-----+-----+
```

Kesalahan pembulatan pada kolom a makin terlihat.

Karena inilah tipe data FLOAT dan DOUBLE tidak direkomendasikan untuk data yang butuh ketelitian. Sebaiknya anda selalu menggunakan tipe data DECIMAL untuk menyimpan angka pecahan.

Satu-satunya alasan menggunakan tipe data FLOAT dan DOUBLE adalah untuk menampung angka yang sangat sangat besar dan tidak bisa ditampung oleh tipe data DECIMAL. Selain itu, nilai tersebut juga bisa mentoleransi sedikit kesalahan pembulatan (*rounding error*). Contoh kasus untuk data ini seperti menyimpan jarak antar bintang.

MySQL juga mendukung nama alternatif untuk tipe data FLOAT dan DOUBLE :

- **DOUBLE [(M[,D])]** = **DOUBLE PRECISION [(M[,D])]** = **REAL [(M[,D])]**.
- **FLOAT4** = tipe data FLOAT tanpa nilai M dan D.
- **FLOAT8** = tipe data DOUBLE tanpa nilai M dan D.

Berikut contoh penggunaannya:

```
CREATE TABLE contoh_float_4 (
    a FLOAT4,
    b REAL(6,2),
    c DOUBLE PRECISION(6,2),
    d FLOAT8
);

DESC contoh_float_4;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| a     | float     | YES  |     | NULL    |       |
| b     | double(6,2) | YES  |     | NULL    |       |
| c     | double(6,2) | YES  |     | NULL    |       |
| d     | double     | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
```

Pada tabel contoh_float_4, setiap kolom di definisikan dengan nama alternatif, namun secara internal tetap diproses MySQL sebagai tipe data FLOAT dan DOUBLE.

8.5 Tipe Data Bit

Tipe data BIT adalah tipe data khusus untuk menyimpan digit biner, yakni angka yang terdiri dari dua digit saja: 0 dan 1.

Untuk bisa memahami penggunaan tipe data ini, saya anggap anda sudah paham tentang sistem bilangan biner. Jika tidak, boleh melewatkannya karena tipe data BIT sendiri sangat jarang dipakai.

Berikut format dasar penulisan kolom dengan tipe data BIT:

`BIT[(M)]`

Nilai M digunakan untuk men-set berapa digit angka biner yang akan disimpan. Nilai M berkisar antara 1 (minimum) hingga 64 (maksimum). Jika tidak ditulis maka dianggap sebagai 1.

Mari kita masuk ke contoh penggunaan:

```
CREATE TABLE contoh_bit_1 (a BIT(3));
```

Saya membuat tabel contoh_bit_1 dengan kolom a yang didefinisikan sebagai BIT(3). Artinya, kolom a bisa diisi angka biner sebanyak 3 digit, mulai dari 000 hingga 111. Dalam nilai desimal, kolom a bisa diisi nilai 0 hingga 7.

Untuk menginput angka ke kolom dengan tipe data BIT, kita bisa langsung menginput angka desimal:

```
INSERT INTO contoh_bit_1 VALUES (0),(3),(7);
```

Jika ingin menginput dengan angka biner, harus dengan format khusus: b'angka_biner', seperti contoh berikut:

```
INSERT INTO contoh_bit_1 VALUES (b'011'),(b'110');
```

Nilai b'011' artinya saya ingin memasukkan angka biner 011, sedangkan nilai b'110' artinya saya ingin menginput angka biner 110. Mari kita lihat hasilnya:

```
SELECT * FROM contoh_bit_1;
+-----+
| a   |
+-----+
|      |
|      |
|      |
|      |
+-----+
```

Apa yang terjadi? Secara bawaan, MySQL tidak bisa menampilkan kolom tipe data BIT secara langsung. Kita harus mengkonversinya ke dalam sistem bilangan tertentu.

Untuk menampilkan kolom a sebagai bilangan biner, bisa menggunakan fungsi BIN() dalam query SELECT:

```
SELECT BIN(a) FROM contoh_bit_1;
+-----+
| BIN(a) |
+-----+
| 0       |
| 11      |
| 111     |
| 11      |
| 110     |
+-----+
```

Untuk menampilkannya sebagai bilangan desimal (basis 10), bisa menggunakan fungsi CAST (nama_kolom as UNSIGNED):

```
SELECT CAST(a as UNSIGNED) FROM contoh_bit_1;
+-----+
| CAST(a as UNSIGNED) |
+-----+
| 0 |
| 3 |
| 7 |
| 3 |
| 6 |
+-----+
```

Untuk menampilkan kolom a dalam sistem bilangan *octal* dan *heksadesimal* juga bisa menggunakan fungsi OCT(a) dan HEX(a).

```
SELECT BIN(a), OCT(a), HEX(a) FROM contoh_bit_1;
+-----+-----+-----+
| BIN(a) | OCT(a) | HEX(a) |
+-----+-----+-----+
| 0      | 0      | 0      |
| 11     | 3      | 3      |
| 111    | 7      | 7      |
| 11     | 3      | 3      |
| 110    | 6      | 6      |
+-----+-----+-----+
```

Sekali lagi, tipe data BIT tidak terlalu banyak dipakai. Implementasinya terbatas untuk aplikasi yang butuh mengolah data dalam bentuk digit biner.

8.6 Tipe Data String

Tipe data **string** adalah sebutan untuk data teks yang terdiri dari huruf-huruf alfabet, seperti: 'a', 'A', 'aku', dan 'Belajar MySQL di DuniaIlkom'.

Untuk membuat data string, penulisannya harus di dalam tanda kutip satu (') atau tanda kutip dua ("), seperti contoh berikut:

```
SELECT 'sedang belajar MySQL';
+-----+
| sedang belajar MySQL |
+-----+
| sedang belajar MySQL |
+-----+

SELECT "sedang belajar MySQL";
+-----+
| sedang belajar MySQL |
+-----+
| sedang belajar MySQL |
+-----+
```

Meskipun MySQL mendukung tanda kutip satu dan tanda kutip dua untuk membuat string, yang disarankan adalah menggunakan tanda kutip satu, sesuai dengan standar bahasa SQL.

Jika di dalam string tersebut terdapat karakter khusus seperti tanda kutip, maka harus di-*escape* menggunakan forward-slash (\). Berikut percobaannya:

```
SELECT 'sedang belajar MySQL di hari jum\'at';
+-----+
| sedang belajar MySQL di hari jum'at |
+-----+
| sedang belajar MySQL di hari jum'at |
+-----+
```

Jika kata jum\'at diatas ditulis tanpa tanda (\'), MySQL akan menganggap tanda kutip tersebut sebagai penutup string.

Berikut daftar karakter khusus yang harus di-*escape* jika ingin ditulis sebagai bagian dari string:

Cara Penulisan String	Karakter Yang Ditampilkan
\0	Karakter NULL ASCII
\'	Tanda kutip satu (')
\"	Tanda kutip dua (")
\b	Karakter backspace
\n	Karakter newline (linefeed)
\r	Karakter carriage return
\t	Karakter tab
\z	ASCII 26 (Control+Z)
\\\	Karakter backslash (\)
\%	Karakter persen (%)
_	Karakter underscore (_)

Khusus untuk karakter \% dan _ hanya perlu jika string tersebut digunakan untuk pencarian (query LIKE), karena karakter % dan _ memiliki makna khusus dalam query LIKE.

Tanda kutip satu atau dua juga tidak perlu di escape jika berada di dalam tanda kutip yang berbeda, misalnya:

```
SELECT "sedang belajar MySQL di hari jum'at";
+-----+
| sedang belajar MySQL di hari jum'at |
+-----+
| sedang belajar MySQL di hari jum'at |
+-----+  
  
SELECT 'I"ll be back';
+-----+
| I"ll be back |
+-----+
| I"ll be back |
+-----+
```

Yang juga perlu diperhatikan, jika karakter angka berada di dalam tanda kutip, akan dianggap sebagai tipe data string, bukan number. Ini sering menjadi sumber error pada saat pembuatan query MySQL.

```
SELECT '9';
+---+
| 9 |
+---+
| 9 |
+---+  
  
SELECT 9;
+---+
| 9 |
+---+
| 9 |
+---+
```

Query `SELECT '9'` akan menghasilkan teks 9 (tipe data **string**), sedangkan query `SELECT 9` akan menghasilkan angka 9 (tipe data **number**).

Jenis Tipe Data String

Untuk pembuatan kolom tabel, MySQL menyediakan beragam tipe data string:

- **CHAR** dan **VARCHAR**
- **BINARY** dan **VARBINARY**

- **BLOB** dan **TEXT**
- **ENUM**
- **SET**

Setiap tipe data memiliki karakteristik masing-masing, mulai dari mekanisme penyimpanan, ukuran, *case sensitivity* (perbedaan huruf besar dan kecil), dll. Kita akan membahasnya satu persatu.

8.7 Tipe Data Char dan Varchar

Tipe data CHAR dan VARCHAR adalah tipe data karakter (string) yang paling sederhana dan juga paling sering digunakan.

Format dasar penulisan tipe data CHAR dan VARCHAR adalah sebagai berikut:

```
CHAR [(M)]
VARCHAR [(M)]
```

Dimana **M** berguna untuk membatasi jumlah karakter yang dialokasikan oleh MySQL.

Misalkan nilai M=5, artinya MySQL menyediakan 5 karakter untuk kolom tersebut. Nilai maksimal M adalah 255 untuk CHAR, dan 65.535 untuk VARCHAR. Jika nilai M tidak di nyatakan, nilai dianggap sebagai M=1.

Disini terlihat bahwa tipe data VARCHAR memiliki jumlah karakter maksimal yang lebih besar daripada CHAR. Selain itu perbedaan antara CHAR dengan VARCHAR terdapat dari cara MySQL mengalokasikan memory penyimpanan.

Ketika sebuah kolom di definisikan sebagai CHAR(5), MySQL menyiapkan 5 karakter (5 byte) untuk setiap baris. Walaupun huruf yang disimpan hanya 1 (butuh 1 byte), di memory tetap butuh 5 byte.

Apabila kolom tersebut di definisikan sebagai VARCHAR(5), MySQL hanya butuh ruang memory sesuai dengan jumlah karakter yang disimpan. Jika kita menginput 2 karakter (2 byte), ukuran memory yang digunakan hanya 2 byte + 1 byte (penjelasannya ada sesaat lagi).

Dari perbedaan ini, terlihat bahwa VARCHAR lebih fleksibel dan hemat tempat dibandingkan dengan tipe data CHAR.

Akan tetapi untuk fleksibilitas ini, tipe VARCHAR memerlukan sedikit proses tambahan (yang sebenarnya nyaris tidak terdeteksi). Selain itu tipe data VARCHAR butuh 1 byte tambahan untuk menyimpan informasi terkait jumlah karakter yang disimpan.

Teknis perbedaan antara CHAR dan VARCHAR dapat dilihat dari tabel berikut:

Data	CHAR(5)	Ukuran Peyimpanan	VARCHAR(5)	Ukuran Peyimpanan
' '	' '	5 byte	' '	1 byte
'du'	'du '	5 byte	'du'	3 byte
'dunia'	'dunia'	5 byte	'dunia'	6 byte
'dunia ilkom'	'dunia'	5 byte	'dunia'	6 byte

Dari tabel terlihat ukuran penyimpanan VARCHAR berubah-ubah tergantung data yang disimpan. Kolom VARCHAR juga butuh 1 byte tambahan yang berfungsi untuk menyimpan panjang data.

Jika kita definisikan sebuah kolom sebagai VARCHAR(5) dan menginputkan kata ‘dunia’, akan butuh 5+1 byte. Berbeda dengan CHAR(5) yang tidak butuh byte tambahan.

Jadi tipe data apa yang sebaiknya dipakai? *Tergantung kebutuhan*.

Tipe data CHAR cocok untuk kolom yang memiliki jumlah karakter tetap, misalnya nomor KTP, ID pelanggan atau NIM mahasiswa. Data seperti ini butuh jumlah karakter yang tetap, sehingga akan selalu terisi penuh. Dengan menyimpannya ke dalam kolom CHAR, pemrosesan data akan menjadi lebih sederhana.

Tipe data VARCHAR lebih cocok untuk kolom yang berisi data berbeda ukuran (panjangnya bervariasi), seperti nama, alamat, atau teks keterangan singkat. Data ini memiliki panjang yang berbeda-beda, sehingga akan lebih hemat memory jika disimpan ke dalam VARCHAR dibandingkan CHAR.

Baik CHAR maupun VARCHAR akan disimpan oleh MySQL secara *case insensitive*. Dimana huruf besar dan kecil dianggap sama.

Mari kita coba membuat tabel dengan tipe data CHAR dan VARCHAR:

```
CREATE TABLE contoh_char_1 (
    a CHAR(5),
    b VARCHAR(5)
);

INSERT INTO contoh_char_1 VALUES ('dunia', 'dunia');

SELECT * FROM contoh_char_1;
+----+----+
| a   | b   |
+----+----+
| dunia | dunia |
+----+----+
1 row in set (0.01 sec)
```

Saya membuat tabel contoh_char_1 dengan dua kolom: a sebagai CHAR(5), dan b sebagai VARCHAR(5). Kedua kolom ini diinput dengan string 'dunia' kemudian ditampilkan menggunakan query SELECT.

Secara internal, kolom tipe data CHAR disimpan dengan **tambahan spasi** di depan dan di belakang data (selama belum memenuhi maksimal karakter). Karakter spasi ini akan dihapus saat ditampilkan. Sistem penyimpanan seperti ini bisa menjadi kendala tersendiri.

Jika kita menginput data yang memiliki spasi di belakang, seperti: 'aku ' (terdapat 3 spasi setelah 'aku'), pada saat data ditampilkan MySQL otomatis menghapus spasi tersebut menjadi 'aku'.

Sedangkan untuk tipe data VARCHAR, akan tetap disimpan dan ditampilkan sebagai 'aku ' (lengkap dengan 3 spasi setelah 'aku').

Mari kita lihat contoh kasusnya:

```
INSERT INTO contoh_char_1 VALUES ('a      ', 'a      ');
```

```
SELECT a,b FROM contoh_char_1;
+-----+-----+
| a     | b     |
+-----+-----+
| dunia | dunia |
| a     | a     |
+-----+-----+
```

Disini saya menginput data baru ke dalam tabel contoh_char_1, yakni karakter 'a ' (huruf a dengan 4 spasi di belakang). Dari hasil query SELECT tampak tidak ada perbedaan karena “spasi” ini memang tidak terlihat.

Untuk bisa melihat efek “penghilangan” spasi, saya akan menggunakan fungsi CHAR_LENGTH(). Fungsi bawaan MySQL ini berfungsi ini menghitung jumlah karakter. Berikut hasilnya:

```
SELECT CHAR_LENGTH(a), CHAR_LENGTH(b) FROM contoh_char_1;
+-----+-----+
| CHAR_LENGTH(a) | CHAR_LENGTH(b) |
+-----+-----+
|           5 |           5 |
|           1 |           5 |
+-----+-----+
```

Di baris pertama berisi angka 5 untuk kolom a, dan 5 untuk kolom b. Ini adalah jumlah karakter yang ada di dalam string 'dunia'. Tidak ada masalah.

Pada baris kedua kolom a berisi angka 1. Artinya, string 'a ' yang diinput ke dalam kolom a terhitung sebanyak 1 karakter. 4 karakter spasi setelah huruf a otomatis dihapus. Inilah efek dari tipe data CHAR.

Sedangkan untuk kolom b tetap terhitung sebanyak 5 karakter (1 huruf a + 4 spasi), karena kolom b menggunakan tipe data VARCHAR.

Perbedaan perlakuan seperti ini bisa menjadi masalah dalam kasus-kasus tertentu. Terutama data yang ‘sensitif’ terhadap spasi, seperti password.

Bagaimana kalau kita coba input *escape character* ke dalam tipe data CHAR dan VARCHAR? Tidak masalah:

```
INSERT INTO contoh_char_1 VALUES ('do\'a', 'do\'a');
```

```
SELECT * FROM contoh_char_1;
+---+---+
| a | b |
+---+---+
| dunia | dunia |
| a | a |
| do'a | do'a |
+---+---+
```

Kedua kolom dari tabel contoh_char_1 hanya bisa menampung 5 karakter. Mari test dengan menginput string yang lebih panjang dari 5 karakter:

```
INSERT INTO contoh_char_1 VALUES ('duniailkom', 'duniailkom');
```

```
SHOW WARNINGS;
+-----+-----+
| Level | Code | Message |
+-----+-----+
| Warning | 1265 | Data truncated for column 'a' at row 1 |
| Warning | 1265 | Data truncated for column 'b' at row 1 |
+-----+-----+
```

```
SELECT * FROM contoh_char_1;
+---+---+
| a | b |
+---+---+
| dunia | dunia |
| a | a |
| do'a | do'a |
| dunia | dunia |
+---+---+
```

MySQL akan mengeluarkan warning: *Data truncated for column* dan otomatis memotong string tersebut.

8.8 Tipe Data Binary dan Varbinary

Pada dasarnya tipe data BINARY dan VARBINARY sama dengan CHAR dan VARCHAR, perbedaannya hanya pada struktur penyimpanan dalam MySQL, dimana tipe data binary akan disimpan secara biner (bit per bit), bukan secara karakter seperti CHAR. Sederhananya, hal ini akan berefek pada case-sensitif data (perbedaan penggunaan huruf besar dan huruf kecil).

Pada tipe data CHAR, jika kita menyimpan data 'A', maka secara internal MySQL akan menyimpannya sebagai karakter 'A', dimana 'A' akan sama dengan 'a'. Namun untuk tipe data

BINARY, ‘A’ akan disimpan sebagai bit dari ‘A’, yaitu 65, dan akan berbeda dengan ‘a’, dimana nilai biner-nya 97.

Format query tipe data BINARY dan VARBINARY adalah sebagai berikut:

```
BINARY [(M)]
VAR BINARY [(M)]
```

Sama seperti tipe data CHAR dan VARCHAR, M adalah jumlah karakter yang dialokasikan oleh MySQL. Misalkan nilai M = 5, MySQL akan menyediakan 5 karakter untuk kolom tersebut. Nilai M maksimal 255 karakter untuk BINARY, dan 65,535 karakter untuk VARBINARY. Jika nilai M tidak di nyatakan, nilai defaultnya M=1.

BINARY disimpan dalam ukuran yang tetap (sama seperti CHAR), sedangkan VARBINARY akan berubah sesuai ukuran data (sama seperti VARCHAR).

Berikut contoh pembuatan tabel untuk tipe data BINARY dan VARBINARY :

```
CREATE TABLE contoh_binary_1 (
    a BINARY(5),
    b VARBINARY(5)
);

INSERT INTO contoh_binary_1 VALUES ('dunia','dunia');

SELECT * FROM contoh_binary_1;
+-----+-----+
| a     | b     |
+-----+-----+
| dunia | dunia |
+-----+-----+

SELECT * FROM contoh_binary_1 where a = 'dunia';
+-----+-----+
| a     | b     |
+-----+-----+
| dunia | dunia |
+-----+-----+

SELECT * FROM contoh_binary_1 where a = 'DUNIA';
Empty set (0.00 sec)
```

Perhatikan bahwa pada query terakhir tidak menampilkan hasil apa-apa, karena dalam BINARY, string ‘dunia’ tidak sama dengan ‘DUNIA’.

8.9 Tipe Data Text

Untuk data string yang lebih besar, MySQL menyediakan tipe data TEXT yang terdiri dari 4 jenis: TINYTEXT, TEXT, MEDIUMTEXT, dan LONGTEXT. Maksimal ukuran masing-masing tipe data dapat dilihat dari tabel dibawah ini:

Tipe Data	Ukuran Maksimum	Jumlah Karakter Maksimum
TINYTEXT	255 byte	255
TEXT	65.535 byte (64 KB)	6.5535
MEDIUMTEXT	16.777.215 byte (16MB)	16.777.215
LONGTEXT	4.294.967.295 (4GB)	4.294.967.295

Mekanisme penyimpanan data untuk tipe data TEXT sama seperti VARCHAR, dimana MySQL hanya akan menyimpan jumlah karakter saja.

Misalkan kita definisikan suatu kolom sebagai LONGTEXT, dan hanya diisi 100 karakter, ukuran penyimpanan yang digunakan juga 100 byte, bukan 4GB.

Berikut contoh query membuat tabel dengan tipe data TEXT :

```
CREATE TABLE contoh_text_1 (
    a TINYTEXT,
    b TEXT,
    c LONGTEXT
);

DESC contoh_text_1;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| a     | tinytext  | YES  |     | NULL    |        |
| b     | text       | YES  |     | NULL    |        |
| c     | longtext   | YES  |     | NULL    |        |
+-----+-----+-----+-----+-----+

INSERT INTO contoh_text_1 VALUES ('duniailkom', 'duniailkom', 'duniailkom.com');

SELECT * FROM contoh_text_1;
+-----+-----+-----+
| a      | b      | c      |
+-----+-----+-----+
| duniailkom | duniailkom | duniailkom.com |
+-----+-----+-----+
```

8.10 Tipe Data Blob

Tipe data BLOB adalah tipe data versi binary dari TEXT, dimana karakter akan disimpan dalam bit. Efeknya, untuk karakter huruf, huruf besar dan kecil akan dibedakan ('A' tidak sama dengan 'a'). Sama seperti TEXT, BLOB juga memiliki beberapa tipe: TINYBLOB, BLOB, MEDIUMBLOB, dan LONGBLOB.

Ukuran serta jangkauan dari BLOB sama seperti TEXT, seperti pada tabel berikut ini:

Tipe Data	Ukuran Maksimum	Jumlah Karakter Maksimum
TINYBLOB	255 byte	255
BLOB	65.535 byte (64 KB)	65.535
MEDIUMBLOB	16.777.215 byte (16MB)	16.777.215
LONGBLOB	4.294.967.295 (4GB)	4.294.967.295

Karena sifatnya yang tersimpan secara binary, tipe data BLOB umumnya digunakan untuk menyimpan data multimedia, seperti gambar dan musik ke dalam tabel MySQL. Berikut contoh penggunaan tipe data BLOB dalam pembuatan tabel:

```

CREATE TABLE contoh_blob_1 (
    a TINYBLOB,
    b BLOB,
    c LONGBLOB
);

DESCRIBE contoh_blob_1;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| a     | tinyblob  | YES  |     | NULL    |        |
| b     | blob       | YES  |     | NULL    |        |
| c     | longblob   | YES  |     | NULL    |        |
+-----+-----+-----+-----+-----+

INSERT INTO contoh_blob_1 VALUES ('duniailkom','duniailkom','duniailkom.com');

SELECT * FROM contoh_blob_1;
+-----+-----+-----+
| a     | b     | c     |
+-----+-----+-----+
| duniailkom | duniailkom | duniailkom.com |
+-----+-----+-----+

```

8.11 Tipe Data Date

Tipe data DATE adalah tipe data khusus untuk menyimpan tanggal (*date*), waktu (*time*), dan gabungan tanggal dan waktu (*datetime*). Tipe data DATE bisa ditulis menggunakan tanda kutip

maupun tanpa tanda kutip, dengan aturan yang akan kita pelajari sesaat lagi.

Tipe data date terdiri dari beragam jenis yang dibedakan berdasarkan format penyimpanan data, yakni: DATE, TIME, DATETIME, TIMESTAMP, dan YEAR.

Berikut detail perbedaan dan format penulisan tipe data DATE MySQL:

Tipe Data	Jangkauan	Ukuran	Zero Value
DATE	1000-01-01 to 9999-12-31	3 byte	0000-00-00
DATETIME	1000-01-01 00:00:01 to 9999-12-31 23:59:59	8 byte	0000-00-00 00:00:00
TIMESTAMP	1970-01-01 00:00:00 to 2038-01-18 22:14:07	4 byte	0000-00-00 00:00:00
TIME	838:59:59 to 838:59:58	3 byte	00:00:00
YEAR(2)	00 to 99	1 byte	00
YEAR(4)	1901 to 2155	1 byte	0000

Setiap tipe data diatas memiliki format inputan yang berbeda-beda. Satu tipe data bisa memiliki lebih dari 1 format penulisan. Tabel berikut merangkum cara penulisan tipe data DATE MySQL:

Tipe Data	Fomat Input
DATETIME	'CCYY-MM-DD hh:mm:ss' CCYYMMDDhhmmss
TIMESTAMP	'YY-MM-DD hh:mm:ss' 'CCYYMMDDhhmmss' 'YYMMDDhhmmss' CCYYMMDDhhmmss
DATE	'CCYY-MM-DD' 'YY-MM-DD' 'CCYYMMDD' 'YYMMDD' CCYYMMDD YYMMDD
TIME	'hh:mm:ss' 'hhmmss' hhmmss
YEAR	'CCYY' 'YY' CCYY YY

Penjelasan format:

- **CCYY** : inputan tahun, dimana YY berupa tahun 2 digit, seperti 98, 78, dan 00, sedangkan untuk CCYY adalah tahun dengan 4 digit, seperti 2001, 1987, 2012. Untuk tahun dengan 2 digit, MySQL akan mengkonversinya dengan aturan 70-99 menjadi 1970-1999 dan 00-69 menjadi 2000-2069.
- **MM**: bulan dalam format dua digit, seperti 05, 07 dan 12.

- **DD**: tanggal dalam format dua digit, seperti 14, 06 dan 30.
- **hh**: jam dalam format 2 digit, seperti 06, 09 dan 12.
- **mm**: menit, dalam format 2 digit, seperti 15, 45 dan 59.
- **ss**: detik, dalam format 2 digit, seperti 10, 40 dan 57..

Jika MySQL tidak dapat membaca format inputan atau data tidak tersedia, nilai DATE akan diisi sesuai dengan kolom *Zero Value*.

Dari tabel diatas, terlihat juga bahwa ada beberapa format yang harus ditulis di dalam tanda kutip, dan ada yang tidak.

Jika data DATE diinput dengan nilai yang berisi karakter tambahan bukan angka seperti spasi, tanda pemisah (-) atau tanda titik dua (:), nilai tersebut **harus ditulis dalam tanda kutip**, contoh: '2017-07-22'. Apabila data tersebut hanya terdiri dari angka saja, bisa ditulis tanpa tanda kutip, seperti 20170722.

Mari kita coba membuat tabel dengan tipe data DATE:

```
CREATE TABLE contoh_date_1 (
    a DATE,
    b TIME,
    c DATETIME,
    d TIMESTAMP,
    e YEAR
);
```

```
DESC contoh_date_1;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default           | Extra          |
+-----+-----+-----+-----+-----+
| a     | date       | YES  |     | NULL              |                |
| b     | time       | YES  |     | NULL              |                |
| c     | datetime   | YES  |     | NULL              |                |
| d     | timestamp  | NO   |     | CURRENT_TIMESTAMP | on update ... |
| e     | year(4)   | YES  |     | NULL              |                |
+-----+-----+-----+-----+-----+
```

Disini saya membuat tabel contoh_date_1 dengan 5 kolom dimana masing-masingnya diisi dengan tipe data DATE yang berbeda-beda.

Dari hasil query DESC contoh_date_1, terlihat kolom e memiliki tipe YEAR(4). Artinya kolom ini akan menyimpan data tahun dalam 4 digit angka.

Kita dapat secara langsung mendefinisikan YEAR(2) jika menginginkan data tahun dengan 2 digit. Namun hal ini tidak disarankan karena bisa bermakna ganda (ambigu). Ketika diinput angka 26, apakah itu artinya tahun 1926 atau 2026? Meskipun MySQL memiliki aturan akan mengkonversinya menjadi 2026.

Selain itu mulai dari MySQL 5.6.6, YEAR(2) dinyatakan *deprecated* dan mungkin akan dihapus di versi yang akan datang.

Saya akan coba input beberapa data ke tabel contoh_date_1:

```
INSERT INTO contoh_date_1 VALUES (
    '2017-07-22',
    '18:45:23',
    '2017-07-22 18:45:23',
    '2017-07-22 18:45:23',
    '2017'
);

INSERT INTO contoh_date_1 VALUES (
    20170722,
    184523,
    20170722184523,
    20170722184523,
    2017
);

SELECT * FROM contoh_date_1;
```

a	b	c	d	e
2017-07-22	18:45:23	2017-07-22 18:45:23	2017-07-22 18:45:23	2017
2017-07-22	18:45:23	2017-07-22 18:45:23	2017-07-22 18:45:23	2017

Saya menginput 2 baris ke tabel contoh_date_1. Query INSERT pertama diinput dengan tanda kutip karena terdapat karakter pemisah di dalam data, yakni '18:45:23'. Di query kedua saya tidak perlu menulis tanda kutip karena tidak data yang diinput hanya berupa angka (tidak ada karakter tambahan).

MySQL memiliki berbagai fungsi bawaan untuk tipe data DATE, salah satunya fungsi NOW() untuk mengambil tanggal server pada saat ini.

```
INSERT INTO contoh_date_1 VALUES (NOW(), NOW(), NOW(), NOW(), NOW());
```

Query OK, 1 row affected, 3 warnings (0.11 sec)

```
SHOW WARNINGS;
```

Level	Code	Message
Note	1265	Data truncated for column 'a' at row 1
Note	1265	Data truncated for column 'b' at row 1
Warning	1265	Data truncated for column 'e' at row 1

```
+-----+-----+-----+-----+
| SELECT * FROM contoh_date_1;
+-----+-----+-----+-----+
| a      | b      | c      | d      | e      |
+-----+-----+-----+-----+
| 2017-07-22 | 18:45:23 | 2017-07-22 18:45:23 | 2017-07-22 18:45:23 | 2017 |
| 2017-07-22 | 18:45:23 | 2017-07-22 18:45:23 | 2017-07-22 18:45:23 | 2017 |
| 2017-07-23 | 07:45:09 | 2017-07-23 07:45:09 | 2017-07-23 07:45:09 | 2017 |
+-----+-----+-----+-----+
```

Fungsi NOW() menginput waktu saat ini ke dalam tabel contoh_date_1. Query INSERT diatas memiliki 3 warning karena fungsi NOW() menghasilkan waktu dalam satuan date dan time sekaligus. Akan tetapi kolom a, b dan e hanya bisa menampung sebagian nilai ini (time saja, atau waktu saja), sehingga terdapat data yang dipotong (*truncated*) dan menjadi warning.

Pemotongan data seperti ini juga berkaitan dengan **SQL MODE**. Jika MySQL di konfigurasi dengan “*strict mode*”, data yang berlebih seperti diatas akan menghasilkan error dan query INSERT gagal dijalankan.

Mulai dari MySQL versi 5.6.4 ke atas, tipe data TIME, DATETIME, dan TIMESTAMP mendukung fitur tambahan *fractional seconds precision (fps)*. Fitur fps ini memungkinkan kita menyimpan ketelitian hingga 1 *microsecond*, yakni 0,000001 detik.

Untuk membuat kolom dengan fps, tambahkan angka antara 1 - 6 ke dalam penulisan tipe data TIME, DATETIME, dan TIMESTAMP.

Jika ditulis sebagai TIME(2), artinya tipe data TIME mendukung hingga 2 tempat desimal setelah 1 detik, seperti 11:02:36.12. Jika ditulis sebagai TIME(6), bisa mendukung hingga 6 tempat desimal, seperti 11:02:36.123456. Berikut contoh penggunaannya:

```
CREATE TABLE contoh_date_2 (
    a TIME(3),
    b DATETIME(4),
    c DATETIME(6)
);
```

```
INSERT INTO contoh_date_2 VALUES (
    '11:02:36.123',
    '2017-07-23 11:02:36.2334',
    '2017-07-23 11:02:36.999999'
);
```

```
SELECT * FROM contoh_date_2;
```

```
+-----+-----+-----+
| a      | b      | c      |
+-----+-----+-----+
| 11:02:36.123 | 2017-07-23 11:02:36.2334 | 2017-07-23 11:02:36.999999 |
+-----+-----+-----+
```

Untuk menginput waktu saat ini dengan nilai **fps**, fungsi NOW() tidak bisa dipakai. Karena fungsi NOW() hanya memiliki ketelitian hingga satuan detik (tidak sampai ke micro detik). Untuk menginput data yang lebih detail, tersedia fungsi CURTIME().

Fungsi CURTIME() diinput dengan 1 argumen yang berisi seberapa detail angka yang ingin dihasilkan, yakni sama dengan nilai **fps**. Untuk menggenerate hingga satuan micro detik, gunakan fungsi CURTIME(6).

```
INSERT INTO contoh_date_2 VALUES (CURTIME(3),CURTIME(4),CURTIME(6));
```

```
SELECT * FROM contoh_date_2;
```

a	b	c
11:02:36.123	2017-07-23 11:02:36.2334	2017-07-23 11:02:36.999999
11:15:19.536	2017-07-23 11:15:19.5363	2017-07-23 11:15:19.536327

Ketelitian hingga microsecond seperti ini biasanya tidak selalu kita perlukan.

8.12 Tipe Data Enum

Selain tipe data standar seperti Numerik, String dan Date, MySQL juga menyediakan tipe data khusus yang dibuat dari kumpulan data yang kita definisikan sendiri. Tipe data bentukan ini terdiri dari ENUM dan SET. Kita akan bahas tipe data ENUM terlebih dahulu.

Tipe data ENUM merupakan tipe data bentukan yang hanya bisa diisi dengan 1 nilai yang sudah kita tentukan sebelumnya. Pilihan ini dapat berisi 1 hingga 65.535 pilihan string.

Contoh penggunaan ENUM misalnya pada kolom jenis kelamin. Untuk kolom ini pilihannya hanya ada dua: laki-laki atau perempuan. Agar data yang diinput ke dalam tabel lebih teratur, kita bisa membuat aturan bahwa kolom tersebut hanya bisa diisi 2 nilai itu saja dan tidak boleh dengan nilai lain.

Atau bisa juga untuk membuat daftar pilihan jurusan di sebuah universitas. Pilihan ini harus sudah tersedia sebelumnya. Kita bisa menggunakan tipe data ENUM untuk memastikan bahwa jurusan yang dipilih adalah jurusan yang telah tersedia saja.

Format dasar penulisan tipe data ENUM adalah sebagai berikut:

```
ENUM ('pilihan_1', 'pilihan_2', 'pilihan_3', ...)
```

Berikut contoh penggunaannya:

```

CREATE TABLE contoh_enum_1 (
    jur ENUM('Ilmu Komputer', 'Ekonomi', 'MIPA', 'Kedokteran')
);
Query OK, 0 rows affected (0.29 sec)

DESC contoh_enum_1;
+-----+-----+-----+
| Field | Type          | Null |
+-----+-----+-----+
| jur   | enum('Ilmu Komputer', 'Ekonomi', 'MIPA', 'Kedokteran') | YES |
+-----+-----+-----+

INSERT INTO contoh_enum_1 VALUES ('Ilmu Komputer');
INSERT INTO contoh_enum_1 VALUES ('Kedokteran');

SELECT * FROM contoh_enum_1;
+-----+
| jur      |
+-----+
| Ilmu Komputer |
| Kedokteran   |
+-----+

```

Saya membuat tabel contoh_enum_1 yang terdiri dari 1 kolom: jur. Kolom jur ini didefinisikan sebagai enum('Ilmu Komputer', 'Ekonomi', 'MIPA', 'Kedokteran'). Artinya, kolom jur hanya bisa diisi dengan salah satu nilai dari 'Ilmu Komputer', 'Ekonomi', 'MIPA', dan 'Kedokteran', dan tidak bisa diisi nilai lain.

Cara pengisian data untuk kolom ENUM sama seperti tipe data string. Atau sebagai alternatif, kolom ENUM juga bisa diinput dengan nomor urut. Angka 1 untuk pilihan pertama, angka 2 untuk pilihan kedua, dst.

```

INSERT INTO contoh_enum_1 VALUES (1);
INSERT INTO contoh_enum_1 VALUES (3);

SELECT * FROM contoh_enum_1;
+-----+
| jur      |
+-----+
| Ilmu Komputer |
| Kedokteran   |
| Ilmu Komputer |
| MIPA        |
+-----+

```

Fitur seperti ini juga bisa digunakan untuk pencarian, seperti contoh berikut:

```
SELECT * FROM contoh_enum_1 WHERE jur = 3;
+----+
| jur |
+----+
| MIPA |
+----+
```

Karena jurusan 'MIPA' adalah pilihan ke-3, query SELECT diatas sama artinya dengan `SELECT * FROM contoh_enum_1 WHERE jur = 'MIPA'`.

Bagaimana jika diinput dengan nilai yang tidak ada di daftar ENUM? Mari kita coba:

```
INSERT INTO contoh_enum_1 VALUES ('Hukum');
Query OK, 1 row affected, 1 warning (0.07 sec)

SHOW WARNINGS;
+-----+-----+-----+
| Level | Code | Message           |
+-----+-----+-----+
| Warning | 1265 | Data truncated for column 'jur' at row 1 |
+-----+-----+-----+

SELECT * FROM contoh_enum_1;
+-----+
| jur      |
+-----+
| Ilmu Komputer |
| Kedokteran   |
| Ilmu Komputer |
| MIPA        |
|             |
+-----+
```

Saya menginput nilai 'Hukum' ke dalam kolom jur. Hasilnya terdapat *warning*: *Data truncated for column 'jur' at row 1*.

Akan tetapi di tabel contoh_enum_1 tetap masuk data kosong di baris terakhir. Ini adalah pengaturan default dari MariaDB bawaan XAMPP. Jika kita ingin agar query seperti ini menghasilkan error dan tidak menginput data apapun, bisa mengubah pengaturan SQL MODE menjadi '*strict mode*'.

8.13 Tipe Data Set

Tipe data SET mirip seperti ENUM, yakni sebuah tipe data bentukan yang pilihannya sudah di definisikan terlebih dahulu. Namun berbeda dengan ENUM yang mengharuskan kita memilih 1

pilihan saja, kolom dengan tipe data SET bisa diisi lebih dari 1 pilihan. Daftar pilihan untuk tipe data SET dibatasi antara 1 hingga 64.

Menggunakan analogi dari form HTML, tipe data SET mirip seperti **check box**. Kita bisa memilih lebih dari 1 pilihan yang ada, bisa 2, 3 atau lebih. Sedangkan tipe data ENUM mirip seperti **radio button**, dimana kita hanya bisa memilih 1 pilihan saja.

Contoh penggunaan tipe data SET adalah untuk pilihan hobi. Bisa saja seseorang memiliki lebih dari 1 hobi, namun tetap dibatasi dari pilihan yang ada (tidak bisa menginput hobi baru).

Format dasar penulisan tipe data SET sama seperti ENUM:

```
SET ('pilihan_1', 'pilihan_2', 'pilihan_3', ...)
```

Berikut contoh penggunaannya:

```
CREATE TABLE contoh_set_1 (
    hob SET('Membaca', 'Menulis', 'Menggambar', 'Main Musik')
);

DESC contoh_set_1;
+-----+-----+-----+-----+
| Field | Type | Null | Key |
+-----+-----+-----+-----+
| hob   | set('Membaca', 'Menulis', 'Menggambar', 'Main Musik') | YES | |
+-----+-----+-----+-----+

INSERT INTO contoh_set_1 VALUES ('Membaca');
INSERT INTO contoh_set_1 VALUES ('Membaca,Main Musik');
INSERT INTO contoh_set_1 VALUES ('Menggambar,Main Musik');

SELECT * FROM contoh_set_1;
+-----+
| hob |
+-----+
| Membaca |
| Membaca,Main Musik |
| Menggambar,Main Musik |
+-----+
```

Jika data yang diinput lebih dari 1, tidak boleh ada spasi diantara pilihan. Penulisan yang benar adalah VALUES ('Menggambar,Main Musik') bukan VALUES ('Menggambar, Main Musik'). Perhatikan bahwa terdapat 1 spasi antara 'Menggambar,(spasi)Main Musik'. Kesalahan seperti ini bisa membuat pusing jika tidak mengetahui letak error-nya.

Sama seperti tipe data ENUM, SET juga bisa diisi dengan angka. Namun perhitungannya lebih kompleks.

Untuk tipe data SET, setiap pilihan memiliki urutan dari kelipatan angka biner. Angka yang diinput sebagai pilihan SET merupakan kombinasi dari total angka biner ini.

Perhatikan tabel berikut:

'Membaca'	'Menulis'	'Menggambar'	'Main Musik'
2^0	2^1	2^2	2^3
1	2	4	8

Pilihan pertama 'Membaca' memiliki nilai 2^0 , atau 1 dalam angka desimal. Pilihan kedua 'Menulis' memiliki nilai 2^1 atau 2 dalam angka desimal, demikian seterusnya.

Jika saya menjalankan query `INSERT INTO contoh_set_1 VALUES (2)`, string yang diinput adalah 'Menulis'. Karena angka 2 kepunyaan string kedua.

Bagaimana kalau `INSERT INTO contoh_set_1 VALUES (3)`? Angka 3 di dapat dari penambahan $1 + 2$. Artinya data yang diinput ada 2, yakni 'Membaca' dan 'Menulis'.

Berikut contoh prakteknya:

```
INSERT INTO contoh_set_1 VALUES (2);
INSERT INTO contoh_set_1 VALUES (3);
INSERT INTO contoh_set_1 VALUES (9);
```

```
SELECT * FROM contoh_set_1;
+-----+
| hob |
+-----+
| Membaca |
| Membaca,Main Musik |
| Menggambar,Main Musik |
| Menulis |
| Membaca,Menulis |
| Membaca,Main Musik |
+-----+
```

Dapatkah anda menjelaskan query terakhir: `INSERT INTO contoh_set_1 VALUES (9)`? Angka 9 didapat dari penambahan pilihan 1 (nilai: 1) dengan pilihan ke 4 (nilai: 8). Karena itulah hasilnya 'Membaca' dan 'Main Musik'.

Sama seperti tipe data ENUM, jika data yang diinput tidak berasal dari pilihan yang ada, akan menghasilkan warning dan data tersebut digantikan 1 baris kosong. Pengaturan ini bisa diubah dari SQL MODE.

8.14 Latihan Tipe Data

Sampai disini kita telah membahas berbagai tipe data yang ada di dalam MySQL, yang bisa dikelompokkan menjadi 3 bagian:

- **Number:** TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT, BOOL, DECIMAL, FLOAT, DOUBLE dan BIT.
- **String:** CHAR, VARCHAR, TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT, BINARY, VARBINARY, TINYBLOB, BLOB, MEDIUMBLOB, LONGBLOB, ENUM dan SET.
- **Date:** DATE, TIME, DATETIME, TIMESTAMP dan YEAR.

Tipe data ENUM dan SET masuk ke kelompok string karena dianggap sebagai kumpulan dari tipe data string.

Agar bisa lebih memahami penggunaan tipe data, tidak ada salahnya kita latihan sejenak untuk merancang tabel MySQL:

Tabel 'film'

Buatlah tabel untuk 5 film dengan [pendapatan tertinggi](#)¹:

No	Judul	Pendapatan (US \$)	Tahun produksi
1	Avatar	2,787,965,087	2009
2	Titanic	2,186,772,302	1997
3	Star Wars: The Force Awakens	2,068,223,624	2015
4	Jurassic World	1,671,713,208	2015
5	The Avengers	1,518,812,988	2012

Tabel 'luas_negara'

Buatlah tabel untuk 5 negara dengan [luas wilayah terbesar](#)²:

No	Negara	Luas Total (km ²)	Luas Daratan (km ²)	Luas Lautan(km ²)	% Luas Lautan
001	Rusia	17,098,246	16,377,742	720,500	4.21
002	Kanada	9,984,670	9,093,507	891,163	8.93
003	Tiongkok (RRC)	9,596,961	9,326,410	270,550	2.82
004	Amerika Serikat	9,525,067	9,147,593	377,424	3.96
005	Brazil	8,515,767	8,460,415	55,352	0.65

Silahkan anda rancang 2 tabel diatas, termasuk menginput data yang ada.

Supaya seragam, nama kolom ditulis dalam huruf kecil semua, jika mengandung spasi bisa diganti dengan underscore (_). Keterangan kolom seperti (US \$) dan (km²) boleh tidak ditulis. Tanda koma (,) dalam angka adalah pemisah ribuan. Angka 17,098,246 diinput sebagai 17098246.

¹https://en.wikipedia.org/wiki/List_of_highest-grossing_films

²https://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_area

8.15 Jawaban Latihan Tipe Data

Baik, untuk menyamakan jawaban atau apabila terdapat kendala, berikut query yang saya pakai:

Tabel 'film'

Query pembuatan tabel:

```
CREATE TABLE film (
    no TINYINT, judul VARCHAR(50),
    pendapatan BIGINT,
    tahun_produksi YEAR(4)
);
```

- Kolom `no` menggunakan `TINYINT` karena isian tabel `film` hanya 5 buah. Tapi jika ada kemungkinan baris tabel mencapai ratusan, akan lebih baik mengubahnya menjadi `SMALLINT`.
- Kolom `judul` menggunakan `VARCHAR(100)` karena saya berasumsi judul film tidak ada yang lebih dari 100 karakter.
- Kolom `pendapatan` menggunakan `BIGINT` karena terdapat pendapatan film sudah lebih dari 2 miliar.
- Kolom `tahun_produksi` menggunakan `YEAR(4)` sesuai dengan data tahun yang akan diisi (4 digit).

Query pengisian tabel:

```
INSERT INTO film VALUES (1, 'Avatar', 2787965087, 2009);
INSERT INTO film VALUES (2, 'Titanic', 2186772302, 1997);
INSERT INTO film VALUES (3, 'Star Wars: The Force Awakens', 2068223624, 2015);
INSERT INTO film VALUES (4, 'Jurassic World', 1671713208, 2015);
INSERT INTO film VALUES (5, 'The Avengers', 1518812988, 2012);
```

Tidak ada hal yang baru pada query diatas. Saya menggunakan 5 perintah `INSERT` untuk menginput data. Anda boleh juga langsung menginput kelimanya dengan 1 query `INSERT`.

Tabel 'luas_negara'

Query pembuatan tabel:

```
CREATE TABLE luas_negara (
    no TINYINT(3) ZEROFILL,
    negara VARCHAR(50),
    luas_total INT,
    luas_daratan INT,
    luas_lautan INT,
    persentase_luas_lautan DECIMAL(4,2)
);
```

- Kolom no menggunakan TINYINT(3) ZEROFILL, dimana atribut ZEROFILL berfungsi untuk menambah angka nol di sisi kiri selama belum cukup 3 digit.
- Kolom negara menggunakan VARCHAR(50) karena saya berasumsi nama negara tidak ada yang lebih dari 50 karakter.
- Kolom luas_total, luas_daratan, dan luas_lautan menggunakan INT karena luas wilayah maksimal hanya 17 juta, artinya masih bisa ditampung oleh tipe data INT.
- Kolom persentase_luas_lautan menggunakan DECIMAL(4,2) karena berbentuk pecahan dengan 2 digit di belakang koma.

Query pengisian tabel:

```
INSERT INTO luas_negara VALUES
(1, 'Rusia', 17098246, 16377742, 720500, 4.21),
(2, 'Kanada', 9984670, 9093507, 891163, 8.93),
(3, 'Tiongkok (RRC)', 9596961, 9326410, 270550, 2.82),
(4, 'Amerika Serikat', 9525067, 9147593, 377424, 3.96),
(5, 'Brazil', 8515767, 8460415, 55352, 0.65);
```

Kali ini saya menginput 5 data dalam 1 buah query INSERT.

Analisis Perancangan Tabel 'luas_negara'

Sedikit diskusi tentang perancangan tabel luas_negara. Sebenarnya kita tidak perlu membuat kolom luas_total dan persentase_luas_lautan. Kedua kolom ini bisa di-generate dari kolom luas_daratan dan luas_lautan.

Kolom luas_total bisa dihasilkan dari luas_daratan + luas_lautan. Sedangkan kolom persentase_luas_lautan didapat dari perhitungan luas_lautan / luas_total * 100.

Dengan menghapus kolom luas_total dan persentase_luas_lautan, data di dalam tabel menjadi lebih spesifik. Proses input dapat di lakukan dengan lebih cepat sekaligus meminimalisir kesalahan.

Menggunakan query SELECT, kita bisa menggenerate kolom luas_total dan persentase_luas_lautan dengan mudah, termasuk seandainya butuh kolom persentase_luas_daratan.

Akan tetapi, teknik seperti ini juga memiliki kekurangan. Setiap kali query SELECT di panggil, MySQL Server butuh memproses data terlebih dahulu. Berbeda jika datanya sudah tersedia dan tinggal di tampilkan. Solusinya, kita bisa menggunakan fitur lanjutan MySQL seperti **view** dan **temporary table**. Kedua materi ini akan kita pelajari dalam bab tersendiri.

Tipe Data JSON

Sebagai informasi tambahan, di MySQL versi 5.7.8 ke atas, terdapat 1 buah tipe data baru: **JSON**. Tipe data ini khusus digunakan untuk menyimpan notasi **JSON** (JavaScript Object Notation). JSON sering dipakai untuk sarana komunikasi antar website, yakni membuat sebuah **API** (Application programming interface).

Contoh query pembuatan dan pengisian data adalah sebagai berikut:

```
CREATE TABLE contoh_json (my_api JSON);
INSERT INTO contoh_json ( '{"nama": "Andi", "umur": 23}' )
```

Tipe data JSON ini baru tersedia di versi MySQL terbaru. Di dalam MariaDB bawaan XAMPP yang saya pakai, tipe data JSON masih belum di dukung. Referensi lengkapnya bisa akses kesini: [The JSON Data Type^a](#).

^a<https://dev.mysql.com/doc/refman/5.7/en/json.html>

Dalam bab kali ini kita telah membahas berbagai tipe data di dalam MySQL. Pengetahuan tentang tipe data sangat penting untuk merancang tabel yang efisien. Berikutnya, kita akan masuk ke materi tentang **Atribut Tipe Data**.

9. Atribut Tipe Data

Dalam bab sebelumnya, kita telah mempelajari berbagai tipe data yang digunakan untuk pembuatan tabel. Kali ini saya akan membahas perintah tambahan yang berfungsi untuk memodifikasi tipe data tersebut. Apakah itu menambah kemampuan, mengurangi, hingga membatasi nilai yang bisa ditampung.

Pada pembahasan tentang tipe data *numeric*, kita sudah berkenalan dengan 2 atribut, yakni `UNSIGNED` dan `ZEROFILL`. Selain itu, masih ada beberapa atribut lain.

Terdapat atribut yang bisa ditulis untuk seluruh tipe data (bersifat generik). Dan ada pula yang hanya ditujukan untuk tipe data tertentu saja (bersifat spesifik). Sebagai contoh, atribut `UNSIGNED` dan `ZEROFILL` hanya bisa digunakan untuk tipe data angka (numeric).

Atribut tipe data ditulis setelah nama kolom dan nama tipe data. Jika atribut tersebut lebih dari 1, dipisahkan dengan tanda spasi. Berikut format dasarnya:

```
nama_kolom tipe_kolom atribut1 atribut2 atribut3
```

Contoh:

```
total_penjualan INT(5) UNSIGNED ZEROFILL NOT NULL
```

Apa yang akan kita pelajari di bab ini kadang dikenal juga sebagai *constraints*, *column modifier* atau *column attributes*.



Kecuali dinyatakan lain, seluruh contoh tabel di input ke dalam database `belajar`. Saya berasumsi anda sudah menjalankan query `USE DATABASE belajar` agar bisa mengikuti materi dalam bab ini.

9.1 Atribut NULL

Dalam pemrograman, nilai `NULL` adalah sebutan untuk ‘tidak ada data’. `NULL` ini berbeda dengan angka 0 maupun string kosong ‘’. Angka nol tetap mewakili sebuah angka (tipe data number), begitu pula dengan string kosong yang tetap berupa tipe data string.

Secara default, hampir seluruh kolom boleh diisi dengan nilai `NULL`, kecuali dinyatakan lain. Jika terdapat atribut `NULL` dalam pendefinisian kolom, artinya kolom tersebut boleh berisi nilai `NULL`.

Meskipun tidak ditulis, secara default (bawaan MySQL) menganggap setiap kolom tertulis atribut `NULL`. Kedua query berikut akan menghasilkan tabel yang sama:

- `CREATE TABLE contoh_null_1 (a INT, b VARCHAR(5), c DATE)`
- `CREATE TABLE contoh_null_1 (a INT NULL, b VARCHAR(5) NULL, c DATE NULL)`

Efeknya, ketiga kolom dalam tabel `contoh_null_1` boleh diisi nilai `NULL`. Berikut prakteknya:

```

CREATE TABLE contoh_null_1 (
    a INT NULL,
    b VARCHAR(5) NULL,
    c DATE
);

INSERT INTO contoh_null_1 VALUES (100, 'kamu', NULL );
INSERT INTO contoh_null_1 VALUES (NULL, 'saya', 20170817);
INSERT INTO contoh_null_1 VALUES (NULL,NULL,NULL);

SELECT * FROM contoh_null_1;
+---+---+---+
| a | b | c |
+---+---+---+
| 100 | kamu | NULL |
| NULL | saya | 2017-08-17 |
| NULL | NULL | NULL |
+---+---+---+

```

Setelah pembuatan tabel, saya menginput nilai NULL ke kolom a, b dan c. Terlihat dari hasil query SELECT bahwa setiap kolom bisa diinput nilai NULL.

Perhatikan juga pada saat pendefinisian tabel tidak terdapat atribut NULL untuk kolom c. Akan tetapi kolom c tetap bisa diisi nilai NULL. Dimana seolah-olah kolom c di definisikan sebagai c DATE NULL.

Untuk memeriksa apakah sebuah kolom bisa diisi dengan nilai NULL atau tidak, jalankan query DESCRIBE:

```

DESCRIBE contoh_null_1;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| a     | int(11)   | YES  |     | NULL    |       |
| b     | varchar(5) | YES  |     | NULL    |       |
| c     | date      | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+

```

Apabila di kolom Null, terdapat nilai YES, artinya field tersebut bisa diinput dengan nilai NULL. Atribut NULL bersifat generik dan bisa ditambahkan ke setiap kolom dengan tipe data apa saja.

9.2 Atribut NOT NULL

Atribut NOT NULL merupakan kebalikan dari atribut NULL. Jika sebuah kolom memiliki atribut NOT NULL, kita tidak bisa mengisi nilai NULL ke kolom tersebut.

Berikut percobaannya:

```

CREATE TABLE contoh_not_null_1 (
    a INT NOT NULL,
    b VARCHAR(5) NOT NULL,
    c DATE NULL
);

INSERT INTO contoh_not_null_1 VALUES (NULL, 'saya', 20170817);
ERROR 1048 (23000): Column 'a' cannot be null

INSERT INTO contoh_not_null_1 VALUES (100, NULL, 20170817);
ERROR 1048 (23000): Column 'b' cannot be null

INSERT INTO contoh_not_null_1 VALUES (100, 'kamu',NULL);
Query OK, 1 row affected (0.07 sec)

SELECT * FROM contoh_not_null_1;
+---+---+---+
| a | b | c |
+---+---+---+
| 100 | kamu | NULL |
+---+---+---+

```

Pada saat pendefinisian tabel contoh_not_null_1, saya menulis atribut NOT NULL ke kolom a dan b. Hasilnya akan keluar pesan error saat query INSERT mencoba menginput nilai NULL.

Pada kolom c, nilai NULL tetap bisa diisi karena tidak memiliki atribut NOT NULL, malah saya definisikan sebagai NULL yang artinya bisa diisi nilai NULL.

Sama seperti atribut NULL, query DESCRIBE juga bisa digunakan untuk memeriksa apakah sebuah kolom bisa diisi nilai NULL atau tidak:

```

DESCRIBE contoh_not_null_1;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| a     | int(11)   | NO  |     | NULL    |       |
| b     | varchar(5) | NO  |     | NULL    |       |
| c     | date       | YES |     | NULL    |       |
+-----+-----+-----+-----+-----+

```

Perhatikan nilai NO dalam kolom NULL untuk field a dan b. Artinya kedua kolom ini tidak bisa diisi nilai NULL.

Atribut NOT NULL biasa dipakai untuk membatasi bahwa sebuah kolom harus berisi nilai dan tidak boleh kosong (NULL). Atribut ini bersifat generik dan bisa ditambahkan ke setiap kolom dengan tipe data apa saja.

9.3 Atribut DEFAULT

Atribut default berfungsi untuk memberikan nilai default ke dalam sebuah kolom. Nilai default ini baru akan dipakai jika tidak ada nilai yang diinput ke kolom tersebut.

Dalam bab tentang “**Dasar Bahasa Query MySQL**”, kita telah melihat bahwa query `INSERT` bisa menginput data ke salah satu kolom saja (tidak harus semua kolom). Untuk kolom yang tidak diinput inilah, nilai default akan digunakan.

Sebenarnya, setiap kolom sudah memiliki nilai default bawaan, yakni `NULL`. Mari kita lihat contoh prakteknya:

```
CREATE TABLE contoh_default_1 (a INT, b VARCHAR(5));
INSERT INTO contoh_default_1 VALUES (25, 'hello');

SELECT * FROM contoh_default_1;
+---+---+
| a | b |
+---+---+
| 25 | hello |
+---+---+
```

Saya membuat tabel `contoh_default_1` dengan dua kolom: `a` sebagai `INT`, dan `b` sebagai `VARCHAR(5)`. Kemudian query `INSERT` dipakai untuk menginput nilai 25 ke dalam kolom `a` dan nilai ‘hello’ ke dalam kolom `b`. Tidak ada masalah.

Sekarang, bagaimana jika hanya salah satu kolom saja yang diinput?

```
INSERT INTO contoh_default_1 (a) VALUES (99);

SELECT * FROM contoh_default_1;
+---+---+
| a | b |
+---+---+
| 25 | hello |
| 99 | NULL |
+---+---+
```

Dalam query `INSERT` diatas, saya hanya mengisi kolom `a` dengan nilai 99. Bagaimana dengan kolom `b`? karena nilainya tidak diinput, MySQL akan menggunakan nilai default `NULL`.

Inilah ‘prilaku’ bawaan dari hampir setiap kolom dalam MySQL. Apabila atribut `DEFAULT` tidak ditulis, dianggap sebagai `NULL`.

Hal yang sama juga berlaku untuk kolom `a`:

```
INSERT INTO contoh_default_1 (b) VALUES ('hey');

SELECT * FROM contoh_default_1;
+-----+
| a    | b    |
+-----+
| 25   | hello |
| 99   | NULL |
| NULL | hey   |
+-----+
```

Terlihat bahwa secara bawaan, NULL menjadi nilai default dari sebuah kolom dalam MySQL. Sekarang, bagaimana cara mengubah nilai default NULL ini dengan nilai lain? fungsi dari atribut DEFAULT.

Format dasar penulisannya adalah sebagai berikut:

```
nama_kolom tipe_data_kolom DEFAULT nilai_default
```

Sebagai contoh praktek, saya akan buat tabel contoh_default_2 dimana kolom a memiliki nilai default 10 dan kolom b menggunakan nilai default ‘siang’:

```
CREATE TABLE contoh_default_2 (
  a INT DEFAULT 10,
  b VARCHAR(5) DEFAULT 'siang'
);
```

Mari kita coba dengan menginput salah satu kolom:

```
INSERT INTO contoh_default_2 (a) VALUES (99);
```

```
SELECT * FROM contoh_default_2;
+-----+
| a    | b    |
+-----+
| 99   | siang |
+-----+
```

```
INSERT INTO contoh_default_2 (b) VALUES ('hey');
```

```
SELECT * FROM contoh_default_2;
+-----+
| a    | b    |
+-----+
| 99   | siang |
| 10   | hey   |
+-----+
```

Pada query `INSERT` pertama, saya menginput hanya 1 nilai, yakni angka 99 ke kolom `a`. Bagaimana dengan kolom `b`? karena tidak memiliki nilai inputan, kolom `b` akan menggunakan nilai default ‘siang’. Begitu juga halnya ketika saya mengisi kolom `b` tanpa memberikan nilai ke kolom `a`. Kolom `a` akan menggunakan angka default 10.

Sesuai dengan namanya, atribut `default` hanya aktif saat tidak ada nilai inputan. Ketika dalam query `INSERT` diisi nilai, nilai default ini tidak lagi dipakai:

```
INSERT INTO contoh_default_2 VALUES (25, 'hello');
```

```
SELECT * FROM contoh_default_2;
+-----+
| a    | b    |
+-----+
| 99  | siang |
| 10  | hey   |
| 25  | hello |
+-----+
```

Untuk melihat nilai default dari sebuah kolom, kita bisa menggunakan query `DESC` / `DESCRIBE`:

```
DESC contoh_default_2;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| a     | int(11)   | YES  |     | 10      |       |
| b     | varchar(5) | YES  |     | siang   |       |
+-----+-----+-----+-----+-----+
```

Di bagian kolom `Default` akan terlihat nilai `default` dari suatu kolom.

Sebagai latihan tambahan, dapatkan anda memahami query pembuatan tabel berikut?

```
CREATE TABLE contoh_default_3 (
  a INT NOT NULL DEFAULT 10,
  b VARCHAR(5) NULL DEFAULT 'siang'
);
```

Tabel `contoh_default_3` diatas memiliki 2 kolom. Dimana saya menggunakan 2 buah atribut tipe data sekaligus:

- Kolom `a` `INT NOT NULL DEFAULT 10` artinya: kolom `a` bertipe `INT`, tidak bisa diisi nilai `NULL`, dan memiliki nilai default 10.
- Kolom `b` `VARCHAR(5) NULL DEFAULT 'siang'` artinya: kolom `b` bertipe string dengan jumlah maksimal karakter 5. Kolom ini bisa diisi nilai `NULL`, dan memiliki nilai default ‘siang’.

Selanjutnya bagaimana dengan query berikut?

```
CREATE TABLE contoh_default_4 (
    a INT NULL DEFAULT NULL,
    b VARCHAR(5) NULL DEFAULT NULL
);
```

Query pembuatan tabel diatas tampak sedikit aneh, tapi sebenarnya sudah kita pelajari.

Kolom a didefinisikan sebagai INT NULL DEFAULT NULL, artinya kolom a bertipe data INT, boleh berisi nilai NULL dan nilai default dari kolom ini juga NULL. Hal yang sama juga berlaku untuk kolom b.

Nilai NULL DEFAULT NULL merupakan nilai bawaan dari MySQL. Maksudnya, ketika sebuah kolom di definisikan tanpa tambahan atribut apa-apa, penulisan seperti inilah yang dipakai. Sehingga tiga query pembuatan tabel dibawah ini akan menghasilkan tabel yang sama:

- CREATE TABLE contoh_default_5 (a INT)
- CREATE TABLE contoh_default_5 (a INT DEFAULT NULL)
- CREATE TABLE contoh_default_5 (a INT NULL DEFAULT NULL)

Berikut percobaannya:

```
CREATE TABLE contoh_default_5 (a INT);
```

```
SHOW CREATE TABLE contoh_default_5 \G
*****
1. row ****
Table: contoh_default_5
Create Table: CREATE TABLE `contoh_default_5` (
  `a` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

Dari hasil query SHOW CREATE TABLE contoh_default_5 terlihat bahwa kolom a di definisikan sebagai a int(11) DEFAULT NULL, padahal pada query pembuatan tabel kita tidak mencantumkan atribut DEFAULT NULL secara tertulis.

Sama seperti atribut NULL dan NOT NULL, atribut DEFAULT juga bisa digunakan ke seluruh tipe data.

9.4 Atribut SIGNED

Atribut SIGNED khusus digunakan pada tipe data **number**. Fungsinya untuk menandakan bahwa tipe data number mendukung angka positif dan negatif. Atribut ini secara default sudah ‘melekat’ ke seluruh tipe data number sehingga tidak perlu dituliskan.

Kedua query CREATE TABLE berikut akan menghasilkan tabel yang sama:

- CREATE TABLE contoh_signed_1 (a INT SIGNED, b DECIMAL(5,2) SIGNED)
- CREATE TABLE contoh_signed_1 (a INT, b DECIMAL(5,2))

Atribut SIGNED umumnya baru dipakai pada saat modifikasi tabel untuk menghapus efek atribut UNSIGNED.

9.5 Atribut UNSIGNED

Atribut UNSIGNED digunakan untuk memperbesar jangkauan tipe data positif integer dengan cara mengalihkan jatah dari nilai negatif ke positif.

Kolom integer yang ditambahkan atribut UNSIGNED tidak lagi bisa menyimpan angka negatif, akan tetapi jangkauan angka positifnya menjadi 2 kali lipat lebih besar.

Tabel berikut merangkum jangkauan dari nilai SIGNED dan UNSIGNED untuk tipe data integer:

Tipe Data	Jangkauan SIGNED	Jangkauan UNSIGNED	Ukuran
TINYINT	-128 to 127	0 to 255	1 byte
SMALLINT	-32,768 to 32,767	0 to 65,535	2 bytes
MEDIUMINT	-8,388,608 to 8,388,607	0 to 16,777,215	3 bytes
INT	-2,147,483,648 to 2,147,483,647	0 to 4,294,967,295	4 bytes
BIGINT	-9,223,372,036,854,775,808 to -9,223,372,036,854,775,807	0 to 18,446,744,073,709,551,615	8 bytes

Jika atribut UNSIGNED ditambahkan ke tipe data DECIMAL, FLOAT, dan DOUBLE, efeknya tidak akan memperbesar jangkauan, tetapi hanya menghapus kemampuan tipe data tersebut untuk menampung angka negatif.

Contoh penggunaan atribut UNSIGNED sudah kita bahas pada materi tentang tipe data **number**. Query DESC bisa digunakan untuk melihat apakah sebuah kolom memiliki atribut UNSIGNED atau bukan.

```
CREATE TABLE contoh_unsigned_1 (
    a INT UNSIGNED,
    b DECIMAL(5,2) SIGNED
);

DESC contoh_unsigned_1;
+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| a     | int(10) unsigned | YES  |     | NULL    |       |
| b     | decimal(5,2)      | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
```

Jika sebuah kolom didefinisikan sebagai UNSIGNED, akan ada keterangan tambahan di bagian type. Untuk SIGNED, tidak ada keterangan apapun karena merupakan atribut default MySQL.

9.6 Atribut ZEROFILL

Atribut ZEROFILL digunakan pada tipe data **number** yang berfungsi untuk mengisi angka 0 di sebelah kiri dan kanan data selama angka tersebut belum sesuai dengan *display width* (jumlah

digit tampilan).

Atribut `ZEROFILL` secara tidak langsung juga menambahkan efek `UNSIGNED`.

Dua query berikut akan menghasilkan tabel yang sama:

- `CREATE TABLE contoh_zerofill_1 (a INT(8) ZEROFILL)`
- `CREATE TABLE contoh_zerofill_1 (a INT(8) ZEROFILL UNSIGNED)`

Untuk mengecek apakah sebuah kolom memiliki atribut `ZEROFILL`, kita bisa menggunakan query `DESC/DESCRIBE`:

```
CREATE TABLE contoh_zerofill_1 (a INT(8) ZEROFILL);
```

```
DESC contoh_zerofill_1;
```

Field	Type	Null	Key	Default	Extra
a	int(8) unsigned zerofill	YES		NULL	

Terlihat bahwa kolom a ditulis sebagai “`int(8) unsigned zerofill`”, meskipun saya tidak menulis atribut `UNSIGNED`.

Contoh penggunaan atribut `ZEROFILL` juga sudah kita bahas pada bab sebelumnya bagian tipe data **number**.

9.7 Atribut CHARACTER SET dan COLLATION

Materi tentang character set dan collation pernah saya bahas dalam bab mengenai pembuatan database. Disana dijelaskan bahwa character set dan collation bisa ditambahkan ke dalam query `CREATE DATABASE`.

Hal yang sama juga berlaku untuk tabel dan kolom. Dengan menulis atribut `CHARACTER SET` dan `COLLATION` pada saat pendefinisian kolom, kita bisa membuat tabel dengan character set dan collation yang berlainan untuk setiap kolom.

Character set dan collation di level kolom ini akan menimpa character set dan collation yang ditetapkan di level tabel maupun di level database. Atribut `CHARACTER SET` dan `COLLATION` hanya bisa ditempatkan pada kolom dengan tipe data `CHAR`, `VARCHAR`, `TEXT`, `ENUM` dan `SET`.

Berikut contoh penggunaan atribut `CHARACTER SET` dan `COLLATION` di dalam pendefinisian kolom:

```

CREATE TABLE contoh_charset_1 (
    a VARCHAR(10) CHARACTER SET utf8 COLLATE utf8_persian_ci,
    b TINYTEXT CHARSET ascii,
    c ENUM('siang','malam')
);

SHOW CREATE table contoh_charset_1 \G
*****
1. row ****
Table: contoh_charset_1
Create Table: CREATE TABLE `contoh_charset_1` (
    `a` varchar(10) CHARACTER SET utf8 COLLATE utf8_persian_ci DEFAULT NULL,
    `b` tinytext CHARACTER SET ascii,
    `c` enum('siang','malam') DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1
1 row in set (0.00 sec)

```

Saya membuat tabel contoh_charset_1 dengan 3 kolom:

- Kolom a di set sebagai VARCHAR(10) dengan tambahan atribut CHARACTER SET utf8 COLLATE utf8_persian_ci. Kedua pengaturan ini akan menjadi character set dan collation untuk kolom a.
- Kolom b di set sebagai TINYTEXT CHARSET ascii. Disini saya tidak menulis collation untuk kolom b. Sehingga yang dipakai adalah collation default untuk charset ascii, yakni ascii_general_ci.
- Kolom c di set sebagai ENUM('siang','malam'). Untuk kolom ini saya tidak menulis character set maupun collation. Jika ini yang terjadi, kolom c akan menggunakan character set dan collation *default* dari tabel contoh_charset_1. Jika tabel contoh_charset_1 juga tidak memiliki character set dan collation, akan diambil dari level database.

Query SHOW CREATE TABLE contoh_charset_1 bisa dipakai untuk memeriksa nilai character set dan collation dari setiap kolom.

Diakhir tampilan terdapat DEFAULT CHARSET=latin1, ini adalah character set bawaan tabel yang diambil dari database belajar. Jika sebuah kolom dalam tabel contoh_charset_1 tidak dituliskan character set-nya, nilai inilah yang dipakai.

Untuk melihat perbedaan penggunaan charset dan collation, kita akan buat beberapa contoh tabel.

Pertama, mari lihat perbedaan antara charset ascii, latin1 dan utf8:

```
CREATE TABLE contoh_charset_2 (
    a VARCHAR(10) CHARACTER SET ascii ,
    b VARCHAR(10) CHARACTER SET latin1,
    c VARCHAR(10) CHARACTER SET utf8
);
```

Saya membuat tabel contoh_charset_2 dengan 3 kolom: a, b dan c. Kolom a di-set dengan charset ascii, kolom b charset latin1 dan kolom c charset utf8.

Perbedaan antar charset ini terletak dari dukungan untuk karakter non-latin, sebagai contoh string ‘Café’ tidak akan bisa diinput ke kolom a:

```
INSERT INTO contoh_charset_2 (a) VALUES ('Café');
ERROR 1366 (22007): Incorrect string value: '\x82' for column 'a' at row 1
```

Ini terjadi karena charset ascii hanya terdiri dari 127 karakter saja dan didalamnya tidak terdapat huruf ‘é’. Untuk kolom b dan c, string ‘Café’ bisa diinput:

```
INSERT INTO contoh_charset_2 (b) VALUES ('Café');
INSERT INTO contoh_charset_2 (c) VALUES ('Café');
```

```
SELECT * FROM contoh_charset_2;
```

a	b	c
NULL	Café	NULL
NULL	NULL	Café

Dalam penggunaan sehari-hari, dampak perbedaan antar charset ini tidak begitu terasa. Charset ascii sekalipun sudah mendukung penuh seluruh alphabet latin yang ada di dalam teks berbahasa indonesia (huruf A-Z dan a-z).



Untuk menginput karakter unicode (utf-8), ada beberapa pengaturan yang mesti diubah. Karena cmd windows tidak mendukung tampilan unicode, kita harus menggunakan interface lain seperti phpMyAdmin atau diinput dari kode program PHP. Karena cukup rumit, saya tidak akan membahasnya.

Jika charset berurusan dengan ketersediaan jenis karakter, maka collation berfungsi untuk mengatur proses pengurutan dan pencarian.

Collation sepantas juga tidak terlalu tampak fungsinya. Pengurutan karakter di huruf latin yang kita gunakan sehari-hari sudah jelas. Huruf a akan diurutkan lebih dulu dari pada huruf b, huruf b lebih dulu daripada huruf c, dst.

Collation ini baru terlihat untuk karakter yang punya banyak huruf a, seperti bahasa negara eropa timur. Untuk bahasa ini ada huruf à, á, â, ã, ä dan å. Yang manakah harus di dahulukan? Hasilnya tergantung dari collation yang dipakai.

Secara garis besar, collation bisa dibagi menjadi 3 kelompok:

- Collation case insensitive (tidak membedakan huruf besar/kecil)
- Collation case sensitive (membedakan huruf besar/kecil)
- Collation binary

Jenis collation ini bisa dilihat dari beberapa karakter terakhir. Jika berakhiran _ci berarti **case insensitive**, jika _cs berarti **case sensitive**, jika _bin berarti **binary**.

Berikut contoh praktik melihat perbedaan dari berbagai collation:

```
CREATE TABLE contoh_collation_1 (
    a VARCHAR(10) CHARSET latin1 COLLATE latin1_swedish_ci,
    b VARCHAR(10) CHARSET latin1 COLLATE latin1_german1_ci,
    c VARCHAR(10) CHARSET latin1 COLLATE latin1_general_cs,
    d VARCHAR(10) CHARSET latin1 COLLATE latin1_bin
);

INSERT INTO contoh_collation_1 VALUES
('Andi', 'Andi', 'Andi', 'Andi'),
('budi', 'budi', 'budi', 'budi'),
('Züma', 'Züma', 'Züma', 'Züma'),
('Zuma', 'Zuma', 'Zuma', 'Zuma')
;

SELECT * FROM contoh_collation_1;
+---+---+---+---+
| a | b | c | d |
+---+---+---+---+
| Andi | Andi | Andi | Andi |
| budi | budi | budi | budi |
| Züma | Züma | Züma | Züma |
| Zuma | Zuma | Zuma | Zuma |
+---+---+---+---+
```

Tabel contoh_collation_1 memiliki 4 kolom dengan tipe data dan charset yang sama, yakni VARCHAR(10) CHARSET latin1. Perbedaannya ada di bagian collation. Setelah itu saya menginput 4 nama ke setiap kolom: Andi, budi, Züma dan Zuma.

Berikut hasil proses pengurutan:

```
SELECT a FROM contoh_collation_1 ORDER BY a ASC;
```

a
Andi
budi
Zuma
Züma

```
SELECT b FROM contoh_collation_1 ORDER BY b ASC;
```

b
Andi
budi
Züma
Zuma

```
SELECT c FROM contoh_collation_1 ORDER BY c ASC;
```

c
Andi
budi
Zuma
Züma

```
SELECT d FROM contoh_collation_1 ORDER BY d ASC;
```

d
Andi
Zuma
Züma
budi

Perhatikan urutan di baris terakhir. Untuk kolom a dan c diisi oleh string Züma. Sedangkan pada kolom d diisi oleh string Zuma. Inilah perbedaan penggunaan collation latin1_german1_ci, dimana huruf ü diurutkan lebih dulu daripada huruf u.

Untuk kolom d, yang berada di posisi terakhir adalah string budi. Hal ini terjadi karena pada collation latin1_bin (yang termasuk collation jenis binary), proses pengurutan dilakukan

berdasarkan nomor urut karakter ASCII. Di dalam tabel karakter ASCII, seluruh huruf besar memiliki nomor urut yang lebih kecil, karena itu string budi berada di urutan terakhir.

Pilihan collation juga berdampak ke proses pencarian / perbandingan:

```
SELECT a FROM contoh_collation_1 WHERE a = 'andi';
```

a
Andi

```
SELECT b FROM contoh_collation_1 WHERE b = 'andi';
```

b
Andi

```
SELECT c FROM contoh_collation_1 WHERE c = 'andi';
```

Empty set (0.00 sec)

```
SELECT d FROM contoh_collation_1 WHERE d = 'andi';
```

Empty set (0.00 sec)

Kolom c dan d bersifat case sensitif sesuai dengan akhiran _cs dan _bin dari nama collation. Akibatnya, string 'andi' dianggap berbeda dengan 'Andi'. Oleh karena itu query SELECT tidak menampilkan hasil apa-apa untuk pencarian string 'andi'.

Sebagai tambahan untuk penulisan nama collation, akan terjadi error apabila collation yang dipakai tidak sesuai dengan charset:

```
CREATE TABLE contoh_collation_2 (
    a VARCHAR(10) CHARSET latin1 COLLATE latin2_general_ci,
);
```

```
ERROR 1253 (42000): COLLATION 'latin2_general_ci' is not valid
for CHARACTER SET 'latin1'
```

Saya mendefinisikan kolom a sebagai VARCHAR(10) CHARSET latin1 COLLATE latin2_general_ci. Hasilnya terjadi error. Ini karena collation latin2_general_ci hanya bisa dipakai untuk charset latin2, bukan latin1.

Untuk melihat jenis-jenis collation, bisa menggunakan query berikut:

```
SHOW COLLATION;
```

Lebih jauh tentang teori seputar charset (character set) dan collation bisa dilihat kembali bab 7. **Database, Character set dan Collation.**

9.8 Atribut BINARY

Atribut BINARY adalah atribut khusus untuk kolom dengan tipe data CHAR, VARCHAR dan TEXT. Efeknya akan menambah collation binary ke dalam kolom tersebut.

Ketiga query pendefinisian kolom dibawah ini akan memiliki efek yang sama (dengan asumsi latin1 sebagai charset default):

- a VARCHAR(10) BINARY
- a VARCHAR(10) COLLATE latin1_bin
- a VARCHAR(10) CHARSET latin1 COLLATE latin1_bin

Efek yang didapat sama seperti collation binary, dimana akan mempengaruhi proses pengurutan kolom dan operasi perbandingan:

```
CREATE TABLE contoh_attbinary_1 (
    a VARCHAR(10) BINARY
);

INSERT INTO contoh_attbinary_1 VALUES ('Andi'), ('budi'), ('Ziko');

SELECT * FROM contoh_attbinary_1 ORDER BY a ASC;
+-----+
| a    |
+-----+
| Andi |
| Ziko |
| budi |
+-----+

SELECT * FROM contoh_attbinary_1 WHERE a='andi';
Empty set (0.00 sec)
```

Dengan menjalankan query SHOW CREATE TABLE, bisa dilihat bahwa kolom a akan memiliki collation binary:

```
SHOW CREATE TABLE contoh_attbinary_1 \G
*****
Table: contoh_attbinary_1
Create Table: CREATE TABLE `contoh_attbinary_1` (
  `a` varchar(10) CHARACTER SET latin1 COLLATE latin1_bin DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

Karena tabel contoh_attbinary_1 secara default menggunakan charset latin1, maka collationnya menjadi latin1_bin.

9.9 Atribut ON UPDATE

Atribut ON UPDATE secara khusus digunakan untuk tipe data TIMESTAMP.

Tipe data TIMESTAMP adalah tipe data tanggal dan waktu dalam format ‘CCYYMMDD hh:mm:ss’, seperti 2017-08-08 18:18:49. Secara default, tipe data ini berprilaku khusus. Mari kita lihat contohnya:

```
CREATE TABLE contoh_timestamp_1 (
  a TINYINT,
  b TIMESTAMP
);

INSERT INTO contoh_timestamp_1 (a) VALUES (1);

SELECT * FROM contoh_timestamp_1;
+-----+-----+
| a    | b          |
+-----+-----+
| 1   | 2017-08-08 19:31:31 |
+-----+-----+
```

Saya membuat tabel contoh_timestamp_1 dengan dua kolom: a sebagai TINYINT, dan b sebagai TIMESTAMP.

Setelah tabel berhasil dibuat, saya menginput 1 data ke dalam kolom a. Perhatikan bahwa yang akan diisi hanya kolom a saja, tidak untuk kolom b.

Hasil query SELECT memperlihatkan bahwa kolom b otomatis berisi data. Dimana data tersebut berupa tanggal dan waktu saat ini (tanggal dan waktu saat perintah tersebut dijalankan). Atau sama artinya dengan menginput fungsi NOW() ke kolom b.

Hal yang sama juga berlaku jika yang diinput ke dalam kolom b berupa nilai NULL:

```
INSERT INTO contoh_timestamp_1 VALUES (2,null);
```

```
SELECT * FROM contoh_timestamp_1;
+-----+
| a      | b          |
+-----+
| 1      | 2017-08-08 19:31:31 |
| 2      | 2017-08-08 19:38:49 |
+-----+
```

Inilah fitur default dari kolom dengan tipe data TIMESTAMP. Namun tidak itu saja, kolom b juga akan diupdate jika kolom a nilainya diubah:

```
UPDATE contoh_timestamp_1 SET a = 9 WHERE a = 1;
```

```
SELECT * FROM contoh_timestamp_1;
+-----+
| a      | b          |
+-----+
| 9      | 2017-08-08 19:42:06 |
| 2      | 2017-08-08 19:38:49 |
+-----+
```

Query UPDATE saya pakai untuk mengubah nilai kolom a menjadi 9 khusus untuk baris dimana kolom a bernilai 1. Hasilnya baris kolom a yang sebelumnya berisi angka 1 sudah berubah menjadi angka 9.

Namun perhatikan isi kolom b, pada baris yang kolom a-nya diupdate, nilai pada kolom b juga berubah dari sebelumnya 2017-08-08 19:31:31 menjadi 2017-08-08 19:42:06. Artinya, ketika kolom a diupdate, isi dari kolom b juga menyesuaikan dengan waktu ketika update dijalankan.

Fitur untuk tipe data TIMESTAMP seperti ini sebenarnya berasal dari 2 buah atribut yang aktif secara default:

- DEFAULT CURRENT_TIMESTAMP
- ON UPDATE CURRENT_TIMESTAMP

Hal ini bisa diperiksa dari query DESC:

```
DESC contoh_timestamp_1;
+-----+-----+-----+-----+
| F | Type      | Null | Default           | Extra          |
+-----+-----+-----+-----+
| a | tinyint(4) | YES  | NULL             |                |
| b | timestamp   | NO   | CURRENT_TIMESTAMP | on update CURRENT_TIMESTAMP |
+-----+-----+-----+-----+
```

CURRENT_TIMESTAMP adalah konstanta khusus yang sama maknanya dengan fungsi NOW(). Keduanya berisi waktu saat ini:

```
SELECT CURRENT_TIMESTAMP, NOW();
+-----+-----+
| CURRENT_TIMESTAMP | NOW()          |
+-----+-----+
| 2017-08-08 20:02:07 | 2017-08-08 20:02:07 |
+-----+-----+
```



Dalam MySQL, terdapat berbagai konstanta dan fungsi untuk menghasilkan waktu saat ini: CURRENT_TIMESTAMP(), NOW(), LOCALTIME, LOCALTIME(), LOCALTIMESTAMP, dan LOCALTIMESTAMP(). Serta fungsi CURTIME() untuk menghasilkan tanggal dan waktu dengan ketelitian hingga microdetik.

Karena kita sudah membahas tentang atribut DEFAULT, dapatkan anda memahami pendefinisian kolom berikut?

b **TIMESTAMP DEFAULT CURRENT_TIMESTAMP**

Artinya, ketika kolom b tidak diisi nilai atau diisi nilai NULL, yang akan digunakan adalah waktu saat ini (di hasilkan dari konstanta CURRENT_TIMESTAMP). Ini merupakan fitur default dari kolom bertipe TIMESTAMP.

Bagaimana jika kita tidak ingin kolom TIMESTAMP berprilaku seperti ini? Tinggal diubah pendefinisian kolomnya, misalnya sebagai berikut:

b **TIMESTAMP DEFAULT 0**

Sekarang, kolom b akan bernilai 0 seandainya tidak diinput data apapun.

Selain DEFAULT CURRENT_TIMESTAMP, kolom TIMESTAMP juga memiliki atribut ON UPDATE CURRENT_TIMESTAMP. Atribut inilah yang “bertanggung jawab” dengan proses update ketika kolom lain di baris yang sama diubah nilainya.

Berikut cara penulisan atribut ini:

b `TIMESTAMP ON UPDATE CURRENT_TIMESTAMP`

Kolom b akan diupdate nilainya jika di baris yang sama terdapat nilai kolom yang berubah.

Penggunaan atribut `DEFAULT CURRENT_TIMESTAMP` dan `ON UPDATE CURRENT_TIMESTAMP` memang sedikit membingungkan. Karena jika tidak ditulis, keduanya langsung aktif. Berikut aturan penggunaannya:

- Jika sebuah kolom `TIMESTAMP` tidak ditulis kedua atribut ini, maka `DEFAULT CURRENT_TIMESTAMP` dan `ON UPDATE CURRENT_TIMESTAMP` akan **aktif**.
- Jika sebuah kolom `TIMESTAMP` ditulis atribut `DEFAULT CURRENT_TIMESTAMP` saja, atribut `ON UPDATE CURRENT_TIMESTAMP` **tidak aktif**, termasuk jika menggunakan nilai default lain seperti `DEFAULT 0`.
- Jika sebuah kolom `TIMESTAMP` ditulis atribut `ON UPDATE CURRENT_TIMESTAMP` saja, nilai default yang digunakan adalah 0.

Silahkan anda coba setiap kemungkinan yang ada, kemudian periksa menggunakan query `DESC`.

Prilaku dimana atribut `DEFAULT CURRENT_TIMESTAMP` dan `ON UPDATE CURRENT_TIMESTAMP` akan aktif apabila tidak ditulis, hanya berlaku untuk kolom `TIMESTAMP pertama` dari sebuah tabel. Untuk kolom `TIMESTAMP` kedua, atribut ini harus ditulis agar bisa aktif:

```
CREATE TABLE contoh_timestamp_2 (
    a TINYINT,
    b TIMESTAMP,
    c TIMESTAMP
);
```

```
DESC contoh_timestamp_2;
```

F	Type	Null	Default	Extra
a	tinyint(4)	YES	NULL	
b	timestamp	NO	CURRENT_TIMESTAMP	on update CURRENT_TIMESTAMP
c	timestamp	NO	0000-00-00 00:00:00	

Terlihat bahwa untuk kolom c, atribut `DEFAULT CURRENT_TIMESTAMP` dan `ON UPDATE CURRENT_TIMESTAMP` tidak langsung aktif, meskipun cara pendefinisianya sama seperti kolom b.

Fitur kolom `TIMESTAMP` seperti ini cocok untuk menambah kolom “date created” atau “date modified” ke dalam data tabel. Misalnya untuk memantau kapan sebuah barang masuk ke gudang dan mengetahui kapan stock barang tersebut terakhir diubah.

9.10 Atribut AUTO_INCREMENT

Atribut AUTO_INCREMENT merupakan salah satu atribut yang paling banyak dipakai. Atribut ini berfungsi untuk men-generate (menghasilkan) angka naik secara otomatis, mulai dari 1, 2, 3, dst.

Atribut AUTO_INCREMENT bisa digunakan untuk tipe data integer dan floating-point (FLOAT atau DECIMAL), meskipun penggunaannya untuk tipe data floating-point tidak umum dipakai.

Mari kita coba:

```
CREATE TABLE contoh_auto_increment_1 (
    a TINYINT AUTO_INCREMENT,
    b VARCHAR(10)
);

ERROR 1075 (42000): Incorrect table definition; there can be only one
auto column and it must be defined as a key
```

Error diatas terjadi karena kolom dengan atribut AUTO_INCREMENT harus di definisikan sebagai ‘key’ atau *index*, misalnya sebagai PRIMARY KEY atau UNIQUE. Materi tentang key ini akan kita bahas setelah ini.

Agar tidak terjadi error, tambahkan PRIMARY KEY ke dalam pendefinisian kolom:

```
CREATE TABLE contoh_auto_increment_1 (
    a TINYINT AUTO_INCREMENT PRIMARY KEY,
    b VARCHAR(10)
);
```

Sekarang, tabel contoh_auto_increment_1 sudah berisi kolom a sebagai AUTO_INCREMENT.

Nilai dari kolom a akan otomatis berisi angka jika kolom tersebut tidak diisi nilai, diisi nilai *null* atau diisi angka 0:

```
INSERT INTO contoh_auto_increment_1 (b) VALUES ('merah');
```

```
SELECT * FROM contoh_auto_increment_1;
+---+-----+
| a | b      |
+---+-----+
| 1 | merah |
+---+-----+
```

```
INSERT INTO contoh_auto_increment_1 VALUES (null, 'hitam');
```

```
SELECT * FROM contoh_auto_increment_1;
+---+-----+
| a | b      |
+---+-----+
```

```
+---+-----+
| 1 | merah |
| 2 | hitam |
+---+-----+
INSERT INTO contoh_auto_increment_1 VALUES (0, 'putih');

SELECT * FROM contoh_auto_increment_1;
+---+-----+
| a | b      |
+---+-----+
| 1 | merah |
| 2 | hitam |
| 3 | putih |
+---+-----+
```

Jika kolom AUTO_INCREMENT diisi dengan angka yang sudah ada, akan menyebabkan error:

```
INSERT INTO contoh_auto_increment_1 VALUES (3, 'kuning');
ERROR 1062 (23000): Duplicate entry '3' for key 'PRIMARY'
```

Apabila kolom AUTO_INCREMENT diisi dengan angka yang belum ada, nomor urut akan ‘lompat’ ke nomor tersebut:

```
INSERT INTO contoh_auto_increment_1 VALUES (10, 'kuning');
INSERT INTO contoh_auto_increment_1 VALUES (null, 'biru');

SELECT * FROM contoh_auto_increment_1;
+---+-----+
| a | b      |
+---+-----+
| 1 | merah |
| 2 | hitam |
| 3 | putih |
| 10 | kuning |
| 11 | biru   |
+---+-----+
```

Karena saya menginput angka 10 ke dalam kolom a, maka angka *auto increment* berikutnya akan naik ke 11, yakni sebagai kelanjutan dari 10.

Perintah ALTER TABLE bisa dipakai untuk mengubah angka auto increment saat ini, dengan catatan nilai tersebut harus lebih besar dari nilai kolom yang ada (tidak boleh lebih kecil):

```
ALTER TABLE contoh_auto_increment_1 AUTO_INCREMENT = 30;

INSERT INTO contoh_auto_increment_1 (b) VALUES ('hijau');

SELECT * FROM contoh_auto_increment_1;
+---+---+
| a | b   |
+---+---+
| 1 | merah |
| 2 | hitam |
| 3 | putih |
| 10 | kuning |
| 11 | biru |
| 30 | hijau |
+---+---+
```

Sekarang, nomor urut kolom *a* akan menaik mulai dari 30.

Fungsi khusus `LAST_INSERT_ID()` bisa dipakai untuk menampilkan angka terakhir yang diinput ke kolom *auto increment*:

```
SELECT LAST_INSERT_ID();
+-----+
| LAST_INSERT_ID() |
+-----+
|          30 |
+-----+
```

Fungsi `LAST_INSERT_ID()` mengambil angka *auto increment* terakhir tanpa memperhatikan tabel. Artinya jika kita menginput lebih dari 1 tabel yang memiliki kolom *auto increment*, yang dipakai adalah tabel terakhir.

Selain itu, fungsi `LAST_INSERT_ID()` hanya berisi angka jika perintah `INSERT` dijalankan dalam session tersebut. Jika belum ada perintah `INSERT`, fungsi `LAST_INSERT_ID()` akan mengembalikan angka 0 meskipun tabelnya sudah berisi data.

Untuk tabel dengan storage engine **InnoDB** (yakni storage engine default dari MySQL), angka yang di-generate tidak akan digunakan ulang. Artinya jika kita menghapus baris terakhir dan kemudian menginput data baru, angka yang digunakan bukan angka yang dihapus tadi, tapi naik ke angka berikutnya:

```
DELETE FROM contoh_auto_increment_1 WHERE a = 30;

SELECT * FROM contoh_auto_increment_1;
+---+-----+
| a | b    |
+---+-----+
| 1 | merah |
| 2 | hitam |
| 3 | putih |
| 10 | kuning |
| 11 | biru   |
+---+-----+

INSERT INTO contoh_auto_increment_1 (b) VALUES ('hijau');

SELECT * FROM contoh_auto_increment_1;
+---+-----+
| a | b    |
+---+-----+
| 1 | merah |
| 2 | hitam |
| 3 | putih |
| 10 | kuning |
| 11 | biru   |
| 31 | hijau  |
+---+-----+
```

Saya menghapus baris terakhir dari tabel contoh_auto_increment_1. Baris terakhir ini berisi angka 30 di kolom AUTO_INCREMENT. Ketika diinput data baru, kolom a tidak kembali berisi angka 30, tapi lanjut menjadi angka 31.

Ada pengecualian untuk kasus seperti ini. Yaitu jika sesaat setelah menghapus baris 30 saya mematikan MySQL Server, urutan angka AUTO_INCREMENT akan kembali ke angka tertinggi terakhir, yang dalam contoh tabel diatas akan menjadi angka 12.

Masih untuk tabel dengan storage engine InnoDB, nomor urut kolom AUTO_INCREMENT tidak dapat di reset. Meskipun kita menghapus seluruh baris dengan query DELETE, nomor urut akan lanjut berdasarkan nilai terakhir.

Satu-satunya cara untuk me-reset nomor urut adalah dengan menjalankan query TRUNCATE. Namun harap diperhatikan bahwa query TRUNCATE juga akan menghapus seluruh data tabel:

```
TRUNCATE contoh_auto_increment_1;

INSERT INTO contoh_auto_increment_1 (b) VALUES ('hijau');

SELECT * FROM contoh_auto_increment_1;
+---+-----+
| a | b      |
+---+-----+
| 1 | hijau |
+---+-----+
```

Di dalam sebuah tabel, tidak bisa dibuat lebih dari 1 kolom dengan atribut AUTO_INCREMENT:

```
CREATE TABLE contoh_auto_increment_2 (
    a TINYINT AUTO_INCREMENT PRIMARY KEY,
    b VARCHAR(10),
    c TINYINT AUTO_INCREMENT PRIMARY KEY
);
```

ERROR 1075 (42000): Incorrect **table** definition; there can be **only** one auto **column and** it must be **defined as a key**

Penggunaan atribut AUTO_INCREMENT secara tidak langsung juga mengaktifkan atribut NOT NULL. Kedua penulisan kolom berikut akan menghasilkan kolom yang sama:

- a TINYINT AUTO_INCREMENT PRIMARY KEY
- a TINYINT NOT NULL AUTO_INCREMENT PRIMARY KEY

Hasil ini bisa terlihat dari query DESC:

```
DESC contoh_auto_increment_1;
+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra           |
+-----+-----+-----+-----+-----+
| a     | tinyint(4)   | NO   | PRI | NULL    | auto_increment |
| b     | varchar(10)  | YES  |     | NULL    |                 |
+-----+-----+-----+-----+-----+
```

Karena penomoran AUTO_INCREMENT dimulai dari angka 1, atribut UNSIGNED sering ditambahkan agar jangkauan angka positif menjadi lebih besar.

Atribut AUTO_INCREMENT yang kita pelajari disini banyak dipakai untuk membuat kolom tabel unik dan berfungsi sebagai penentu identitas sebuah baris, atau dikenal juga sebagai PRIMARY KEY. Inilah yang akan kita bahas berikutnya.

9.11 Atribut PRIMARY KEY

PRIMARY KEY adalah atribut untuk membuat suatu kolom menjadi primary key.

Dalam teori database, **primary key** adalah kolom utama yang berfungsi sebagai identitas setiap baris. Dalam satu tabel, hanya bisa terdapat 1 primary key (tidak boleh lebih).

Karena fungsinya sebagai ‘identitas’, data di dalam kolom primary key tidak boleh kosong (NOT NULL), dan tidak boleh berulang (setiap data harus unik).

Umumnya primary key hanya dibuat untuk 1 kolom saja, tapi juga memungkinkan untuk menjadikan gabungan 2 kolom sebagai primary key (disebut juga sebagai *composite primary key*). Mengenai **composite primary key** akan kita bahas dalam materi terpisah.

Tipe data apapun bisa digunakan sebagai primary key, walaupun yang paling umum adalah untuk tipe data *integer*.

Berikut cara penulisan kolom dengan atribut PRIMARY KEY:

```
CREATE TABLE contoh_primary_key_1(
    a INT PRIMARY KEY,
    b VARCHAR(10)
);

DESC contoh_primary_key_1;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| a     | int(11)   | NO   | PRI | NULL    |       |
| b     | varchar(10) | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
```

Dari hasil query DESC, terlihat bahwa kolom **key** pada field **a** berisi ‘PRI’ yang menandakan bahwa field **a** berperan sebagai **primary key**. Selain itu juga terdapat nilai NO untuk kolom NULL yang menandakan file **a** tidak bisa diisi nilai NULL.

Berikut percobaan menginput nilai NULL ke dalam kolom **a**:

```
INSERT INTO contoh_primary_key_1 VALUES (null, 'hijau');
ERROR 1048 (23000): Column 'a' cannot be null
```

```
INSERT INTO contoh_primary_key_1 (b) VALUES ('hijau');
ERROR 1364 (HY000): Field 'a' doesn't have a default value
```

Kolom **a** juga tidak bisa diinput angka yang sama, karena kolom primary key harus bersifat unik (*unique*):

```
INSERT INTO contoh_primary_key_1 VALUES (1, 'hijau');
Query OK, 1 row affected (0.07 sec)

INSERT INTO contoh_primary_key_1 VALUES (1, 'merah');
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
```

Namun tidak masalah untuk kolom selain *primary key*:

```
INSERT INTO contoh_primary_key_1 VALUES (5, 'hijau');
```

```
SELECT * FROM contoh_primary_key_1;
+---+---+
| a | b      |
+---+---+
| 1 | hijau |
| 5 | hijau |
+---+---+
```

Biasanya, kolom dengan atribut PRIMARY KEY ditempatkan sebagai kolom pertama, tapi ini bukan sebuah keharusan:

```
CREATE TABLE contoh_primary_key_2(
    a INT,
    b VARCHAR(10) PRIMARY KEY
);

DESC contoh_primary_key_2;
+-----+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| a     | int(11)   | YES  |      | NULL    |       |
| b     | varchar(10) | NO   | PRI | NULL    |       |
+-----+-----+-----+-----+-----+-----+
```

Disini saya membuat kolom kedua yang bertipe data VARCHAR sebagai **primary key**.

Lebih lanjut tentang *primary key* ini akan kita bahas kembali dalam bagian khusus mengenai **referential integrity**, yakni peranan **primary key** dan **foreign key** dalam proses interaksi antar tabel.

9.12 Atribut UNIQUE

Atribut **UNIQUE** digunakan untuk membatasi kolom agar tidak boleh memiliki data yang sama (data kolom harus unik).

Berikut contoh penggunaannya:

```
CREATE TABLE contoh_unique_key_1(
    a INT UNIQUE,
    b VARCHAR(10)
);

INSERT INTO contoh_unique_key_1 VALUES (1, 'hijau');
Query OK, 1 row affected (0.06 sec)

INSERT INTO contoh_unique_key_1 VALUES (1, 'merah');
ERROR 1062 (23000): Duplicate entry '1' for key 'a'
```

Kolom a saya definisikan dengan tambahan atribut UNIQUE. Hasilnya, ketika angka 1 diinput untuk kedua kali, akan terjadi error.

Namun berbeda dengan PRIMARY KEY, atribut UNIQUE masih membolehkan nilai NULL:

```
DESC contoh_unique_key_1;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| a     | int(11)   | YES  | UNI | NULL    |       |
| b     | varchar(10) | YES  |      | NULL    |       |
+-----+-----+-----+-----+-----+

INSERT INTO contoh_unique_key_1 VALUES (null, 'merah');
INSERT INTO contoh_unique_key_1 VALUES (null, 'biru');

SELECT * FROM contoh_unique_key_1;
+-----+-----+
| a   | b   |
+-----+-----+
| 1  | hijau |
| NULL | merah |
| NULL | biru  |
+-----+-----+
```

Ini memang sedikit aneh karena kita bisa menginput lebih dari 1 nilai NULL ke dalam kolom dengan atribut UNIQUE. Alasannya, di dalam MySQL setiap nilai NULL dianggap berbeda.

Ketika kolom di set sebagai primary key, atribut UNIQUE ini aktif secara default beserta atribut NOT NULL. Kedua query pembuatan kolom di bawah ini dianggap sama:

- a INT PRIMARY KEY
- a INT PRIMARY KEY UNIQUE NOT NULL

Sebagai tambahan, MySQL juga memiliki tipe data khusus: SERIAL. Tipe data ini sama artinya dengan BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE:

```
CREATE TABLE contoh_serial_1 (
    a SERIAL,
    b VARCHAR(10)
);
```

```
DESC contoh_serial_1;
```

Field	Type	Null	Key	Default	Extra
a	bigint(20) unsigned	NO	PRI	NULL	auto_increment
b	varchar(10)	YES		NULL	

Dengan menggunakan BIGINT UNSIGNED, angka maksimum yang bisa ditampung oleh kolom SERIAL menjadi sangat besar, yakni hingga 18,446,744,073,709,551,615.

9.13 Menggabungkan Atribut

Hampir semua atribut tipe data yang kita bahas dalam bab ini bisa digabung. Urutan penulisannya: dahulukan atribut yang spesifik ke sebuah tipe data, kemudian diikuti oleh atribut generik.

Misalkan saya ingin menambahkan atribut NOT NULL dan ZEROFILL ke kolom tipe data INT(5), penulisannya adalah: a INT(5) ZEROFILL NOT NULL. Karena atribut ZEROFILL spesifik ke tipe data number. Sedangkan atribut NOT NULL bersifat generik dan bisa digunakan oleh semua tipe data.

Jika penulisannya dibalik menjadi a INT(5) NOT NULL ZEROFILL akan menghasilkan error:

```
CREATE TABLE contoh_gabungan_atribut_1(
    a INT(5) NOT NULL ZEROFILL
);
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual
that corresponds to your MariaDB server version for the right syntax to use
near 'ZEROFILL)' at line 2

CREATE TABLE contoh_gabungan_atribut_1(
    a INT(5) ZEROFILL NOT NULL
);
Query OK, 0 rows affected (0.28 sec)
```

Sebagai latihan, bisakah anda membetulkan error dari query pembuatan tabel berikut:

```
CREATE TABLE contoh_gabungan_atribut_2(
    a BIGINT NOT NULL PRIMARY KEY AUTO_INCREMENT UNSIGNED,
    b VARCHAR(5) NULL DEFAULT 'hello' CHARACTER SET utf8 COLLATE utf8_persian_ci
);
```

ERROR 1064 (42000): You have an error **in** your **SQL** syntax; **check** the manual that corresponds **to** your MariaDB server **version for** the **right** syntax **to** use near **'UNSIGNED, b VARCHAR(5) NULL DEFAULT 'hello' CHARACTER SET utf8**
COLLATE utf8_pe' **at** line 2

Perhatikan urutan atributnya, UNSIGNED lebih spesifik ke tipe data BIGINT, begitu juga CHARACTER SET dan COLLATE yang lebih spesifik ke tipe data VARCHAR.

Penulisan yang benar adalah sebagai berikut:

```
CREATE TABLE contoh_gabungan_atribut_2(
    a BIGINT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT ,
    b VARCHAR(5) CHARACTER SET utf8 COLLATE utf8_persian_ci NULL DEFAULT 'hello'
);
```

Query OK, 0 rows affected (0.26 sec)

Sebagai latihan terakhir dalam bab ini, bisakah anda jelaskan maksud dari setiap kolom?

```
CREATE TABLE contoh_gabungan_atribut_3(
    a BIGINT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,
    b VARCHAR(5) CHARACTER SET utf8 COLLATE utf8_bin NULL DEFAULT 'hello',
    c TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    d DECIMAL(5,2) ZEROFILL NULL DEFAULT NULL
);
```

Seluruh atribut ini sudah kita bahas sebelumnya. Query DESCRIBE dan SHOW CREATE TABLE bisa memperlihatkan beberapa atribut yang ‘tidak tertulis’ tapi aktif secara bawaan.

Dengan memahami berbagai atribut tipe data, kita bisa merancang tabel yang sesuai dengan kebutuhan dan tuntutan aplikasi nantinya.

Selanjutnya, kita akan membahas tentang **Server SQL Mode**.

10. Server SQL Mode

Sepanjang pembahasan materi di bab 8. **Tipe Data MySQL**, beberapa kali saya menyinggung tentang *SQL mode*. **SQL Mode** (atau dikenal juga sebagai **Server SQL Mode**) merupakan mode pengaturan di dalam MySQL.

Salah satu fungsi SQL Mode adalah untuk mengatur seberapa ketat MySQL bisa menerima data yang salah. Misalkan bagaimana jika angka 1000 diinput ke kolom TINYINT? Atau apa yang terjadi jika tanggal yang diinput adalah 30 Februari 2017? Hal seperti ini bisa diatur dari SQL Mode.

Aturan dan batasan yang bisa dikonfigurasi dari SQL Mode cukup banyak, total terdapat sekitar 22 konfigurasi.

Saya memutuskan untuk tidak membahas semuanya karena sebagian besar pengaturan ini butuh materi lanjutan. Atau pengaturan tersebut sangat jarang dipakai. Kita akan membahas beberapa yang penting saja, terutama yang berkaitan dengan **strict mode**.

Jika anda tertarik untuk mempelajari seluruh pengaturan dalam SQL Mode, bisa membaca referensinya ke sini: [Server SQL Modes¹](#).



Kecuali dinyatakan lain, seluruh contoh tabel di input ke dalam database `belajar`. Saya berasumsi anda sudah menjalankan query `USE DATABASE belajar` agar bisa mengikuti materi dalam bab ini.

10.1 Level Penerapan SQL Mode

Pengaturan dari SQL Mode ini bisa di terapkan di 3 tempat:

- Level session
- Level global
- Level file my.ini

Ketiga level diatas tidak mempengaruhi settingan apa yang bisa atau tidak bisa dipakai. Ini hanya berkaitan dengan seberapa “permanen” settingan tersebut berlaku.

Pengaturan MySQL Mode di **level session** hanya berlaku untuk user yang aktif saat ini saja, yakni selama jendela MySQL client (jendela cmd windows) belum ditutup. Ketika user tersebut keluar, pengaturan kembali ke settingan awal.

Pengaturan MySQL Mode di **level global** berlaku selama MySQL server berjalan. Dalam praktik kita, pengaturan global ini aktif selama tombol “stop” dari **XAMPP Control Panel** belum di-klik

¹<https://dev.mysql.com/doc/refman/5.7/en/sql-mode.html>

atau *mysqld.exe* tidak di shutdown. Ketika server dimatikan, pengaturan kembali ke settingan awal.

Agar pengaturan SQL mode bisa permanen, kita harus mengubahnya langsung dari file my.ini. Selain di set secara manual, MySQL juga memiliki pengaturan SQL Mode bawaan. Ini bisa di set saat proses *build MySQL*. Yakni ketika pembuatan file instalasi MySQL. Sebagai contoh, MySQL yang diinstall secara stand alone bisa jadi memiliki pengaturan yang berbeda dengan MySQL yang digabung ke dalam XAMPP.

Untuk melihat pengaturan apa saja yang berlaku, bisa menggunakan dua perintah berikut:

- `SELECT @@SESSION.sql_mode;`
- `SELECT @@GLOBAL.sql_mode;`

Perintah pertama untuk melihat settingan di level session, sedangkan perintah kedua untuk melihat settingan di level global. Mari kita coba:

```
SELECT @@SESSION.sql_mode;
+-----+
| @@SESSION.sql_mode |
+-----+
| NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+-----+  
  
SELECT @@GLOBAL.sql_mode;
+-----+
| @@GLOBAL.sql_mode |
+-----+
| NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+-----+
```

Perintah diatas saya dapat dari MariaDB bawaan XAMPP. Jika anda menggunakan MySQL /MariaDB yang diinstall terpisah, bisa jadi hasilnya akan berbeda.



Untuk MariaDB 10.2.5 yang diinstall secara stand alone, cukup banyak mode yang aktif: `STRICT_TRANS_TABLES,ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION`

Dari percobaan diatas, pengaturan SQL Mode di level session maupun global sama-sama menghasilkan teks berikut:

`NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION`

Perhatikan tanda koma di tengah, artinya terdapat 2 mode pengaturan yakni:

- NO_AUTO_CREATE_USER
- NO_ENGINESUBSTITUTION

Berikut penjelasannya:

- Pengaturan NO_AUTO_CREATE_USER mencegah query GRANT (query pembuatan hak akses) secara otomatis membuat user baru jika hak akses itu diberikan kepada user yang tidak ada sebelumnya.
- Pengaturan NO_ENGINESUBSTITUTION mencegah pembuatan tabel ketika storage engine yang dipilih tidak tersedia (akan menghasilkan error). Jika pengaturan ini tidak aktif, MySQL otomatis membuat tabel menggunakan storage engine default dan menghasilkan warning.

Jika anda tidak paham maksud kedua pengaturan ini, jangan khawatir. Saya akan membahas tentang query GRANT serta *storage engine* nantinya di bab terpisah. Untuk saat ini cukup diketahui bahwa secara bawaan terdapat 2 pengaturan SQL Mode yang aktif di MariaDB bawaan XAMPP.

Di dalam MySQL/MariaDB, terdapat 2 kategori pesan kesalahan: **warning** dan **error**. Jika sebuah perintah query menghasilkan warning, query tersebut akan tetap diproses oleh MySQL. Sedangkan jika query tersebut menghasilkan error, proses akan langsung berhenti.

Sebuah pesan yang sama bisa dikategorikan sebagai warning maupun error, tergantung settingan dari SQL Mode.

10.2 Strict Mode

Pengaturan yang paling sering dipakai dan berdampak cukup besar dalam SQL Mode adalah STRICT_ALL_TABLES dan STRICT_TRANS_TABLES. Kedua pengaturan ini dikenal sebagai '**strict mode**', yakni mode dimana MySQL akan menerapkan aturan yang ketat saat mengeksekusi sebuah query.

Ketika MySQL berjalan di **strict mode**, akan terjadi **error** jika data yang diinput ke dalam tabel tidak sesuai dengan tipe data, bukan **warning** sebagaimana di mode normal.

Di mode non-strict mode (mode normal), ketika diinput angka 1000 ke kolom TINYINT, MySQL otomatis menggantinya menjadi 127 (nilai maksimal untuk tipe data TINYINT adalah 127) beserta sebuah pesan warning. Di dalam strict mode, inputan tersebut akan menghasilkan error dan query gagal berjalan.

Untuk masuk ke strict mode, kita butuh salah satu dari 2 pengaturan: STRICT_ALL_TABLES atau STRICT_TRANS_TABLES. Keduanya nyaris tidak berbeda.

Perbedaan dari kedua aturan ini hanya untuk kasus yang sangat spesifik, yakni ketika ditemukan error saat mengeksekusi banyak query INSERT/UPDATE untuk tabel *non-transactional*. Errornya pun harus di query ke-2 atau sesudahnya.

Pada kasus seperti, STRICT_ALL_TABLES akan membatalkan sisa query yang belum dijalankan (terdapat resiko tabel diupdate sebagian). Sedangkan pada pengaturan STRICT_TRANS_TABLES, query yang error akan diabaikan dan sisanya dijalankan semua (tabel tetap di update semua, tapi ada kemungkinan terdapat data yang tidak valid).

Penjelasan diatas mungkin sedikit susah dipahami, terlebih kita belum masuk ke materi tabel *transactional*.

Namun untuk saat ini boleh dibilang bahwa jika ditemukan salah satu dari aturan STRICT_ALL_TABLES atau STRICT_TRANS_TABLES, maka MySQL sedang aktif di **strict mode**.

Agar bisa melihat perbedaan antara ‘mode normal’ dan ‘strict mode’, kita akan langsung lihat contohnya:

```
CREATE TABLE contoh_strict_1 (
    a TINYINT,
    b VARCHAR(5),
    c DATE
);

INSERT INTO contoh_strict_1 VALUES (1000,"DuniaIlkom",20170230);
Query OK, 1 row affected, 3 warnings (0.09 sec)

SHOW WARNINGS;
+-----+-----+
| Level | Code | Message |
+-----+-----+
| Warning | 1264 | Out of range value for column 'a' at row 1 |
| Warning | 1265 | Data truncated for column 'b' at row 1 |
| Warning | 1265 | Data truncated for column 'c' at row 1 |
+-----+-----+

SELECT * FROM contoh_strict_1;
+---+---+---+
| a | b | c |
+---+---+---+
| 127 | Dunia | 0000-00-00 |
+---+---+---+
1 row in set (0.00 sec)
```

Tabel contoh_strict_1 memiliki 3 kolom: a sebagai TINYINT, b sebagai VARCHAR(5) dan c sebagai DATE.

Selanjutnya saya input 3 data yang sebenarnya tidak valid ke masing-masing kolom:

- Angka 1000 ke kolom a (melebihi limit tipe data TINYINT)
- String ‘DuniaIlkom’ (melebihi limit maksimal karakter dari VARCHAR(5))
- Tanggal 20170230 (bulan februari hanya sampai tanggal 28 untuk tahun 2017)

Meskipun data-data ini tidak sesuai ketentuan, tabel contoh_strict_1 tetap menerima data ini dengan memotongnya menjadi:

- Angka 1000 menjadi 127, yakni nilai maksimum dari TINYINT.
- String ‘Duniai1kom’ dipotong menjadi 5 karakter saja menjadi ‘Dunia’.
- Tanggal 20170230 menjadi 0000-00-00.

Seperti yang terlihat, ketiga data diatas tetap bisa diinput ke dalam tabel. Hanya saja terdapat pesan kesalahan berupa warning.

Selanjutnya, mari kita coba mengaktifkan ‘strict mode’. Untuk mengubah SQL Mode secara session dan global, format perintahnya adalah berikut:

```
SET sql_mode = 'JENIS_PENGATURAN_DISINI, ...';
SET GLOBAL sql_mode = 'JENIS_PENGATURAN_DISINI, ...';
```

Pengaturan baris pertama untuk men-set di level session, sedangkan yang kedua untuk level global.

Untuk percobaan kali ini, kita tidak harus mengubah keduanya, tapi cukup di session saja. Jalankan perintah berikut untuk mengaktifkan ‘strict mode’ di level session:

```
SET sql_mode = 'STRICT_ALL_TABLES';
```

Untuk memastikan ‘strict mode’ telah aktif, jalankan kembali query SELECT @@SESSION.sql_mode:

```
SELECT @@SESSION.sql_mode;
+-----+
| @@SESSION.sql_mode |
+-----+
| STRICT_ALL_TABLES |
+-----+
1 row in set (0.00 sec)
```

Baik, pengaturan sudah aktif. Mari kita coba insert kembali data yang tidak valid:

```
INSERT INTO contoh_strict_1 VALUES (1000, "Duniai1kom", 20170230);
ERROR 1264 (22003): Out of range value for column 'a' at row 1
```

Sekarang akan tampil pesan error *Out of range value for column ‘a’ at row 1*. Yang artinya nilai untuk kolom a berada di luar jangkauan. Query tersebut gagal di eksekusi dan tidak ada satupun data yang diinput ke dalam tabel.

Dari kasus ini juga bisa terlihat bahwa MySQL hanya menampilkan error yang paling awal saja (hanya kolom a). Saya akan ubah nilai 1000 menjadi 80 dan menjalankan ulang query INSERT:

```
INSERT INTO contoh_strict_1 VALUES (80, "Duniailkom", 20170230);
ERROR 1406 (22001): Data too long for column 'b' at row 1
```

Nilai 80 sudah berada di dalam jangkauan tipe data TINYINT, sehingga tidak ada masalah. Namun sekarang giliran kolom b yang error: *Data too long for column 'b' at row 1*. Artinya data terlalu panjang untuk kolom b.

Kembali, saya harus perbaiki data ini dengan mengubahnya menjadi ‘Dunia’:

```
INSERT INTO contoh_strict_1 VALUES (80, "Dunia", 20170230);
ERROR 1292 (22007): Incorrect date value: '20170230' for column 'c' at row 1
```

Sesuai prediksi, MySQL juga komplain terkait tanggal yang salah: *Incorrect date value: '20170230' for column 'c' at row 1*. Data ini harus diubah agar sesuai dengan tanggal yang benar. Saya akan edit menjadi 20170228:

```
INSERT INTO contoh_strict_1 VALUES (80, "Dunia", 20170228);
Query OK, 1 row affected (0.11 sec)
```

```
SELECT * FROM contoh_strict_1;
+-----+-----+-----+
| a    | b    | c      |
+-----+-----+-----+
| 127 | Dunia | 0000-00-00 |
| 80  | Dunia | 2017-02-28 |
+-----+-----+-----+
```

Sekarang tidak ada masalah dan data sukses masuk ke tabel contoh_strict_1. Dalam kebanyakan kasus, strict mode seperti ini bisa membawa keuntungan. Dengan mewajibkan setiap data harus sesuai tipe kolom, data yang di dalam tabel menjadi lebih terjaga dan rapi.

Mendapatkan pesan error jauh lebih baik daripada mendapati data yang salah. Error tampil pada saat itu juga dan bisa langsung diperbaiki. Tetapi data yang salah bisa baru diketahui di bulan depan atau saat-saat mendesak.

Saya sering melihat ada website yang menampilkan tanggal sebagai 1 Januari 1970. Ini adalah hasil dari angka 0 dari kolom dengan tipe data DATE, nilai 0 ini kemudian dikonversi oleh PHP menjadi 1 Januari 1970. Untuk mencegah hal seperti ini, sebaiknya strict mode selalu diaktifkan.

Dalam contoh sebelumnya kita melihat pesan error untuk tipe data TINYINT, VARCHAR dan DATE. Hal yang sama juga berlaku untuk tipe data lain seperti SET dan ENUM.

Jika anda masih ingat, di dalam pembahasan tipe data ENUM dan SET kita mendapati baris kosong jika data yang diinput tidak sesuai dengan daftar yang ada di ENUM dan SET. Di dalam strict mode, hasilnya akan berbeda:

```
INSERT INTO contoh_enum_1 VALUES ('Hukum');
ERROR 1265 (01000): Data truncated for column 'jur' at row 1
```

Saya mencoba menginput string ‘Hukum’ ke dalam tabel contoh_enum_1. Hasilnya akan menjadi error karena jurusan ‘Hukum’ memang tidak ada di dalam pendefinisian tabel contoh_enum_1.

Yang lebih penting lagi, tidak ada baris kosong yang diinput ke dalam tabel sebagaimana pada mode normal (bukan strict mode).

10.3 Menginput Lebih dari 1 Mode

Strict mode barulah salah satu dari 22 pengaturan yang bisa kita input ke dalam SQL Mode. Jika ingin menambahkan aturan lain, penulisannya disambung ke dalam 1 string panjang.

Sebagai contoh, untuk mengaktifkan STRICT_ALL_TABLES dan NO_ENGINE_SUBSTITUTION, perintahnya sebagai berikut:

```
SET sql_mode = 'STRICT_ALL_TABLES,NO_ENGINE_SUBSTITUTION';
```

Yang juga harus menjadi catatan, tidak boleh terdapat spasi diantara tanda koma pemisah SQL Mode. Jika ditulis seperti ini, akan menghasilkan error:

```
SET sql_mode = 'STRICT_ALL_TABLES, NO_ENGINE_SUBSTITUTION';
```

```
ERROR 1231 (42000): Variable 'sql_mode' can't be set to the value
of ' NO_ENGINE_SUBSTITUTION'
```

Perhatikan terdapat 1 spasi setelah tanda koma: 'STRICT_ALL_TABLES, (spasi) NO_ENGINE_-SUBSTITUTION'.

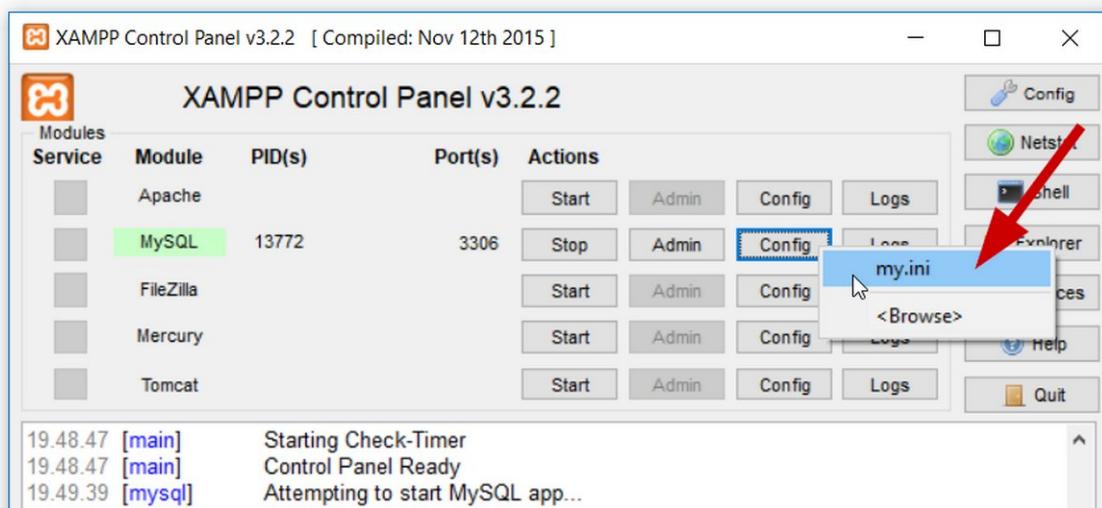
10.4 Mengubah SQL Mode dari my.ini

Strict mode yang kita aktifkan sebelumnya hanya berlaku di level session. Jika kita keluar dari MySQL client dengan mengetik perintah exit, kemudian masuk kembali, pengaturan STRICT_ALL_TABLES tidak lagi aktif.

Agar SQL Mode bisa permanen, pengaturan harus dibuat dari file settingan MySQL, yakni my.ini.

Silahkan buka file my.ini yang berada di C:\xampp\mysql\bin\my.ini. Untuk membukanya bisa menggunakan notepad bawaan windows maupun aplikasi teks editor lain seperti Notepad++.

Atau bisa juga diakses dari XAMPP Control Panel, kemudian di baris MySQL pilih tombol Config -> my.ini



Gambar: Membuka file my.ini dari XAMPP Control Panel

Setelah file `my.ini` terbuka, cari block pengaturan yang diawali dengan `[mysqld]`, lalu tambahkan perintah berikut di baris terakhir:

```
sql_mode="NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION,STRICT_ALL_TABLES"
```

Penulisannya harus sama persis seperti diatas dan dalam 1 baris panjang. Jika ada tambahan karakter lain diantara aturan diatas (seperti spasi), bisa menyebabkan error dan MySQL Server gagal berjalan.

```

26  # The MySQL server
27  [mysqld]
28  port= 3306
29  socket = "C:/xampp/mysql/mysql.sock"
30  basedir = "C:/xampp/mysql"
31  tmpdir = "C:/xampp/tmp"
32  datadir = "C:/xampp/mysql/data"
33  pid_file = "mysql.pid"
34  # enable-named-pipe
35  key_buffer = 16M
36  max_allowed_packet = 1M
37  sort_buffer_size = 512K
38  net_buffer_length = 8K
39  read_buffer_size = 256K
40  read_rnd_buffer_size = 512K
41  myisam_sort_buffer_size = 8M
42  log_error = "mysql_error.log"
43  sql_mode="NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION,STRICT_ALL_TABLES"
44

```

Gambar: Cara penambahan SQL Mode ke dalam my.ini

Terdapat 3 pengaturan yang saya terapkan, yakni:

- NO_AUTO_CREATE_USER
- NO_ENGINE_SUBSTITUTION
- STRICT_ALL_TABLES

Pengaturan ini akan menambahkan strict mode serta membiarkan pengaturan awal dari MariaDB bawaan XAMPP.

Silahkan restart MySQL Server dan cek kembali menggunakan query `SELECT @@SESSION.sql_mode;`:

```
SELECT @@SESSION.sql_mode;
+-----+
| @@SESSION.sql_mode |
+-----+
| STRICT_ALL_TABLES,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+-----+
```

Jika tampak tampilan seperti ini, artinya pengaturan MySQL Mode dari file `my.ini` sudah sukses berjalan.

Pengaturan yang diterapkan dari `my.ini` sebenarnya adalah pengaturan yang paling “lemah”. Efek ini bisa ditimpa pada level global dan level session. Dengan catatan, user tersebut memiliki hak (*privilege*) untuk mengubah SQL Mode.

User root yang kita gunakan selama ini memiliki hak akses untuk mengubah SQL Mode secara global maupun session (memiliki **SUPER privilege**).

Artinya, jika kita menjalankan query `SET GLOBAL sql_mode` atau `SET sql_mode` di dalam cmd windows, aturan dari `my.ini` akan tertimpa.

Dalam bab ini kita telah membahas tentang SQL Mode, terutama efek dari **strict mode**. Untuk pembahasan selanjutnya hingga akhir buku, saya akan menggunakan strict mode yang kita praktekkan disini, sesuai dengan settingan yang telah diubah dari file `my.ini`.

Berikutnya, kita akan membahas dengan lebih detail query `CREATE TABLE`, `ALTER TABLE`, `RENAME TABLE` dan `DROP TABLE` dalam bab tentang **Membuat, Mengubah dan Menghapus Tabel**.

11. Membuat, Mengubah dan Menghapus Tabel

Query pembuatan tabel (`CREATE TABLE`) sudah sering kita gunakan. Namun masih ada beberapa fitur dan perintah tambahan yang belum saya bahas.

Dalam bab kali ini kita akan mempelajari apa saja perintah tambahan yang bisa digunakan untuk pembuatan tabel, mulai dari membuat *temporary table*, mengubah *storage engine*, mengubah awal penomoran `AUTO_INCREMENT` hingga menambah komentar ke dalam tabel.

Setelah itu juga akan dipelajari tentang cara mengubah nama tabel, kolom tabel, dan terakhir cara menghapus tabel.

Query yang akan dipakai adalah `CREATE TABLE`, `ALTER TABLE`, `RENAME TABLE` dan `DROP TABLE`.



Kecuali dinyatakan lain, seluruh contoh tabel di input ke dalam database `belajar`. Saya berasumsi anda sudah menjalankan query `USE DATABASE belajar` agar bisa mengikuti materi dalam bab ini.

11.1 Format Dasar `CREATE TABLE`

Tabel merupakan inti dari database, oleh karena itu MySQL menyediakan beragam perintah dan fitur tambahan untuk pembuatan tabel.

Mengutip dari [Manual MySQL¹](#), berikut format dasar penulisan dari query `CREATE TABLE`:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
  (create_definition,...)
  [table_options]
  [partition_options]
```

Bagian yang berada di dalam tanda kurung siku merupakan perintah opsional dan boleh tidak ditulis.

- `create_definition` adalah tempat untuk mendefinisikan kolom tabel. Disinilah nama kolom, tipe data kolom dan berbagai atribut tipe data ditulis. Ini semua sudah kita bahas dalam bab sebelumnya.
- `table_options` nantinya diisi dengan berbagai settingan tabel, seperti *storage engine*, *charset*, *collation*, serta *comment*. Inilah yang akan kita bahas dalam bab ini.

¹<https://dev.mysql.com/doc/refman/5.7/en/create-table.html>

- **partition_options** adalah tempat mendefinisikan perintah untuk memecah tabel (*partitioning*). Pemecahan tabel biasa dipakai untuk meningkatkan performa serta mempermudah pengelolaan tabel-tabel besar (yang terdiri dari ratusan ribu hingga jutaan baris). **Partitioning** termasuk materi advanced dan tidak akan saya bahas.

Dengan mengabaikan perintah opsional, format dasar query CREATE TABLE menjadi sebagai berikut:

```
CREATE TABLE tbl_name (create_definition, ...)
```

Format seperti ini sudah sering kita pakai, misalnya dalam membuat tabel berikut:

```
CREATE TABLE contoh_tabel_1 (
    a INT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,
    b VARCHAR(10) CHARACTER SET utf8 COLLATE utf8_general_ci
);
```

Kode diatas akan membuat sebuah tabel contoh_tabel_1 dengan dua buah kolom: a dan b.

Kolom a akan menampung nilai angka INTEGER, tidak boleh berisi nilai NULL, merupakan PRIMARY KEY dan angka akan di generate secara otomatis (AUTO_INCREMENT).

Kolom b di definisikan untuk menampung nilai string VARCHAR dengan maksimal 10 karakter. Kolom ini menggunakan charset utf8 dan collation utf8_general_ci.

Aturan Penulisan Nama Tabel dan Kolom Tabel

Di dalam MySQL/MariaDB, nama tabel dan nama kolom termasuk ke dalam kelompok **identifier**. Aturan penulisan dari *identifier* adalah sebagai berikut:

- Bisa terdiri dari angka, huruf besar dan kecil, garis bawah (underscore), tanda dollar, serta karakter *unicode* dengan kode U+0080 hingga U+FFFF.
- Boleh diawali dengan angka, tapi tidak boleh seluruhnya terdiri dari angka.
- Tidak boleh terdapat karakter whitespace (spasi, tab, dst)
- Tidak boleh terdapat karakter khusus seperti titik (.), koma (,) atau @.
- Tidak boleh menggunakan teks yang memiliki makna dalam perintah query. String ‘select’ tidak bisa digunakan sebagai nama tabel maupun nama kolom.
- Panjang maksimal untuk identifier adalah 64 karakter, kecuali untuk query AS (alias) bisa 256 karakter.

Berikut contoh identifier yang **tidak memenuhi syarat**:

- contoh tabel -> mengandung spasi.
- contoh_t@bel -> mengandung karakter @.

- 12345 -> seluruhnya terdiri dari angka.

Berikut identifier yang dibolehkan:

- contoh_t9bl3
- 123a
- \$aku13a

Contoh penggunaan:

```
CREATE TABLE contoh_t9b13 (
    123a INT,
    $aku13a VARCHAR(5)
);
Query OK, 0 rows affected (0.26 sec)
```

```
DESC contoh_t9b13;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| 123a  | int(11)   | YES  |     | NULL    |       |
| $aku13a | varchar(5) | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
```

Penulisan Identifier dengan Backtick

Identifier boleh ditulis tanpa atau dengan tanda kutip *backtick*, yakni karakter yang berada di sebelah kiri angka 1 pada keyboard, seperti contoh berikut:

```
CREATE TABLE `contoh_tabel_2` (
    `kolom_1` INT UNSIGNED,
    `kolom_2` VARCHAR(10)
);
```

Penulisan dengan *backtick* sering dipakai agar lebih jelas membedakan mana bagian *identifier* dan mana bagian dari perintah SQL.

Berbagai aplikasi GUI MySQL seperti **phpMyAdmin** juga otomatis menambahkan karakter *backtick* ketika menggenerate query MySQL.

Query SHOW CREATE TABLE yang biasa dipakai untuk melihat informasi pembuatan tabel juga menggunakan karakter *backtick* diantara nama tabel dan nama kolom tabel:

```
SHOW CREATE TABLE contoh_tabel_2 \G
*****
1. row ****
Table: contoh_tabel_2
Create Table: CREATE TABLE `contoh_tabel_2` (
  `kolom_1` int(10) unsigned DEFAULT NULL,
  `kolom_2` varchar(10) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

Selain memperjelas query pembuatan tabel, penggunaan *backtick* membawa keuntungan lain. Kita bisa membuat nama identifier dengan karakter yang sebelumnya tidak bisa dipakai, seperti spasi, seluruhnya terdiri dari angka, atau nama tabel yang sama dengan perintah SQL.

Berikut percobaannya:

```
CREATE TABLE `contoh dengan spasi`(
  `Nama Lengkap` VARCHAR(50),
  `tangg#1 l@hir` DATE,
  `12345` VARCHAR(150),
  `select` INT
);

DESC `contoh dengan spasi`;
+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| Nama Lengkap | varchar(50) | YES | | NULL | |
| tangg#1 l@hir | date | YES | | NULL | |
| 12345 | varchar(150) | YES | | NULL | |
| select | int(11) | YES | | NULL | |
+-----+-----+-----+-----+-----+
```

Nama tabel dalam contoh diatas adalah `contoh dengan spasi`. Terdapat dua buah tanda spasi di dalam nama tabel ini. Begitu juga dengan nama kolom yang semuanya menyalahi aturan penamaan identifier yang kita bahas sebelumnya.

Penulisan seperti ini tetap valid dan diperbolehkan, selama ditulis dalam tanda *backtick*. Untuk merujuk nama tabel dan kolom yang ditulis seperti ini, kita juga harus menggunakan tanda *backtick*:

```
INSERT INTO `contoh dengan spasi` (`tangg#l l@hir`, `12345`)
VALUES (19960629, 'Jl. Perintis kemerdekaan no.8');

SELECT * FROM `contoh dengan spasi`;
+-----+-----+-----+
| Nama Lengkap | tangg#l l@hir | 12345           | select |
+-----+-----+-----+
| NULL          | 1996-06-29   | Jl. Perintis kemerdekaan no.8 | NULL   |
+-----+-----+-----+
```

Tanpa *backtick*, query diatas akan menghasilkan error.

Maksimum Jumlah Kolom dan Ukuran Kolom

MySQL membatasi jumlah kolom maksimum per tabel sebanyak **4096** buah kolom. Batasan ini bisa lebih kecil tergantung berbagai faktor seperti storage engine yang dipakai. Sebagai contoh, **InnoDB** sebagai *storage engine* default MySQL dan MariaDB, membatasi maksimum 1017 kolom dalam satu tabel.

Jumlah ini sebenarnya sudah lebih dari cukup. Membuat satu tabel dengan terlalu banyak kolom (lebih dari 30 kolom), sudah tidak efisien dan sebaiknya dipecah ke dalam beberapa tabel.

Selain itu, panjang maksimal untuk nama tabel dan judul kolom adalah 64 karakter. Jika lebih akan menghasilkan error:

```
CREATE TABLE
nama_tabel_yang_amat_sangat_panjang_dan_melebihi_batasan_64_karakter (
    a INT
);

ERROR 1103 (42000): Incorrect table name 'nama_tabel_yang_amat_sangat_
panjang_dan_melebihi_batasan_64_karakter'
```

```
CREATE TABLE contoh_tabel_3 (
    nama_kolom_yang_amat_sangat_panjang_dan_melebihi_batasan_64_karakter INT
);

ERROR 1059 (42000): Identifier name 'nama_kolom_yang_amat_sangat_panjang_
dan_melebihi_batasan_64_karakter' is too long
```

Dalam contoh diatas, saya mencoba menulis nama tabel dan nama kolom lebih dari 64 karakter, hasilnya terdapat error: *Identifier name ... is too long*.

Ukuran baris maksimum per tabel adalah **65.535 byte**. Maksudnya, dalam 1 baris total datanya tidak boleh melebihi 65.535 byte. Ini hanya terjadi jika di sebuah tabel terdapat banyak kolom CHAR, VARCHAR, BINARY atau VARBINARY dimana ukuran masing-masing kolom sangat besar.

Tipe data INT butuh ruang penyimpanan sebesar 4 byte. Namun untuk kolom VARCHAR bisa butuh 20.000 byte jika di definisikan sebagai VARCHAR(20000).

```
CREATE TABLE contoh_tabel_4 (
    a VARCHAR(30000),
    b VARCHAR(20000),
    c VARCHAR(20000)
);
```

ERROR 1118 (42000): **Row size too large.** The maximum **row size for the used table type, not** counting BLOBS, **is** 65535. This includes **storage** overhead, **check** the manual. You have **to** change **some** columns **to TEXT or BLOBs**

Dalam tabel contoh_tabel_4 diatas, jumlah total 1 baris kolom adalah $30000 + 20000 + 20000 = 70000$ byte, sudah melebihi batas maksimum 65.535 byte per baris.

Solusinya, ganti kolom yang butuh banyak teks dengan tipe data TEXT atau BLOB. Kedua tipe data ini bisa menampung teks yang sangat besar tapi hanya terhitung 9 sampai 12 byte dalam satu baris. Ini karena secara teknis tipe data TEXT dan BLOB disimpan terpisah dari baris tabel.

Identifier Qualifiers

Identifier qualifiers adalah istilah yang merujuk ke cara pemanggilan sebuah *identifier*. Jika anda masih ingat, nama database, nama tabel dan nama kolom semuanya adalah bagian dari *identifier*.

Terdapat 2 jenis cara pemanggilan identifier: **unqualified** dan **qualified**. Perbedaannya ada dari ruang lingkup pemanggilan.

Hampir seluruh query pembuatan tabel yang kita jalankan sebelum ini mengharuskan kita memilih database terlebih dahulu dengan perintah USE. Misalnya USE belajar untuk memilih database belajar.

Sebenarnya, kita bisa saja langsung membuat tabel tanpa harus memilih database terlebih dahulu. Caranya, gunakan penulisan **qualified identifier** dengan format: `nama_database.nama_tabel`.

Sebagai contoh praktek, silahkan anda tutup MySQL client dengan mengetik perintah exit. Kemudian login kembali sebagai user root. Karena kita belum memilih database apapun, cursornya masih berbentuk: MariaDB [(none)]>.

Mari coba buat sebuah tabel:

```
MariaDB [(none)]> CREATE TABLE contoh_tabel_5 (
    a INT,
    b VARCHAR(10)
);
```

ERROR 1046 (3D000): **No database selected**

Hasilnya terdapat error karena kita belum memilih database apa yang ingin dipakai. Dengan kata lain, MySQL belum mendapat informasi ke database mana tabel contoh_tabel_5 mesti diinput.

Sekarang jalankan query berikut ini:

```
MariaDB [(none)]> CREATE TABLE belajar.contoh_tabel_5 (
    a INT,
    b VARCHAR(10)
);

Query OK, 0 rows affected (0.30 sec)
```

Kali ini saya menggunakan *qualified identifier* dengan menulis nama tabel: `belajar.contoh_tabel_5`. Artinya, tabel `contoh_tabel_5` adalah anggota dari database `belajar`. Tanda titik (.) digunakan sebagai pemisah antara **nama database** dengan **nama tabel**.

Cara penulisan seperti ini juga berlaku untuk query lain, seperti `INSERT` dan `SELECT`:

```
MariaDB [(none)]> INSERT INTO belajar.contoh_tabel_5 VALUES (1, 'putih');
Query OK, 1 row affected (0.07 sec)
```

```
MariaDB [(none)]> SELECT * FROM belajar.contoh_tabel_5;
+-----+-----+
| a     | b      |
+-----+-----+
| 1     | putih |
+-----+-----+
1 row in set (0.00 sec)
```

Perhatikan bahwa cursor masih berupa `MariaDB [(none)]>` yang artinya saya belum memilih database apapun.

Untuk merujuk ke nama kolom, penulisan *qualified identifier*-nya adalah sebagai berikut:

`nama_database.nama_tabel.nama_kolom`

Berikut contohnya:

```
MariaDB [(none)]> SELECT belajar.contoh_tabel_5.b
FROM belajar.contoh_tabel_5;
+-----+
| b     |
+-----+
| putih |
+-----+
1 row in set (0.03 sec)
```

Query `SELECT belajar.contoh_tabel_5.b` bisa dibaca: *Tampilkan isi kolom b dari tabel contoh_tabel_5 yang berada di dalam database belajar.*

Penulisan *qualified identifier* seperti ini baru diperlukan jika terjadi ambigu pada penulisan query. Ini sering terjadi ketika kita menggabungkan beberapa tabel dan terdapat kolom yang bernama sama.

Jika nama kolom / nama tabel / nama database tidak ambigu, dalam arti MySQL “tahu” tabel mana yang dimaksud, kita tidak perlu menulis lengkap *qualified identifier*. Misalnya, query SELECT sebelum ini juga bisa ditulis sebagai berikut:

```
MariaDB [(none)]> SELECT b FROM belajar.contoh_tabel_5;
+---+
| b   |
+---+
| putih |
+---+
1 row in set (0.03 sec)
```

Query diatas tertulis `SELECT b`, kolom `b` yang mana? Karena terdapat query lanjutan: `FROM belajar.contoh_tabel_5`, tidak ada makna ambigu disini. MySQL paham bahwa kolom `b` yang dimaksud berasal dari tabel `contoh_tabel_5` yang ada di database `belajar`.

Penulisan kolom `b` yang tanpa nama tabel dan nama kolom seperti ini disebut sebagai **unqualified identifier**, atau boleh juga disebut sebagai penulisan singkat.

Dengan menggunakan query `USE belajar`, kita menginstruksikan kepada MySQL bahwa jadikan database `belajar` sebagai database *default*. Jika ada penulisan tabel tanpa menyebut nama database-nya (*unqualified identifier*), gunakan database ini.

Setelah menjalankan query `USE belajar`, ketiga query berikut akan bermakna sama:

- `SELECT b FROM contoh_tabel_5.`
- `SELECT b FROM belajar.contoh_tabel_5.`
- `SELECT contoh_tabel_5.b FROM contoh_tabel_5.`
- `SELECT contoh_tabel_5.b FROM belajar.contoh_tabel_5.`
- `SELECT belajar.contoh_tabel_5.b FROM contoh_tabel_5.`

Efek lain dari penulisan *qualified identifier* adalah, kita bisa menampilkan tabel dari database lain tanpa perlu menggunakan query `USE` terlebih dahulu. Seperti contoh berikut:

```
MariaDB [belajar]> SHOW TABLES FROM indonesia;
+-----+
| Tables_in_indonesia |
+-----+
| provinsi             |
+-----+
1 row in set (0.00 sec)

MariaDB [belajar]> SELECT * FROM indonesia.provinsi;
+-----+-----+-----+-----+-----+
| id_prov | nama_prov      | ibu_kota      | populasi | tgl_diresmikan |
+-----+-----+-----+-----+-----+
| 1       | DKI Jakarta     | Jakarta       | 10012271  | 1961-08-28    |
| 2       | Banten           | Serang         | 11704877  | 2000-10-04    |
| 4       | Kepulauan Riau   | Tanjung Pinang | 1917415   | 2002-10-25    |
| 5       | Papua Barat     | Manokwari     | NULL       | NULL          |
+-----+-----+-----+-----+-----+
4 rows in set (0.05 sec)
```

Perhatikan bahwa cursor saat ini, tertulis MariaDB [belajar]>, artinya kita sedang ada di dalam database belajar. Seluruh query yang akan ditulis, dianggap untuk tabel di dalam database ini, kecuali dinyatakan lain.

Dengan menulis SELECT * FROM indonesia.provinsi, saya ingin menampilkan isi tabel provinsi yang ada di database indonesia. Tanpa menuliskan nama database-nya, MySQL akan mencari tabel provinsi yang ada di database belajar.

Sebagai informasi tambahan, jika kita ingin menulis karakter *backtick* ke dalam *qualified identifier*, ditulis untuk setiap *identifier* seperti contoh berikut:

```
SELECT `belajar`.`contoh_tabel_5`.`b` FROM `belajar`.`contoh_tabel_5`;
```

Penggunaan qualified identifier seperti diatas akan banyak kita pakai saat memproses query yang kompleks dan melibatkan beberapa tabel (SELECT ... JOIN).

11.2 CREATE TABLE ... IF NOT EXIST

Perintah tambahan IF NOT EXIST bisa dipakai untuk menghindari pesan error ketika tabel yang dibuat sudah ada sebelumnya. Hal ini menjadi penting ketika kita merancang banyak perintah query yang jalan sekaligus, misalnya dari kode program PHP.

Dalam buku **PHP Uncover** saya membuat sebuah file PHP yang isinya otomatis membuat dan mengisi 2 tabel MySQL dalam sekali akses. Jika MySQL menghasilkan error, seluruh proses akan berhenti dan query selanjutnya tidak akan diproses.

Berikut hasil yang didapat ketika saya menjalankan query CREATE TABLE untuk tabel yang sudah ada:

```
CREATE TABLE contoh_exist_1 (
    a INT,
    b VARCHAR(10)
);
```

ERROR 1050 (42S01): **Table 'contoh_exist_1' already exists**

Berikut hasil yang didapat dengan tambahan perintah IF NOT EXIST:

```
CREATE TABLE IF NOT EXISTS contoh_exist_1 (
    a INT,
    b VARCHAR(10)
);
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

```
SHOW WARNINGS;
+-----+-----+
| Level | Code | Message           |
+-----+-----+
| Note  | 1050 | Table 'contoh_exist_1' already exists |
+-----+-----+
1 row in set (0.00 sec)
```

Dengan tambahan perintah IF NOT EXISTS, pesan error akan berubah menjadi warning. Namun yang lebih penting, jika query tersebut dijalankan dari PHP, query berikutnya tetap akan diproses meskipun tampil pesan warning.

11.3 CREATE TEMPORARY TABLE

Sesuai dengan namanya, **temporary table** digunakan untuk membuat tabel sementara (*temporary*). Temporary table hanya aktif di dalam session saat ini. Ketika session berakhir, misalnya saat jendela cmd windows ditutup atau menjalankan perintah exit, temporary table otomatis akan terhapus.

Temporary table aktif di level session, sehingga dua user yang berbeda bisa membuat temporary tabel dengan nama yang sama. Setiap user hanya bisa mengakses tabel ‘kepunyaan’ masing-masing (tidak bisa mengakses temporary tabel user lain).

Temporary table juga tidak akan tampil ketika query SHOW TABLES dijalankan.

Berikut contoh pembuatan temporary table:

```
CREATE TEMPORARY TABLE contoh_temp_table_1 (
    a INT,
    b VARCHAR(10)
);
```

Hampir seluruh perintah query bisa dijalankan untuk temporary table, termasuk INSERT, SELECT, DESC, dll:

```
INSERT INTO contoh_temp_table_1 VALUES (1, "biru");
```

```
SELECT * FROM contoh_temp_table_1;
+----+----+
| a | b |
+----+----+
| 1 | biru |
+----+----+
```

Meskipun temporary table otomatis dihapus saat session selesai, kita juga bisa menghapusnya secara manual dengan query DROP TABLE seperti halnya tabel biasa:

```
DROP TABLE contoh_temp_table_1;
```

Temporary table biasa dipakai sebagai tabel percobaan (*dummy table*). Atau bisa juga sebagai tabel perantara ketika memproses query yang cukup kompleks, yakni query yang tidak bisa dijalankan dengan 1 perintah saja (butuh beberapa tahap).

Sebagai informasi tambahan, jika temporary tabel dibuat dari dalam kode PHP, tabel tersebut hanya ada selama PHP memproses halaman. Setelah itu, temporary table otomatis akan terhapus.

11.4 CREATE TABLE ... LIKE

Query CREATE TABLE...LIKE bisa dipakai untuk mengcopy struktur tabel lain. Format penulisannya adalah sebagai berikut:

```
CREATE TABLE tabel_baru LIKE tabel_asal
```

Sebagai contoh praktik, saya akan membuat tabel contoh_tabel_asal:

```

CREATE TABLE contoh_tabel_asal (
    a INT AUTO_INCREMENT PRIMARY KEY,
    b VARCHAR(10),
    c DATE
);

DESC contoh_tabel_asal;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra       |
+-----+-----+-----+-----+-----+
| a     | int(11)   | NO   | PRI | NULL    | auto_increment |
| b     | varchar(10) | YES  |     | NULL    |               |
| c     | date       | YES  |     | NULL    |               |
+-----+-----+-----+-----+-----+

INSERT INTO contoh_tabel_asal VALUES
    (null, 'merah', 20170101),
    (null, 'biru', 20170201),
    (null, 'kuning', 20170301)
;

SELECT * FROM contoh_tabel_asal;
+---+---+---+
| a | b   | c           |
+---+---+---+
| 1 | merah | 2017-01-01 |
| 2 | biru   | 2017-02-01 |
| 3 | kuning | 2017-03-01 |
+---+---+---+

```

Saya membuat tabel contoh_tabel_asal dengan 3 kolom: a, b dan c. Kemudian menginput 3 data dan menampilkan hasilnya.

Mari kita copy tabel ini:

```

CREATE TABLE contoh_copy_1 LIKE contoh_tabel_asal;

DESC contoh_copy_1;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra       |
+-----+-----+-----+-----+-----+
| a     | int(11)   | NO   | PRI | NULL    | auto_increment |
| b     | varchar(10) | YES  |     | NULL    |               |
| c     | date       | YES  |     | NULL    |               |
+-----+-----+-----+-----+-----+

```

```
SELECT * FROM contoh_copy_1;
Empty set (0.00 sec)
```

Dari hasil query DESC contoh_copy_1, terlihat bahwa struktur contoh_tabel_asal sudah dicopy ke tabel contoh_copy_1. Terlihat juga bahwa atribut seperti PRIMARY KEY dan AUTO_INCREMENT juga ikut di copy.

Hanya saja, query CREATE TABLE ... LIKE tidak akan men-copy isi tabel. Jika kita menginginkan hal ini, bisa menggunakan query CREATE TABLE ... SELECT.

11.5 CREATE TABLE ... SELECT

Query CREATE TABLE ... LIKE yang kita pelajari sebelumnya bisa dipakai untuk menduplikasi struktur tabel. Namun bagaimana jika isi data dari tabel lama juga ingin dicopy? Inilah fungsi dari query CREATE TABLE ... SELECT.

Query CREATE TABLE ... SELECT bisa digunakan untuk mengcopy tabel berserta isi tabel. Query ini pada dasarnya adalah gabungan dari 2 query: CREATE dan SELECT. Berikut format dasar penulisan query ini:

```
CREATE TABLE tabel_baru SELECT nama_kolom FROM tabel_asal
```

Mari kita coba dengan mengcopy tabel contoh_tabel_asal:

```
CREATE TABLE contoh_copy_2 SELECT * FROM contoh_tabel_asal;
```

```
DESC contoh_copy_2;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| a     | int(11)   | NO   |     | 0        |       |
| b     | varchar(10) | YES  |     | NULL    |       |
| c     | date      | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
```

```
SELECT * FROM contoh_copy_2;
+---+---+---+
| a | b | c |
+---+---+---+
| 1 | merah | 2017-01-01 |
| 2 | biru  | 2017-02-01 |
| 3 | kuning | 2017-03-01 |
+---+---+---+
```

Tabel contoh_copy_2 akan memiliki kolom dan data yang sama seperti tabel contoh_tabel_asal.

Namun terdapat pengecualian, dapatkah anda melihat perbedaannya dari hasil query DESC? Betul, struktur tabel tidak betul-betul sama. Kolom a kehilangan atribut AUTO_INCREMENT PRIMARY KEY. Akibatnya, penomoran kolom a di tabel contoh_copy_2 tidak lagi otomatis:

```
INSERT INTO contoh_copy_2 VALUES (0, 'ungu', 20170401);
```

```
SELECT * FROM contoh_copy_2;
+---+-----+-----+
| a | b      | c      |
+---+-----+-----+
| 1 | merah  | 2017-01-01 |
| 2 | biru    | 2017-02-01 |
| 3 | kuning  | 2017-03-01 |
| 0 | ungu    | 2017-04-01 |
+---+-----+-----+
```

Jika kolom a tetap sebagai auto increment, ketika diinput angka 0 akan otomatis menjadi angka 4, yakni urutan selanjutnya dari penomoran auto increment.

Hanya saja, kolom a tetap berstatus NOT NULL:

```
INSERT INTO contoh_tabel_baru VALUES (null, 'hijau', 20170501);
ERROR 1048 (23000): Column 'a' cannot be null
```

Dari percobaan ini, kita bisa mengambil kesimpulan bahwa query CREATE TABLE ... SELECT hanya mengcopy struktur, isi data, dan beberapa atribut (tidak semua).

Akan tetapi, query CREATE TABLE ... SELECT menawarkan fleksibilitas dalam menentukan kolom apa saja yang ingin di copy. Misalnya saya bisa mengcopy kolom a dan c saja dari tabel contoh_tabel_asal (tidak untuk kolom b):

```
CREATE TABLE contoh_copy_3 SELECT a, c FROM contoh_tabel_asal;
```

```
SELECT * FROM contoh_copy_3;
+---+-----+
| a | c      |
+---+-----+
| 1 | 2017-01-01 |
| 2 | 2017-02-01 |
| 3 | 2017-03-01 |
+---+-----+
3 rows in set (0.00 sec)
```

Disini tabel contoh_copy_3 hanya akan berisi kolom a dan c dari contoh_tabel_asal.

Dengan menyusun query SELECT yang cukup kompleks, kita bisa mengcopy potongan kolom dari berbagai tabel (tidak hanya berasal satu tabel).

11.6 Table Cloning

Dua perintah query yang baru saja kita pelajari sebelum ini, bisa dipakai untuk mengcopy tabel. Namun masing-masingnya memiliki kelemahan:

- Query `CREATE TABLE ... LIKE` bisa mengcopy seluruh struktur tabel, termasuk atribut, key dan index. Tapi tidak mengcopy data yang ada.
- Query `CREATE TABLE ... SELECT` bisa mengcopy struktur tabel beserta data yang ada, tapi tidak mengcopy beberapa atribut, key dan index.

Jadi, bagaimana cara meng-cloning tabel yang sama persis dengan tabel asalnya? yakni mengcopy seluruh struktur, atribut, key, index serta langsung berisi data?

Solusinya, kita bisa menggunakan 2 buah query terpisah. Pertama, jalankan query `CREATE TABLE ... LIKE` untuk mengcopy struktur tabel, kemudian gunakan query `INSERT ... SELECT` untuk mengcopy data tabel.

Query `INSERT ... SELECT` baru akan kita pelajari dalam bab selanjutnya, namun karena berkaitan dengan cara men-copy tabel, akan saya cantumkan disini.

Berikut cara membuat cloningan contoh_tabel_asal:

```
CREATE TABLE contoh_copy_4 LIKE contoh_tabel_asal;
```

```
DESC contoh_copy_4;
```

Field	Type	Null	Key	Default	Extra
a	int(11)	NO	PRI	NULL	auto_increment
b	varchar(10)	YES		NULL	
c	date	YES		NULL	

```
INSERT INTO contoh_copy_4 SELECT * FROM contoh_tabel_asal;
```

```
SELECT * FROM contoh_copy_4;
```

a	b	c
1	merah	2017-01-01
2	biru	2017-02-01
3	kuning	2017-03-01

Dengan menggunakan 2 kali perintah query, tabel `contoh_copy_4` akan menjadi cloningan `contoh_tabel_asal`, termasuk seluruh struktur, tipe data, atribut, serta isi tabel.

Sebagai pembuktian efek auto increment juga ikut dicopy, mari tambah 1 data baru ke tabel `contoh_copy_4`:

```
INSERT INTO contoh_copy_4 VALUES (0, 'ungu', 20170401);
```

```
SELECT * FROM contoh_copy_4;
+---+-----+-----+
| a | b      | c          |
+---+-----+-----+
| 1 | merah | 2017-01-01 |
| 2 | biru   | 2017-02-01 |
| 3 | kuning | 2017-03-01 |
| 4 | ungu   | 2017-04-01 |
+---+-----+-----+
```

Terlihat bahwa kolom a berisi angka 4, padahal saya menginput nilai 0. Ini hanya bisa terjadi jika kolom a memiliki atribut auto increment, dimana angka 4 adalah kelanjutan dari nomor sebelumnya.

Salah satu trik yang sering digunakan adalah membuat temporary table yang berisi cloningan tabel lain. Di dalam temporary table ini kita bisa mencoba berbagai query secara aman, tidak khawatir ada yang salah. Jika query sudah sesuai, baru diterapkan ke tabel asli.

11.7 Table Character Set dan Collation

Dalam bab sebelumnya kita telah membahas cara membuat **character set** dan **collation** di level kolom. Hal yang sama juga bisa dilakukan di level tabel.

Berikut format dasar penulisan character set dan collation di level tabel:

```
CREATE TABLE nama_tabel
(create_definition, ...)
CHARACTER SET nama_charset COLLATE nama_collation;
```

Perintah CHARACTER SET dan COLLATE bersifat opsional (tidak harus ditulis). Jika tidak ada, akan dipakai charset dan collation bawaan database, yang biasanya berupa CHARACTER SET latin1 dan COLLATE latin1_swedish_ci.

Dengan mendefinisikan charset dan collation di level tabel, nilainya akan menimpa charset dan collation dari level database.

Berikut contoh prakteknya:

```
CREATE TABLE contoh_charset_3 (
    a INT,
    b VARCHAR(10)
) CHARACTER SET utf8 COLLATE utf8_german2_ci;

SHOW CREATE TABLE contoh_charset_3 \G
*****
Table: contoh_charset_3
Create Table: CREATE TABLE `contoh_charset_3` (
    `a` int(11) DEFAULT NULL,
    `b` varchar(10) COLLATE utf8_german2_ci DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_german2_ci
```

Tabel contoh_charset_3 saya definisikan dengan tambahan CHARACTER SET utf8 COLLATE utf8_german2_ci. Inilah nilai charset dan collation yang akan digunakan di setiap kolom yang ada di dalam tabel.

Nilai ini bisa ditimpak pada level kolom jika terdapat atribut CHARACTER SET dan COLLATE.

11.8 Storage Engine

Storage engine adalah istilah yang merujuk ke tipe struktur tabel. Di dalam MySQL dan MariaDB, terdapat beberapa storage engine, diantaranya: MyISAM, InnoDB, MEMORY, dan CSV. Storage engine dikenal juga dengan istilah “*table handlers*”.

Setiap storage engine memiliki prilaku dan fitur tersendiri. Ada yang mendukung *transaction*, ada yang memiliki performa tinggi untuk proses pembacaan, ada yang hanya bisa disimpan di memory, hingga ada yang harus disimpan ke hardware tertentu.

Dalam versi terbaru MySQL dan MariaDB, storage engine menggunakan “*plug in architecture*”. Artinya, programmer pihak ketiga (bukan dari perusahaan Oracle / MariaDB) bisa membuat storage engine sendiri untuk ditambahkan ke dalam MySQL.

Storage engine yang bisa kita gunakan tergantung dari versi MySQL / MariaDB. Untuk memeriksa apa saja storage engine yang tersedia, jalankan query SHOW ENGINES. Berikut hasil yang saya dapat dari MariaDB bawaan XAMPP:

```
SHOW ENGINES\G
*****
Engine: CSV
Support: YES
Comment: CSV storage engine
Transactions: NO
XA: NO
Savepoints: NO
*****
```

```
Engine: InnoDB
Support: DEFAULT
Comment: Percona-XtraDB, Supports transactions, row-level locking,
          foreign keys and encryption for tables
Transactions: YES
XA: YES
Savepoints: YES
***** 3. row *****
Engine: MEMORY
Support: YES
Comment: Hash based, stored in memory, useful for temporary tables
Transactions: NO
XA: NO
Savepoints: NO
***** 4. row *****
Engine: MyISAM
Support: YES
Comment: MyISAM storage engine
Transactions: NO
XA: NO
Savepoints: NO
***** 5. row *****
Engine: MRG_MyISAM
Support: YES
Comment: Collection of identical MyISAM tables
Transactions: NO
XA: NO
Savepoints: NO
***** 6. row *****
Engine: Aria
Support: YES
Comment: Crash-safe tables with MyISAM heritage
Transactions: NO
XA: NO
Savepoints: NO
***** 7. row *****
Engine: PERFORMANCE_SCHEMA
Support: YES
Comment: Performance Schema
Transactions: NO
XA: NO
Savepoints: NO
***** 8. row *****
Engine: SEQUENCE
Support: YES
Comment: Generated tables filled with sequential values
```

Transactions: YES

XA: NO

Savepoints: YES

Saya menjalankan query `SHOW ENGINES\G`, dengan tambahan karakter \G agar hasil tampilan memanjang ke bawah. Jika tidak, tabel akan tampak berantakan karena terdapat baris yang terlalu panjang.

Terdapat 8 baris data, berikut penjelasan dari setiap kolom:

- **Engine:** Nama dari storage engine
- **Support:** Jika berisi YES, artinya storage engine tersedia dan bisa digunakan. Jika NO berarti storage engine tidak aktif. Jika tertulis DEFAULT artinya storage engine tersedia dan menjadi pilihan default dari setiap tabel.
- **Comment:** Berisi penjelasan singkat dari setiap storage engine
- **Transactions:** Jika berisi YES, artinya storage engine mendukung *transaction*, jika NO berarti tidak mendukung *transaction*. Materi *transaction* akan kita bahas dalam bab tersendiri.
- **XA:** Apakah storage engine mendukung *distributed transactions* (tidak akan kita bahas dalam buku ini)
- **Savepoints:** Apakah storage engine mendukung pembatalan transaction sebagian (*partial transaction rollback*).

Dari query diatas, terdapat 8 jenis database engine yang ada di MariaDB bawaan XAMPP, yakni: CSV, InnoDB, MEMORY, MyISAM, MRG_MyISAM, ARIA, PERFORMANCE_SCHEMA, dan SEQUENCE.

Tabel berikut berisi penjelasan singkat masing-masing storage engine:

Storage Engine	Keterangan
InnoDB	Merupakan storage engine default di dalam MySQL / MariaDB, mendukung transaction, foreign key dan penguncian level baris (row locking).
MyISAM	Jenis storage engine lama yang ditujukan untuk tabel dengan banyak proses pembacaan. Tidak mendukung transaction. MyISAM dulunya menjadi tabel default MySQL.
MEMORY	Tabel dan data hanya bisa disimpan di memory RAM (akan terhapus saat server di matikan).
CSV	Tabel dengan format penyimpanan CSV (comma-separated values format).
Aria	Perbaikan dari MyISAM, mendukung transaction. Dulunya bernama Maria, tapi diubah menjadi Aria agar tidak rancu dengan MariaDB.
MRG_MyISAM	Storage engine yang terdiri dari kumpulan beberapa tabel MySQL yang digabung.
SEQUENCE	Storage engine khusus untuk menghasilkan nomor urut (sequence).
PERFORMANCE_SCHEMA	Storage engine khusus untuk internal MySQL, bukan untuk penyimpanan data.

Selain daftar diatas, masih banyak storage engine lain yang didukung oleh MySQL dan MariaDB, seperti **ARCHIVE**, **FEDERATED**, **NDBCLUSTER**, **XtraDB** dan **TokuDB**. Akan tetapi storage engine tidak aktif secara default.

Disini juga bisa terlihat perbedaan antara MariaDB dan MySQL. Ada storage engine yang khusus untuk MariaDB, dan ada juga yang khusus untuk MySQL. Referensi lengkapnya bisa dibaca dari Manual resmi masing-masing database: [MariaDB Storage Engines²](https://mariadb.com/kb/en/mariadb/storage-engines/) dan [MySQL Storage Engines³](https://dev.mysql.com/doc/refman/5.7/en/storage-engines.html).

Berikut format dasar penulisan query dengan menambahkan pilihan storage engine:

```
CREATE TABLE nama_tabel
(create_definition,...)
ENGINE = nama_storage_engine
```

Perintah **ENGINE** bersifat opsional (tidak harus ditulis). Jika tidak ada, akan digunakan storage engine default, yakni **InnoDB**. Mari kita bahas beberapa storage engine diatas.

InnoDB Storage Engine

InnoDB adalah storage engine default dari MySQL dan MariaDB. Jika perintah **ENGINE** tidak ditulis, inilah storage engine yang dipakai. Artinya, seluruh tabel yang kita buat dari awal pembahasan buku ini, semuanya menggunakan InnoDB.

Fitur utama dari InnoDB yakni mendukung *transaction* (bisa membatalkan query), mendukung *foreign key* (untuk membuat *referential integrity*) dan penguncian di level baris (*row locking*).

InnoDB dirancang sebagai “all purpose” storage engine, yakni sesuai untuk keperluan apa saja. Berikut contoh pendefinisian tabel InnoDB:

```
CREATE TABLE contoh_innodb_1 (
    a INT,
    b VARCHAR(10)
) ENGINE = InnoDB;

SHOW CREATE TABLE contoh_innodb_1 \G
*****
Table: contoh_innodb_1
Create Table: CREATE TABLE `contoh_innodb_1` (
    `a` int(11) DEFAULT NULL,
    `b` varchar(10) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

Karena InnoDB adalah storage engine default, sebenarnya kita tidak perlu menulis perintah **ENGINE** diatas.

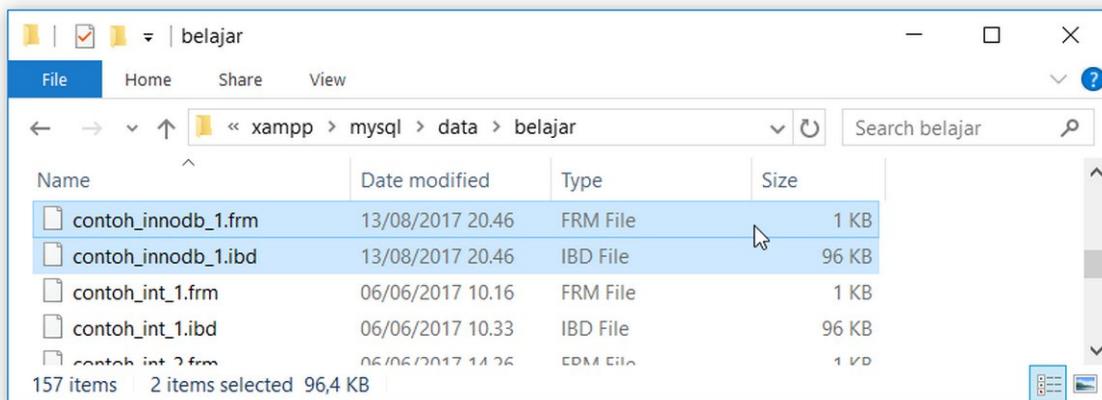
InnoDB storage engine disimpan ke dalam 2 file:

²<https://mariadb.com/kb/en/mariadb/storage-engines/>

³<https://dev.mysql.com/doc/refman/5.7/en/storage-engines.html>

- **contoh_innodb_1.frm**: menyimpan pendefenisian struktur tabel.
- **contoh_innodb_1.ibd**: menyimpan data / isi tabel.

File diatas bisa anda lihat ke folder C:\xampp\mysql\data\belajar.



Gambar: File untuk InnoDB storage engine

Sedikit catatan, file penyimpanan InnoDB bisa jadi tidak ditemukan. Karena secara default, InnoDB menggunakan mekanisme penyimpanan *shared system tablespace*, yakni beberapa tabel disimpan ke dalam 1 file saja (sharing).

Hal ini yang membuat kita tidak bisa memindahkan / mengcopy tabel InnoDB secara langsung dengan cara mengcopy file .frm dan .ibd. Teknik seperti ini biasa dipakai untuk tabel yang dibuat menggunakan MyISAM.

Dalam MariaDB bawaan XAMPP yang saya gunakan, file innoDB diset terpisah untuk setiap tabel, jadi tidak menggunakan shared system tablespace.



Terdapat XtraDB storage engine yang merupakan sebagai versi tambahan (*enhanced*) dari InnoDB. XtraDB dikembangkan oleh [Percona⁴](#), sebuah perusahaan penyedia support dan layanan untuk database MySQL dan MariaDB.

MyISAM Storage Engine

MyISAM dulunya pernah menjadi storage engine default MySQL. MyISAM cocok untuk tabel yang lebih banyak melakukan proses pembacaan (query SELECT) daripada penulisan (query INSERT).

Dulunya MyISAM dipilih karena menawarkan performa yang lebih baik daripada InnoDB. Namun dalam perkembangannya, InnoDB memberikan performa yang hampir sama. Selain itu MyISAM juga belum mendukung *transaction* dan *foreign key*.

Secara umum tidak banyak alasan menggunakan MyISAM daripada InnoDB, kecuali anda memiliki database lama yang butuh fitur-fitur dari MyISAM.

Berikut contoh pendefinisian tabel MyISAM :

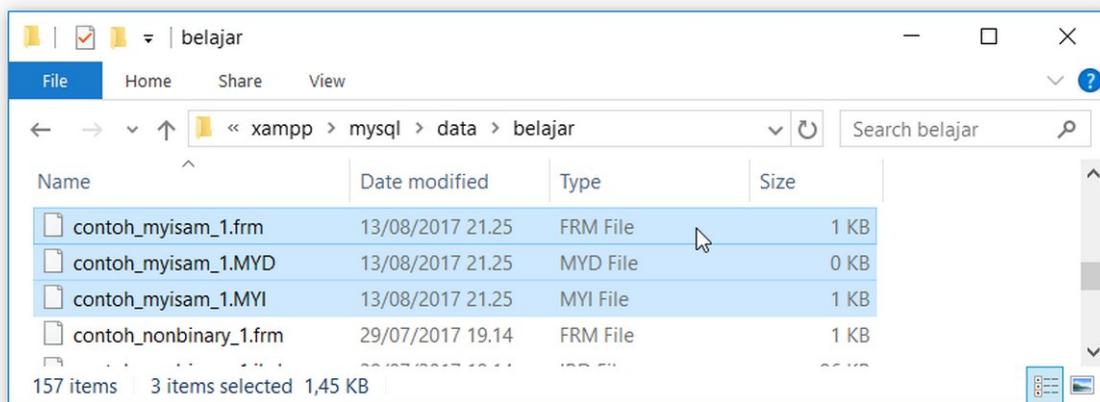
⁴<https://www.percona.com>

```
CREATE TABLE contoh_myisam_1 (
    a INT,
    b VARCHAR(10)
) ENGINE = MyISAM;

SHOW CREATE TABLE contoh_myisam_1 \G
*****
Table: contoh_myisam_1
Create Table: CREATE TABLE `contoh_myisam_1` (
    `a` int(11) DEFAULT NULL,
    `b` varchar(10) DEFAULT NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1
```

MyISAM storage engine disimpan ke dalam 3 file:

- **contoh_myisam_1.frm**: menyimpan pendefenisian struktur tabel.
- **contoh_myisam_1.MYD**: menyimpan data / isi tabel.
- **contoh_myisam_1.MYI**: menyimpan index tabel.



Gambar: File untuk MyISAM storage engine

Dengan mengcopy ketika file ini, kita bisa memindahkan seluruh isi tabel MyISAM ke server MySQL lain atau digunakan sebagai backup.

MEMORY Storage Engine

MEMORY adalah storage engine khusus yang media penyimpanannya berada di memory RAM, bukan di harddisk. Akibatnya, isi tabel akan terhapus ketika server di restart atau dimatikan.

Tabel MEMORY sendiri akan dibuat ulang ketika server restart. Karena pendefinisian tabel tetap disimpan di dalam harddisk (dalam file .frm), akan tetapi tabel ini tidak berisi data apapun (kosong).

MEMORY storage engine mirip seperti temporary table, namun MEMORY tabel bisa diakses oleh user lain, tidak seperti temporary table yang hanya bisa diakses oleh user pembuat saja.

Fitur pembeda dari MEMORY storage engine adalah, tidak mendukung tipe data yang memiliki panjang bervariasi (*variable-length types*) seperti BLOB dan TEXT. Tipe data VARCHAR masih bisa dipakai karena secara internal diproses sebagai CHAR.

Karena menggunakan panjang tipe data yang tetap dan berada di memory RAM, MEMORY tabel memiliki keunggulan bisa diakses dengan cepat.

Berikut contoh pendefinisian MEMORY tabel:

```
CREATE TABLE contoh_memory_1 (
    a INT,
    b VARCHAR(10)
) ENGINE = MEMORY;

SHOW CREATE TABLE contoh_memory_1 \G
*****
Table: contoh_memory_1
Create Table: CREATE TABLE `contoh_memory_1` (
    `a` int(11) DEFAULT NULL,
    `b` varchar(10) DEFAULT NULL
) ENGINE=MEMORY DEFAULT CHARSET=latin1
```

Sebagai pembuktian ciri khas tabel MEMORY, mari kita coba input beberapa data:

```
INSERT INTO contoh_memory_1 VALUES (1, 'merah'),(10, 'biru'),(100, 'kuning');

SELECT * FROM contoh_memory_1;
+-----+-----+
| a    | b    |
+-----+-----+
|    1 | merah |
|   10 | biru  |
| 100 | kuning|
+-----+-----+
```

Kemudian, restart MySQL Server dengan cara men-klik tombol Stop dari XAMPP Control Panel, lalu klik kembali tombol Start.

Selanjutnya, jalankan query SELECT:

```
MariaDB [belajar]> SELECT * FROM contoh_memory_1;

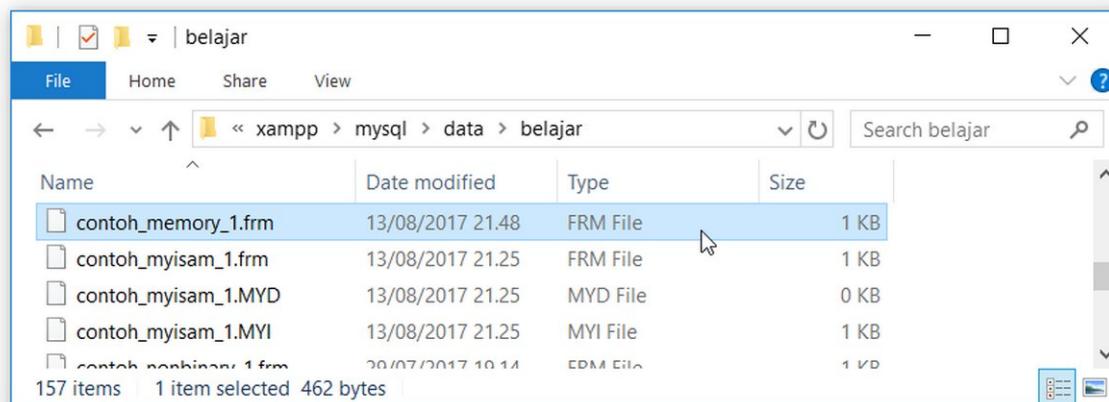
ERROR 2006 (HY000): MySQL server has gone away
No connection. Trying to reconnect...
Connection id:      2
Current database: belajar

Empty set (0.03 sec)
```

ERROR 2006 tampil akibat saya mematikan server MySQL dan menghidupkannya kembali. Hasil dari query `SELECT` sendiri ada di baris terakhir, yakni `Empty set (0.03 sec)`. Artinya, isi tabel `contoh_memory_1` sudah terhapus.

Namun tabel `contoh_memory_1` sendiri masih ada karena dibuat ulang saat server mulai berjalan.

Jika anda memeriksa file penyimpanan database `belajar` di `C:\xampp\mysql\data\belajar`, akan terdapat file `contoh_memory_1.frm`. File ini berisi struktur tabel `contoh_memory_1`, namun isi dari tabel disimpan ke dalam RAM, bukan di harddisk. Karena itulah isi tabel otomatis terhapus ketika server dimatikan.



Gambar: File untuk MEMORY storage engine

MEMORY storage engine tidak mendukung tipe data TEXT dan BLOG. Jika kita mendefinisikan tipe data ini, akan tampil pesan error:

```
CREATE TABLE contoh_memory_2 (
    a INT,
    b TEXT
) ENGINE=MEMORY;
```

```
ERROR 1163 (42000): Storage engine MEMORY doesn't support BLOB/TEXT columns
```

MEMORY storage engine sebelumnya dikenal juga sebagai bernama **HEAP storage engine**.

CSV Storage Engine

CSV adalah singkatan dari *Comma Separated Values*. Sesuai dengan namanya, data tabel disimpan ke dalam file teks yang dipisah dengan tanda koma.

Keunggulan dari CSV adalah data tabel bisa langsung dibuka dari aplikasi teks editor seperti **Notepad++** maupun aplikasi spreadsheet seperti **Microsoft Excel**. Kekurangannya, CSV storage engine tidak mendukung index, tidak bisa diisi nilai NULL dan tidak mendukung transaction.

Berikut contoh pendefinisian tabel CSV :

```
CREATE TABLE contoh_csv_1 (
    a INT,
    b VARCHAR(10)
) ENGINE = CSV;
```

```
ERROR 1178 (42000): The storage engine for the table doesn't support
nullable columns
```

Apa yang terjadi? Pesan error diatas tampil karena MySQL protes bahwa CSV storage engine tidak mendukung tabel dengan nilai NULL. Jika anda masih ingat, secara default seluruh kolom MySQL bisa diisi dengan nilai NULL, kecuali ditulis atribut NOT NULL.

Mari kita tambahkan atribut NOT NULL:

```
CREATE TABLE contoh_csv_1 (
    a INT NOT NULL,
    b VARCHAR(10) NOT NULL
) ENGINE = CSV;
```

```
SHOW CREATE TABLE contoh_csv_1 \G
*****
Table: contoh_csv_1
Create Table: CREATE TABLE `contoh_csv_1` (
    `a` int(11) NOT NULL,
    `b` varchar(10) NOT NULL
) ENGINE=CSV DEFAULT CHARSET=latin1
```

Baik, tabel contoh_csv_1 sudah berhasil dibuat. Agar bisa melihat langsung isi dari file CVS, saya akan input beberapa data:

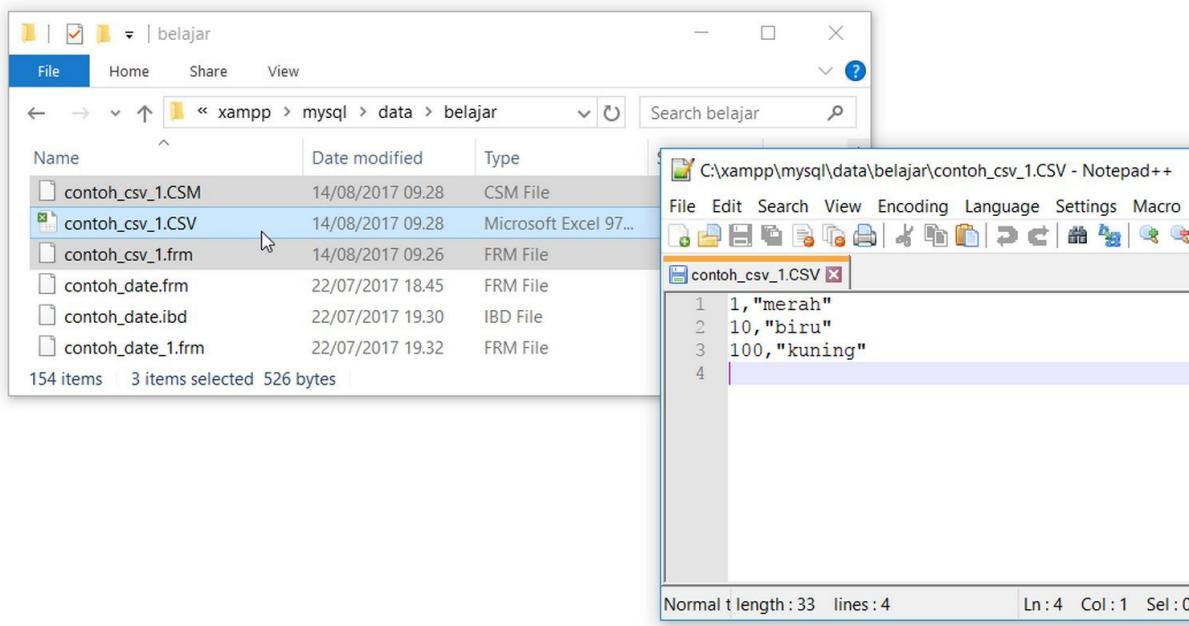
```
INSERT INTO contoh_csv_1 VALUES (1, 'merah'),(10, 'biru'),(100, 'kuning');
```

```
Query OK, 3 rows affected (0.07 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

Selanjutnya, buka folder C:\xampp\mysql\data\belajar. Anda akan menemukan 3 file dengan nama contoh_csv_1:

- **contoh_csv_1.CSM**: menyimpan metadata tentang tabel, seperti jumlah total baris.
- **contoh_csv_1.CSV**: menyimpan data / isi tabel.
- **contoh_csv_1.frm**: menyimpan pendefinisian struktur tabel.

File contoh_csv_1.CSV bisa anda buka langsung dari **Notepad++** atau **Microsoft Excel**, isinya berupa data tabel dengan pemisah tanda koma.



Gambar: Tampilan file contoh_csv_1.CSV di Notepad++

Format CSV biasa dipakai sebagai file penyimpanan “universal” dan sebagai sarana export data antar aplikasi. Hampir semua aplikasi database dan spreadsheet mendukung export dan import dalam format CSV.

Aria Storage Engine

Aria adalah storage engine yang khusus ada di MariaDB, namun tidak menutup kemungkinan akan tersedia di MySQL.

Aria merupakan versi perbaikan dari MyISAM. Storage engine ini memiliki fitur sudah mendukung *transaction* (namun belum aktif secara default), memiliki tipe data *OpenGIS* dan *crash safe* (tabel otomatis di perbaiki jika terjadi crash / server mati tiba-tiba).

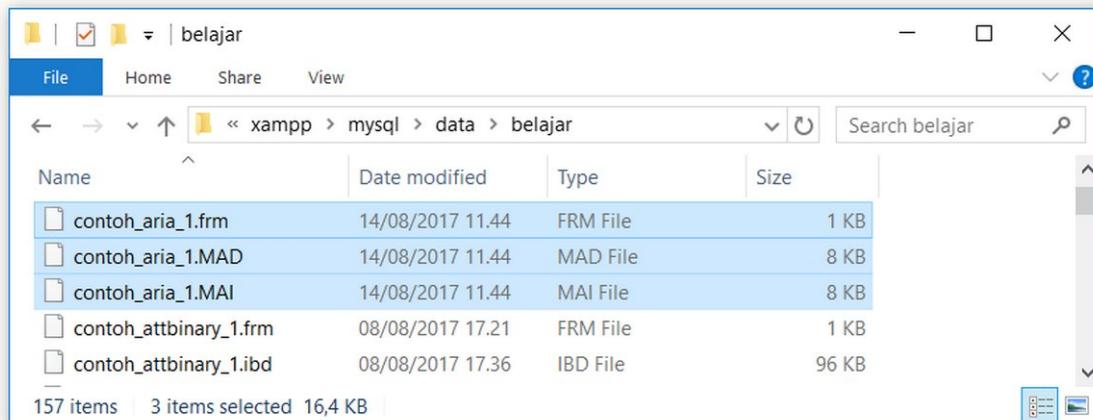
Kedepannya, Aria dirancang sebagai storage engine default di MariaDB. Berikut contoh pendefinisiannya:

```
CREATE TABLE contoh_aria_1 (
    a INT,
    b VARCHAR(10)
) ENGINE = Aria;

SHOW CREATE TABLE contoh_aria_1 \G
*****
Table: contoh_aria_1
Create Table: CREATE TABLE `contoh_aria_1` (
    `a` int(11) DEFAULT NULL,
    `b` varchar(10) DEFAULT NULL
) ENGINE=Aria DEFAULT CHARSET=latin1 PAGE_CHECKSUM=1
```

Aria storage engine disimpan ke dalam 3 file:

- **contoh_aria_1.frm**: menyimpan pendefinisian struktur tabel.
- **contoh_aria_1.MAD**: menyimpan data / isi tabel.
- **contoh_aria_1.MAI**: menyimpan index tabel.



Gambar: File untuk Aria storage engine

Aria sebenarnya sudah dirancang oleh Monty sejak tahun 2007 (sewaktu masih di MySQL AB). Saat itu storage engine ini diberi nama **Maria**.

MySQL AB kemudian dibeli oleh Oracle dan Monty keluar dari MySQL untuk mendirikan MariaDB. Memiliki 2 nama “Maria”, yakni MariaDB database server dan Maria storage engine sering membuat bingung, oleh karena itu Maria storage engine diganti menjadi Aria.

11.9 TABLE OPTION ... AUTO_INCREMENT

TABLE OPTION adalah settingan atau pengaturan tambahan pada saat pembuatan tabel. Perintah ini ditulis setelah pendefinisian kolom tabel.

Berikut posisi peletakan table option di dalam query CREATE TABLE:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
  (create_definition,...)
  [table_options]
  [partition_options]
```

TABLE OPTION sepenuhnya opsional dan boleh tidak ditulis.

Sebenarnya kita telah mempelajari 3 pengaturan table option, yakni CHARACTER SET, COLLATION dan ENGINE. Semua pilihan ini ditulis setelah pendefinisian kolom.

Salah satu pengaturan yang bisa kita set adalah nomor awal dari AUTO_INCREMENT. Secara default, penomoran kolom AUTO_INCREMENT mulai dari angka 1, kemudian naik secara berurutan ke 2, 3, 4, dst.

Kita bisa mengatur nomor awal ini menggunakan TABLE OPTION. Berikut format dasarnya:

```
CREATE TABLE nama_tabel
  (create_definition,...)
  AUTO_INCREMENT = nomor_awal
```

Mari masuk ke contoh praktik, saya ingin membuat tabel dimana kolom auto increment mulai dari angka 500:

```
CREATE TABLE contoh_auto_increment_3 (
  a INT AUTO_INCREMENT PRIMARY KEY,
  b VARCHAR(10)
) AUTO_INCREMENT = 500;

INSERT INTO contoh_auto_increment_3 (b) VALUES ('merah'), ('biru'), ('kuning');

SELECT * FROM contoh_auto_increment_3;
+----+----+
| a   | b    |
+----+----+
| 500 | merah |
| 501 | biru  |
| 502 | kuning |
+----+----+
```

Terlihat kolom pertama dari contoh_auto_increment_3 mulai dari angka 500 dan menaik ke 501, 502, dst.

11.10 TABLE OPTION ... COMMENT

Terdapat pilihan untuk menambah komentar (*comment*) ke dalam struktur tabel. Komentar ini akan melekat ke struktur tabel, bukan ke dalam tabel (tidak menambah kolom baru).

Fungsi komentar sama seperti pada bahasa pemrograman pada umumnya, yakni sebagai tempat untuk menulis keterangan tambahan atau sekedar catatan pengingat.

Komentar akan diabaikan oleh MySQL dan baru terlihat saat menjalankan query SHOW CREATE TABLE. Berikut format dasarnya:

```
CREATE TABLE nama_tabel
(create_definition,...)
COMMENT='Isi komentar disini';
```

Mari kita lihat praktik membuat komentar ke dalam struktur tabel:

```
CREATE TABLE contoh_comment_1 (
    a INT AUTO_INCREMENT PRIMARY KEY,
    b VARCHAR(10)
) COMMENT = 'Ini adalah tabel untuk belajar membuat komentar';

SHOW CREATE TABLE contoh_comment_1 \G
*****
Table: contoh_comment_1
Create Table: CREATE TABLE `contoh_comment_1` (
    `a` int(11) NOT NULL AUTO_INCREMENT,
    `b` varchar(10) DEFAULT NULL,
    PRIMARY KEY (`a`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COMMENT='Ini adalah tabel
untuk belajar membuat komentar'
```

Dari hasil query SHOW CREATE TABLE, komentar akan terlihat di baris paling bawah.

11.11 Format Dasar ALTER TABLE

Setelah tabel dibuat, ada kalanya kita ingin mengubah struktur tabel. Apakah itu mengganti nama tabel, nama kolom, tipe data kolom, menambah kolom baru, menghapus kolom yang sudah ada, hingga mengganti *storage engine* dari sebuah tabel. Untuk keperluan ini, tersedia query ALTER.

Berikut format dasar dari query ALTER dalam MySQL / MariaDB:

```
ALTER TABLE nama_tabel [alter_specification [, alter_specification] ...]
```

alter_specification berisi struktur tabel yang ingin diubah. Mari akan bahas satu persatu.

11.12 ALTER TABLE ... RENAME

Query ALTER TABLE ... RENAME digunakan untuk mengubah nama tabel. Format penulisan query ini adalah sebagai berikut:

```
ALTER TABLE nama_tabel_lama RENAME [TO | AS] nama_tabel_baru
```

Perintah TO atau AS boleh ditulis dan boleh juga tidak. Keduanya hanya tambahan supaya sesuai dengan standar bahasa SQL. Dengan kata lain, ketiga perintah berikut dianggap sama:

- **ALTER TABLE** nama_tabel_lama **RENAME** nama_tabel_baru.
- **ALTER TABLE** nama_tabel_lama **RENAME** **TO** nama_tabel_baru.
- **ALTER TABLE** nama_tabel_lama **RENAME** **AS** nama_tabel_baru.

Ketiga perintah ini akan menukar `nama_tabel_lama` menjadi `nama_tabel_baru`. Mari kita lihat prakteknya:

```
CREATE TABLE contoh_alter_1 (
    a INT AUTO_INCREMENT PRIMARY KEY,
    b VARCHAR(10)
);

ALTER TABLE contoh_alter_1 RENAME TO contoh_alter_99;

DESC contoh_alter_99;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra       |
+-----+-----+-----+-----+-----+
| a     | int(11)   | NO   | PRI | NULL    | auto_increment |
| b     | varchar(10) | YES  |      | NULL    |             |
+-----+-----+-----+-----+-----+
```

```
DESC contoh_alter_1;
ERROR 1146 (42S02): Table 'belajar.contoh_alter_1' doesn't exist
```

Disini saya membuat tabel `contoh_alter_1` sebagai tabel awal, kemudian menggunakan query `ALTER TABLE ... RENAME` untuk mengubah namanya menjadi `contoh_alter_99`.

Hasil query `DESC contoh_alter_99` memperlihatkan bahwa tabel `contoh_alter_99` memiliki struktur yang sama persis dengan tabel `contoh_alter_1`. Sedangkan query `DESC contoh_alter_1` akan menghasilkan error dengan keterangan *tabel tidak ditemukan*. Karena tentu saja kita sudah mengubah namanya menjadi `contoh_alter_99`.

Query `ALTER TABLE ... RENAME` juga bisa dipakai untuk memindahkan sebuah tabel ke database lain. Caranya, dengan menulis *identifier qualifiers* lengkap beserta nama database-nya.

Sebagai contoh, saya ingin memindahkan tabel `contoh_alter_99` dari database `belajar` ke database `indonesia`. Agar query ini bisa berjalan, database `belajar` dan `indonesia` harus sudah tersedia. Berikut query yang diperlukan:

```
ALTER TABLE belajar.contoh_alter_99 RENAME TO indonesia.contoh_alter_10;
```

Perhatikan penulisan *identifier qualifiers* dari tabel lama ke tabel baru. *belajar.contoh_alter_99* artinya tabel *contoh_alter_99* yang berada di database *belajar*, yang saya pindahkan ke *indonesia.contoh_alter_10*.

Selain memindahkan tabel, saya juga mengubah namanya. Sekarang, yang ada di database *indonesia* bukan lagi tabel *contoh_alter_99*, tapi tabel *contoh_alter_10*.

Untuk memeriksanya, silahkan pindah sebentar ke database *indonesia* dan jalankan query **SHOW TABLES**:

```
MariaDB [belajar]> USE indonesia;
Database changed
```

```
MariaDB [indonesia]> SHOW TABLES;
+-----+
| Tables_in_indonesia |
+-----+
| contoh_alter_10      |
| provinsi             |
+-----+
```

```
MariaDB [indonesia]> DESC contoh_alter_10;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra       |
+-----+-----+-----+-----+-----+
| a     | int(11)   | NO   | PRI | NULL    | auto_increment |
| b     | varchar(10) | YES  |     | NULL    |               |
+-----+-----+-----+-----+-----+
```

Seperti yang terlihat, tabel *contoh_alter_99* sudah pindah ke database *indonesia* dengan nama tabel baru: *contoh_alter_10*.

Sebagai latihan, bisakah anda pindahkan kembali tabel *contoh_alter_10* ke database *belajar* dan mengubah namanya *contoh_alter_1*?

Berikut query yang dibutuhkan:

```
ALTER TABLE indonesia.contoh_alter_10 RENAME TO belajar.contoh_alter_1;
```

Query **ALTER TABLE ... RENAME** akan menghasilkan error jika nama tabel baru “bentrok” dengan nama tabel yang sudah ada:

```
ALTER TABLE contoh_alter_1 RENAME TO contoh_comment_1;
ERROR 1050 (42S01): Table 'contoh_comment_1' already exists
```

Error diatas terjadi karena di dalam database *belajar* sudah ada tabel *contoh_comment_1*.

11.13 RENAME TABLE ... TO

Selain menggunakan query ALTER TABLE ... RENAME, tersedia juga query RENAME TABLE ... TO yang sama-sama bisa dipakai untuk mengubah nama tabel.

Format dasar penulisan query RENAME TABLE ... TO adalah sebagai berikut:

```
RENAME TABLE nama_tabel_lama TO nama_tabel_baru;
```

Berikut contoh penggunaannya:

```
CREATE TABLE IF NOT EXISTS contoh_alter_1 (
    a INT AUTO_INCREMENT PRIMARY KEY,
    b VARCHAR(10)
);

RENAME TABLE contoh_alter_1 TO contoh_alter_20;

DESC contoh_alter_20;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra           |
+-----+-----+-----+-----+-----+
| a     | int(11)   | NO   | PRI | NULL    | auto_increment |
| b     | varchar(10)| YES  |      | NULL    |                 |
+-----+-----+-----+-----+-----+
```

Pertama, saya ingin memastikan tabel contoh_alter_1 sudah tersedia. Query CREATE TABLE IF NOT EXISTS contoh_alter_1 akan membuat tabel contoh_alter_1 jika belum ada. Selanjutnya, saya mengganti nama tabel contoh_alter_1 menjadi contoh_alter_20.

Query RENAME TABLE ... TO bisa dipakai untuk mengubah beberapa tabel sekaligus dalam 1 perintah query. Berikut percobaannya:

```
CREATE TABLE contoh_alter_21 LIKE contoh_alter_20;

RENAME TABLE
    contoh_alter_20 TO contoh_alter_1,
    contoh_alter_21 TO contoh_alter_2
;
```

Sebelum query RENAME, saya membuat tabel contoh_alter_21 terlebih dahulu. Dimana tabel ini dicopy dari contoh_alter_20. Semoga anda masih ingat dengan query CREATE TABLE ... LIKE.

Kemudian, query RENAME TABLE akan mengubah nama 2 tabel sekaligus. Tabel contoh_alter_20 menjadi contoh_alter_1, dan contoh_alter_21 menjadi contoh_alter_2.

Jika kita menggunakan query ALTER TABLE ... RENAME untuk hal yang sama, akan butuh 2 perintah query yang dijalankan secara terpisah.

11.14 ALTER TABLE ... TABLE OPTION

Perintah ALTER berikutnya yang akan kita bahas adalah mengubah TABLE OPTION dari sebuah tabel.

Sewaktu pembuatan tabel, perintah TABLE OPTION ini ditulis setelah pendefinisian kolom. Contohnya seperti ENGINE, CHARACTER SET, AUTO_INCREMENT dan COMMENT.

Format penulisan query ALTER TABLE ... TABLE OPTION adalah sebagai berikut:

```
ALTER TABLE nama_tabel table_options
```

Bagian TABLE OPTION umumnya terdiri dari dua bagian, yakni **nama option** dan **nilainya**, seperti CHARSET = utf8 atau ENGINE = MyISAM.

Sebagai contoh, saya ingin mengubah CHARSET tabel contoh_alter_1 menjadi utf8, berikut query yang dibutuhkan:

```
SHOW CREATE TABLE contoh_alter_1 \G
*****
Table: contoh_alter_1
Create Table: CREATE TABLE `contoh_alter_1` (
    `a` int(11) NOT NULL AUTO_INCREMENT,
    `b` varchar(10) DEFAULT NULL,
    PRIMARY KEY (`a`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1

ALTER TABLE contoh_alter_1 CHARSET = utf8;
```

```
SHOW CREATE TABLE contoh_alter_1 \G
*****
Table: contoh_alter_1
Create Table: CREATE TABLE `contoh_alter_1` (
    `a` int(11) NOT NULL AUTO_INCREMENT,
    `b` varchar(10) CHARACTER SET latin1 DEFAULT NULL,
    PRIMARY KEY (`a`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

Dari query SHOW CREATE TABLE, terlihat bahwa charset dari tabel contoh_alter_1 sudah berubah dari sebelumnya latin1 (default MySQL) menjadi utf8.

Bagaimana dengan mengubah *storage engine* jadi MyISAM? Caranya juga sama:

```
ALTER TABLE contoh_alter_1 ENGINE = MyISAM;

SHOW CREATE TABLE contoh_alter_1 \G
*****
Table: contoh_alter_1
Create Table: CREATE TABLE `contoh_alter_1` (
  `a` int(11) NOT NULL AUTO_INCREMENT,
  `b` varchar(10) CHARACTER SET latin1 DEFAULT NULL,
  PRIMARY KEY (`a`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8
```

Sekarang tabel contoh_alter_1 sudah berubah dari InnoDB ke MyISAM.

Salah satu pengaturan yang juga sering diubah menggunakan TABLE OPTION adalah untuk mengatur angka **auto increment**:

```
INSERT INTO contoh_alter_1 (b) VALUES ('putih'), ('hitam');
```

```
SELECT * FROM contoh_alter_1;
+---+-----+
| a | b      |
+---+-----+
| 1 | putih |
| 2 | hitam |
+---+-----+
```

```
ALTER TABLE contoh_alter_1 AUTO_INCREMENT = 50;
```

```
INSERT INTO contoh_alter_1 (b) VALUES ('biru'), ('merah');
```

```
SELECT * FROM contoh_alter_1;
+---+-----+
| a | b      |
+---+-----+
| 1 | putih |
| 2 | hitam |
| 50 | biru   |
| 51 | merah |
+---+-----+
```

Saya menginput beberapa data ke tabel contoh_alter_1. Karena kolom a sudah di set sebagai AUTO INCREMENT, nilai kolom a akan menaik dari 1, 2, dst.

Menggunakan query ALTER, saya mengubah nomor AUTO INCREMENT menjadi 50. Dengan demikian, penomoran auto increment akan ‘lompat’ ke angka 50.

Untuk membuktikannya, saya menginput kembali beberapa data. Terlihat, nomor auto increment akan mulai dari 50, 51, dst.

11.15 ALTER TABLE ... MODIFY

Berikutnya, kita masuk ke query untuk mengubah tipe data dan atribut dari sebuah kolom. Query yang digunakan adalah `ALTER TABLE ... MODIFY`, dengan format penulisan sebagai berikut:

```
ALTER TABLE nama_tabel MODIFY nama_kolom tipe_kolom [atribut_kolom];
```

Sebagai contoh, saya ingin mengubah kolom a menjadi `TINYINT UNSIGNED`, berikut penulisan querynya:

```
ALTER TABLE contoh_alter_1 MODIFY a TINYINT UNSIGNED;
```

```
DESC contoh_alter_1;
```

Field	Type	Null	Key	Default	Extra
a	tinyint(3) unsigned	NO	PRI	NULL	
b	varchar(10)	YES		NULL	

Sekarang, kolom a sudah berubah tipe data menjadi `TINYINT UNSIGNED`.

Perhatikan juga hasil dari query `DESC contoh_alter_1`, kolom a kehilangan atribut `auto increment`. Jika kita ingin kolom a tetap sebagai *auto increment*, atribut ini harus ditulis ulang di dalam query `ALTER`:

```
ALTER TABLE contoh_alter_1 MODIFY a TINYINT UNSIGNED AUTO_INCREMENT;
```

```
DESC contoh_alter_1;
```

Field	Type	Null	Key	Default	Extra
a	tinyint(3) unsigned	NO	PRI	NULL	auto_increment
b	varchar(10)	YES		NULL	

Penulisan query `ALTER TABLE ... MODIFY` juga bisa disambung untuk mengubah 2 kolom sekaligus (atau lebih). Berikut contoh querynya:

```
ALTER TABLE contoh_alter_1
MODIFY a MEDIUMINT(3) ZEROFILL,
MODIFY b CHAR(20)
;

DESC contoh_alter_1;
+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+
| a     | mediumint(3) unsigned zerofill | NO   | PRI | NULL    |       |
| b     | char(20)           | YES  |     | NULL    |       |
+-----+-----+-----+-----+
```

Terlihat, baik kolom a maupun kolom b sudah bertukar tipe data.

11.16 ALTER TABLE ... CHANGE

Query **ALTER TABLE ... CHANGE** bisa digunakan untuk mengubah nama kolom sekaligus tipe datanya. Berikut format dasar dari query ini:

```
ALTER TABLE nama_tabel
CHANGE nama_kolom nama_kolom_baru tipe_kolom [atribut_kolom];
```

Sebagai contoh, jika saya ingin mengubah nama kolom a menjadi c dan tipe datanya menjadi TINYINT, query yang dipakai adalah sebagai berikut:

```
ALTER TABLE contoh_alter_1 CHANGE a c TINYINT;
```

```
DESC contoh_alter_1;
+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+
| c     | tinyint(4) | NO   | PRI | NULL    |       |
| b     | char(20)   | YES  |     | NULL    |       |
+-----+-----+-----+-----+
```

Sekarang, kolom a sudah berubah menjadi c dan tipe datanya menjadi TINYINT.

Dalam MySQL, kita tidak bisa mengubah nama kolom tanpa mencantumkan kembali tipe datanya. Untuk memeriksa apa tipe data kolom saat ini, bisa menggunakan query **SHOW CREATE TABLE**.

Sebagai contoh, saya ingin mengganti nama kolom b menjadi d, tapi tipe datanya tetap. Untuk keperluan ini, saya akan memeriksa tipe data kolom b terlebih dahulu:

```
SHOW CREATE TABLE contoh_alter_1 \G
*****
1. row *****

Table: contoh_alter_1
Create Table: CREATE TABLE `contoh_alter_1` (
  `c` tinyint(4) NOT NULL,
  `b` char(20) DEFAULT NULL,
  PRIMARY KEY (`c`)
) ENGINE=MyISAM AUTO_INCREMENT=52 DEFAULT CHARSET=utf8

ALTER TABLE contoh_alter_1 CHANGE b d char(20) DEFAULT NULL;

DESC contoh_alter_1;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| c     | tinyint(4) | NO   | PRI | NULL    |       |
| d     | char(20)   | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

Dari hasil query SHOW CREATE TABLE, terlihat bahwa kolom b bertipe char(20) DEFAULT NULL. Karena saya ingin mengganti nama kolom b menjadi d, maka tipe data ini akan ditulis ulang.

Atribut DEFAULT NULL sebenarnya boleh tidak ditulis, karena atribut ini sudah menjadi nilai default dari hampir semua tipe data kolom dalam MySQL.

Dalam kasus lain, apabila ingin mengganti tipe data kolom tanpa mengubah namanya, nama kolom lama dan nama kolom baru tetap harus ditulis. Hanya saja keduanya akan bernilai sama.

Sebagai contoh, saya ingin mengganti tipe data untuk kolom d menjadi VARCHAR(50) dengan tambahan atribut CHARSET utf8, querynya adalah sebagai berikut:

```
ALTER TABLE contoh_alter_1 CHANGE d d VARCHAR(20) CHARSET utf8;
```

```
DESC contoh_alter_1;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| c     | tinyint(4) | NO   | PRI | NULL    |       |
| d     | varchar(20) | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
```

Perhatikan penulisan kolom dari query ALTER TABLE ... CHANGE, dimana saya menulis dua kali nama kolom: d d VARCHAR(20) CHARSET utf8. Nama d pertama sebagai kolom lama, dan d kedua sebagai nama kolom baru.

Kita juga bisa menggunakan query ALTER TABLE ... CHANGE untuk mengubah beberapa kolom sekaligus dalam 1 perintah query.

Sebagai contoh, saya ingin mengubah nama kolom c dan d menjadi a dan b. Berikut query yang bisa dipakai:

```
ALTER TABLE contoh_alter_1
    CHANGE c a INT AUTO_INCREMENT,
    CHANGE d b VARCHAR(10)
;

DESC contoh_alter_1;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra       |
+-----+-----+-----+-----+-----+
| a     | int(11)   | NO   | PRI | NULL    | auto_increment |
| b     | varchar(10) | YES  |     | NULL    |             |
+-----+-----+-----+-----+-----+
```

Sekarang, tabel contoh_alter_1 kembali memiliki 2 kolom bernama a dan b.

Hati-Hati Dengan Perubahan Kolom

Mengubah tipe data kolom ke tipe data lain bisa menyebabkan masalah. Sebagai contoh, mengubah kolom dari BIGINT ke TINYINT berpotensi “memotong” data yang ada. Karena jangkauan tipe data TINYINT jauh lebih kecil daripada BIGINT.

Jika MySQL berjalan dalam **strict mode**, perubahan seperti ini akan menghasilkan error dan query gagal berjalan. Akan tetapi, error ini baru tampil ada data yang harus dipotong. Apabila data yang ada masih bisa diakomodasi oleh tipe data baru, tidak akan tampil pesan error.

Di dalam **mode normal**, akan terjadi proses konversi. Dimana data yang lama dipotong agar muat ke tipe data baru.

Agar lebih paham, mari kita coba membuat beberapa contoh kasus. Pertama, saya akan menginput nilai 500 ke dalam kolom a, kemudian mengubah tipe data kolom a menjadi TINYINT:

```
DROP TABLE contoh_alter_1;

CREATE TABLE contoh_alter_1 (
    a INT,
    b VARCHAR(10)
);

INSERT INTO contoh_alter_1 VALUES (500, 'putih');

SELECT * FROM contoh_alter_1;
+-----+
| a    | b    |
+-----+
```

```
| 500 | putih |
+-----+-----+
```

```
ALTER TABLE contoh_alter_1 MODIFY a TINYINT;
ERROR 1264 (22003): Out of range value for column 'a' at row 1
```

Di awal, saya menghapus tabel contoh_alter_1 dan membuat ulang tabel tersebut. Perhatikan bahwa kolom a bertipe INT dan kolom b bertipe VARCHAR(10).

Kemudian saya menginput angka 500 ke kolom a, dan string putih ke kolom b. Tidak ada masalah. Kedua data sesuai dengan tipe data dan terlihat dari hasil query SELECT.

Ketika kolom a saya coba ubah ke TINYINT, hasilnya akan error: *Out of range value for column 'a' at row 1*. Ini terjadi karena angka 500 dari kolom a tidak bisa ditampung oleh kolom TINYINT. Jika anda masih ingat, nilai maksimum untuk kolom TINYINT adalah 127.

Akan tetapi, jika kita mencoba mengubah kolom b dari VARCHAR(10) ke CHAR(5), tidak akan menjadi masalah:

```
ALTER TABLE contoh_alter_1 MODIFY b CHAR(5);
```

```
DESC contoh_alter_1;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| a     | int(11)   | YES  |     | NULL    |       |
| b     | char(5)   | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
```

Kenapa bisa jalan? Padahal CHAR(5) juga lebih kecil daripada VARCHAR(10)?

Ini terjadi karena data saat ini yang ada di kolom b tidak ada lebih dari 5 karakter. String putih terdiri dari 5 karakter sehingga masih bisa ditampung oleh CHAR(5) dan tidak terjadi “pemotongan data”.

Error untuk kolom a seperti yang kita dapati sebelum ini hanya terjadi karena MySQL aktif dalam **strict mode**. Bagaimana apabila hal yang sama di lakukan dalam mode ‘normal’? mari kita coba:

```
SELECT @@SESSION.sql_mode;
+-----+
| @@SESSION.sql_mode          |
+-----+
| STRICT_ALL_TABLES,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+-----+
```

```
SET sql_mode = '';
```

```
SELECT @@SESSION.sql_mode;
+-----+
| @@SESSION.sql_mode |
+-----+
|                   |
+-----+  
  
ALTER TABLE contoh_alter_1 MODIFY a TINYINT;
Query OK, 1 row affected, 1 warning (0.78 sec)
Records: 1 Duplicates: 0 Warnings: 1  
  
SELECT * FROM contoh_alter_1;
+-----+
| a    | b    |
+-----+
| 127 | putih |
+-----+
```

Saya memeriksa SQL mode menggunakan perintah `SELECT @@SESSION.sql_mode`, hasilnya terdapat pengaturan `STRICT_ALL_TABLES`. Ini menandakan selama ini kita masuk ke *strict mode*. Pengaturan strict mode ini kita buat dalam bab sebelumnya.

Selanjutnya saya menjalankan query `SET sql_mode = ''`. Perintah ini akan menghapus seluruh pengaturan SQL Mode. Dengan kata lain, kita sudah keluar dari strict mode. Untuk memastikan, saya menjalankan query `SELECT @@SESSION.sql_mode` sekali lagi. Seperti yang bisa ditebak, tidak ada pengaturan yang aktif.

Kita masuk ke contoh kasus. Saya kembali mencoba mengubah tipe data kolom `a` menjadi `TINYINT`, dan berhasil!

Hanya saja, dari tampilan query `SELECT`, Angka 500 akan dikonversi menjadi 127. Angka 127 ini sendiri merupakan nilai maksimal dari tipe data `TINYINT`.

Perubahan seperti ini harap menjadi pertimbangan ketika mengubah kolom ke tipe data yang lebih kecil. Idealnya, kita hanya mengubah tipe data jika hal tersebut tidak berpengaruh ke data yang ada.

Silahkan anda logout sebentar dari MySQL client agar pengaturan strict mode kembali aktif.

11.17 ALTER TABLE ... ADD COLUMN

Query `ALTER TABLE ... ADD COLUMN` digunakan untuk menambah kolom baru ke dalam tabel. Format dasar penulisannya adalah sebagai berikut:

```
ALTER TABLE nama_tabel
  ADD [COLUMN] nama_kolom tipe_kolom [atribut_kolom]
  [FIRST | AFTER nama_kolom]
```

Kita akan bahas berbagai variasi dari query **ALTER TABLE ... ADD COLUMN**. Perintah di dalam tanda kurung siku bersifat opsional dan boleh tidak ditulis.

Sebagai contoh pertama, saya ingin menambah kolom c ke dalam tabel `contoh_alter_1`. Kolom c ini memiliki tipe data `DECIMAL(4,2)` dengan atribut `NOT NULL`. Berikut query-nya:

```
ALTER TABLE contoh_alter_1 ADD c DECIMAL(4,2) NOT NULL;
```

```
DESC contoh_alter_1;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| a     | tinyint(4) | YES  |     | NULL    |       |
| b     | char(5)    | YES  |     | NULL    |       |
| c     | decimal(4,2)| NO   |     | NULL    |       |
+-----+-----+-----+-----+-----+
```

Jika tidak memiliki perintah tambahan, kolom c akan berada di kolom paling kanan (kolom terakhir). Bagaimana jika kita ingin agar kolom baru ini ditempatkan di sebelah kiri sebagai kolom pertama? Tambahkan perintah `FIRST`.

Kali ini saya ingin menambah kolom d dengan tipe data `BINARY` sebagai kolom pertama. Query yang diperlukan adalah sebagai berikut:

```
ALTER TABLE contoh_alter_1 ADD d BINARY FIRST;
```

```
DESC contoh_alter_1;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| d     | binary(1) | YES  |     | NULL    |       |
| a     | tinyint(4) | YES  |     | NULL    |       |
| b     | char(5)    | YES  |     | NULL    |       |
| c     | decimal(4,2)| NO   |     | NULL    |       |
+-----+-----+-----+-----+-----+
```

Dengan menambahkan perintah `FIRST` di akhir query, kolom d akan menjadi kolom pertama.

Kombinasi lain dari query **ALTER TABLE ... ADD COLUMN** adalah menempatkan kolom di posisi tertentu. Untuk ini kita butuh perintah `AFTER`.

Sebagai contoh, saya ingin menambah kolom e dengan tipe data `BIGINT UNSIGNED` ke dalam tabel `contoh_alter_1`. Kolom e ditempatkan setelah kolom b. Berikut penulisan querynya:

```
ALTER TABLE contoh_alter_1 ADD e BIGINT UNSIGNED AFTER b;
```

```
DESC contoh_alter_1;
```

Field	Type	Null	Key	Default	Extra
d	binary(1)	YES		NULL	
a	tinyint(4)	YES		NULL	
b	char(5)	YES		NULL	
e	bigint(20) unsigned	YES		NULL	
c	decimal(4,2)	NO		NULL	

Dari hasil query **DESC** contoh_alter_1, bisa terlihat bahwa kolom e berada setelah kolom b.

Kita juga bisa menambahkan perintah **COLUMN** setelah **ADD**. Kedua penulisan query berikut dianggap sama:

- **ALTER TABLE** contoh_alter_1 **ADD** c DECIMAL(4,2) NOT NULL.
- **ALTER TABLE** contoh_alter_1 **ADD COLUMN** c DECIMAL(4,2) NOT NULL.

Variasi penulisan ini agar sesuai dengan standar bahasa SQL.

Terakhir, query **ALTER TABLE** ... **ADD COLUMN** bisa digunakan untuk menambah banyak kolom dalam 1 kali penulisan. Berikut contohnya:

```
ALTER TABLE contoh_alter_1
  ADD COLUMN f CHAR(2) FIRST,
  ADD COLUMN g TIME(4) AFTER c
;
```

```
DESC contoh_alter_1;
```

Field	Type	Null	Key	Default	Extra
f	char(2)	YES		NULL	
d	binary(1)	YES		NULL	
a	tinyint(4)	YES		NULL	
b	char(5)	YES		NULL	
e	bigint(20) unsigned	YES		NULL	
c	decimal(4,2)	NO		NULL	
g	time(4)	YES		NULL	

Dalam query diatas, saya menambah 2 kolom: f dan g ke dalam tabel contoh_alter_1. Kolom f akan ditempatkan di sisi paling kiri karena terdapat perintah **FIRST**. Sedangkan kolom g berada di paling kanan karena tidak memiliki perintah tambahan apapun.

11.18 ALTER TABLE ... DROP COLUMNS

Query ini adalah kebalikan dari query sebelumnya. Perintah ALTER TABLE ... DROP COLUMNS digunakan untuk menghapus kolom dari sebuah tabel.

Berikut format dasar penulisan query ALTER TABLE ... DROP COLUMNS:

```
ALTER nama_tabel DROP [COLUMN] nama_kolom [RESTRICT | CASCADE]
```

Sebagai contoh, saya ingin menghapus kolom g dari tabel contoh_alter_1. Berikut contoh querinya:

```
ALTER TABLE contoh_alter_1 DROP COLUMN g;
```

Perintah COLUMN bersifat opsional dan boleh tidak ditulis. Query diatas juga sama-sama akan menghapus kolom g jika ditulis menjadi:

```
ALTER TABLE contoh_alter_1 DROP g;
```

Query ALTER TABLE ... DROP COLUMNS juga bisa disambung untuk menghapus banyak kolom sekaligus. Sebagai contoh, saya ingin menghapus kolom c, d, e dan f dari contoh_alter_1:

```
ALTER TABLE contoh_alter_1
  DROP COLUMN c,
  DROP COLUMN d,
  DROP COLUMN e,
  DROP COLUMN f
;

DESC contoh_alter_1;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| a     | tinyint(4) | YES  |      | NULL    |       |
| b     | char(5)    | YES  |      | NULL    |       |
+-----+-----+-----+-----+-----+
```

Sekarang, tabel contoh_alter_1 sudah kembali ke kondisi semula, yakni hanya memiliki 2 kolom: a dan b.

Dari format penulisan query ALTER TABLE ... DROP COLUMNS , terdapat perintah tambahan RESTRICT dan CASCADE. Kedua perintah ini tidak berefek apapun dan hanya digunakan agar sesuai dengan standar bahasa SQL.

11.19 Format Dasar DROP TABLE

Query DROP TABLE bisa dipakai untuk menghapus tabel. Query ini sangat powerfull dan cukup “berbahaya”. MySQL tidak akan memberikan konfirmasi saat sebuah tabel dihapus. Selain itu tidak ada tombol “undo” untuk mengembalikan tabel ke kondisi sebelumnya.

Berikut format dasar dari query DROP TABLE:

```
DROP [TEMPORARY] TABLE [IF EXISTS]
  nama_tabel [, nama_tabel] ...
  [RESTRICT | CASCADE]
```

Sama seperti penjelasan untuk query lain, perintah di dalam tanda kurung siku merupakan perintah opsional dan boleh tidak dituliskan.

Khusus perintah tambahan RESTRICT dan CASCADE, keduanya tidak memiliki efek apapun. Kedua perintah ini disediakan agar memudahkan proses import dari sistem database lain.

11.20 DROP TABLE

Kita sudah berulang kali menggunakan query DROP TABLE untuk menghapus tabel. Berikut contohnya:

```
CREATE TABLE contoh_drop_1 (
  a INT
);
```

```
DROP TABLE contoh_drop_1;
```

Query DROP TABLE juga bisa digunakan untuk menghapus beberapa tabel dalam 1 perintah. Nama tabel yang ingin dihapus dipisah dengan tanda koma. Berikut contoh penggunaannya:

```
CREATE TABLE contoh_drop_1 (
  a INT
);
```

```
CREATE TABLE contoh_drop_2 LIKE contoh_drop_1;
```

```
DROP TABLE contoh_drop_1, contoh_drop_2;
Query OK, 0 rows affected (0.38 sec)
```

Disini saya membuat tabel contoh_drop_1 dan contoh_drop_2. Kemudian menggunakan 1 query DROP TABLE untuk menghapus kedua tabel.

Query DROP TABLE juga memiliki perintah tambahan DROP TEMPORARY TABLE dan DROP TABLE IF EXISTS.

Query DROP TEMPORARY TABLE khusus digunakan untuk menghapus *temporary table*, seperti contoh berikut:

```
CREATE TEMPORARY TABLE contoh_temporary_1 (
    a INT AUTO_INCREMENT PRIMARY KEY,
    b VARCHAR(10)
);
```

```
DROP TEMPORARY TABLE contoh_temporary_1;
Query OK, 0 rows affected (0.12 sec)
```

Dengan tambahan perintah TEMPORARY, kita bisa memastikan bahwa tabel yang dihapus adalah tabel *temporary*, bukan tabel reguler.

Query `DROP TABLE IF EXISTS` bisa dipakai untuk menyembunyikan pesan error jika tabel yang akan dihapus ternyata sudah tidak ada. Tanpa perintah ini, MySQL akan mengeluarkan error. Berikut percobaannya:

```
DROP TABLE tidak_ada;
ERROR 1051 (42S02): Unknown table 'belajar.tidak_ada'
```

```
DROP TABLE IF EXISTS tidak_ada;
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

```
SHOW WARNINGS;
+-----+-----+
| Level | Code | Message           |
+-----+-----+
| Note  | 1051 | Unknown table 'belajar.tidak_ada' |
+-----+-----+
```

Terlihat bahwa query `DROP TABLE IF EXISTS` akan mengkonversi pesan error menjadi warning.

11.21 Latihan Pembuatan Tabel

Sebagai penutup bab tentang membuat, mengubah dan menghapus tabel, saya sudah siapkan latihan untuk menguji pemahaman anda seputar materi yang telah kita pelajari.

Silahkan buka cmd MySQL client dan selesaikan 9 instruksi berikut:



Untuk memeriksa struktur tabel apakah sudah sesuai atau belum, bisa menggunakan query `SHOW CREATE TABLE` dan `DESCRIBE`.

- Buat tabel `latihan_1` dengan dua kolom, `a` dan `b` dengan spesifikasi:
 - Kolom `a` di definisikan sebagai `INTEGER`, memiliki atribut `UNSIGNED`, di set sebagai `PRIMARY KEY` dan penomoran otomatis `AUTO_INCREMENT`.

- Kolom b di definisikan sebagai VARCHAR yang sanggup menampung maksimal 10 karakter. Jika kolom ini tidak diisi, string ‘merdeka’ akan menjadi string default. Kolom b juga tidak bisa diisi nilai NULL.
 - Tabel latihan_1 menggunakan storage engine MyISAM serta charset ascii. Nomor urut auto increment dimulai dari 100.
 - Silahkan buat tabel latihan_1 dengan spesifikasi diatas sebelum lanjut ke nomor 2.
2. Ubah storage engine tabel latihan_1 menjadi InnoDB.
 3. Ubah nama kolom b menjadi c dari tabel latihan_1, dimana kolom c ini juga memiliki tipe data yang sama dengan kolom b (termasuk atribut).
 4. Tambah kolom baru bernama b ke dalam tabel latihan_1. Kolom b di definisikan dengan tipe data DATE. Jika kolom b tidak diisi nilai, tanggal 01-01-2018 akan menjadi nilai default. Tempatkan kolom b setelah kolom a. Artinya, setelah langkah ini dijalankan tabel latihan_1 akan memiliki 3 kolom: a, b dan c.
 5. Buat tabel latihan_2 yang memiliki struktur sama persis seperti tabel latihan_1.
 6. Ubah storage engine tabel latihan_2 menjadi MEMORY.
 7. Ubah nama tabel latihan_2 menjadi latihan_3.
 8. Buat temporary table dengan nama latihan_4. Tabel ini berisi 1 kolom a sebagai TIMESTAMP.
 9. Hapus tabel latihan_1, latihan_3 dan latihan_4 dalam 1 perintah query.

11.22 Jawaban Latihan Pembuatan Tabel

Jika anda merasa kesulitan atau sekedar menyamakan jawaban, berikut query yang saya pakai untuk latihan sebelumnya:

```

1.
CREATE TABLE latihan_1 (
    a INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    b VARCHAR(10) DEFAULT 'merdeka' NOT NULL
) ENGINE = MyISAM CHARSET=ascii AUTO_INCREMENT = 100;

2.
ALTER TABLE latihan_1 ENGINE = InnoDB;

3.
ALTER TABLE latihan_1 CHANGE b c VARCHAR(10) DEFAULT 'merdeka' NOT NULL;

4.
ALTER TABLE latihan_1 ADD b DATE DEFAULT 20180101 AFTER a;

5.
CREATE TABLE latihan_2 LIKE latihan_1;

6.

```

```
ALTER TABLE latihan_2 ENGINE = MEMORY;

7.
RENAME TABLE latihan_2 TO latihan_3;

8.
CREATE TEMPORARY TABLE latihan_4 (
    a TIMESTAMP
);

9.
DROP TABLE latihan_1, latihan_3, latihan_4;
```

Dalam bab ini kita membahas berbagai query terkait pembuatan, mengubah dan menghapus tabel. Dengan bekal kemampuan ini, anda sudah bisa membuat tabel untuk berbagai keperluan.

12. Menginput Data Tabel

Dalam bab ini kita akan mempelajari query untuk menginput data ke dalam tabel. Query yang dimaksud adalah `INSERT`.

Sepanjang pembahasan bab-bab sebelumnya, saya sudah beberapa kali menggunakan query `INSERT`. Kali ini akan dibahas perintah tambahan yang bisa dipakai untuk menginput data ke dalam database.

Setidaknya terdapat 3 bentuk dari query `INSERT`:

- `INSERT ... VALUE`
- `INSERT ... SET`
- `INSERT ... SELECT`

Kita akan bahas satu persatu.



Kecuali dinyatakan lain, seluruh contoh tabel di input ke dalam database `belajar`. Saya berasumsi anda sudah menjalankan query `USE DATABASE belajar` agar bisa mengikuti materi dalam bab ini.

12.1 INSERT ... VALUE

Query `INSERT ... VALUE` adalah bentuk paling sederhana, sekaligus yang paling banyak kita pakai. Format dasar penulisannya sebagai berikut:

```
INSERT [INTO] nama_tabel [nama_kolom] VALUES (nilai1, nilai2, ...)
```

Terlihat bahwa perintah `INTO` sebenarnya bersifat opsional (boleh tidak ditulis), begitu juga dengan `nama_kolom` tabel. Kita akan lihat berbagai variasi perintah ini.

Sebagai tabel praktik, saya akan membuat tabel `contoh_insert_1`:

```
CREATE TABLE contoh_insert_1 (
    a INT,
    b VARCHAR(10),
    c DATE
);
```

Tabel `contoh_insert_1` di definisikan dengan 3 kolom: `a`, `b` dan `c`. Kolom `a` di set dengan tipe data `INT`, `b` sebagai `VARCHAR(10)` dan kolom `c` sebagai `DATE`.

Variasi pertama dari query `INSERT ... VALUE` adalah tanpa menyertakan nama kolom:

```
INSERT contoh_insert_1 VALUES (10, 'merah', 20180101);
Query OK, 1 row affected (0.06 sec)
```

```
SELECT * FROM contoh_insert_1;
+-----+-----+-----+
| a    | b    | c    |
+-----+-----+-----+
| 10  | merah | 2018-01-01 |
+-----+-----+-----+
```

Jika query **INSERT** ditulis tanpa menyebutkan nama kolom, kita harus mengisi data untuk semua kolom dan secara berurutan. Tabel **contoh_insert_1** memiliki 3 kolom, oleh karena itu saya harus menulis 3 nilai secara berurutan. Nilai pertama untuk kolom **a**, nilai kedua untuk kolom **b**, dan nilai ketiga untuk kolom **c**.

Perhatikan juga bahwa dalam query diatas tidak ditulis perintah **INTO**. Hal ini tetap valid dan tidak error. Hanya saja, kebanyakan penulisan query **INSERT** tetap menyertakan perintah **INTO** seperti berikut:

```
INSERT INTO contoh_insert_1 VALUES (11, 'biru', 20190101);
Query OK, 1 row affected (0.06 sec)
```

```
SELECT * FROM contoh_insert_1;
+-----+-----+-----+
| a    | b    | c    |
+-----+-----+-----+
| 10  | merah | 2018-01-01 |
| 11  | biru  | 2019-01-01 |
+-----+-----+-----+
```

Ketika proses input berhasil, MySQL akan menginformasikan berapa banyak baris (**row**) yang ditambah. Keterangan *Query OK, 1 row affected (0.06 sec)* artinya query sukses diproses dengan waktu 0.06 detik dan terdapat 1 baris yang diproses (*1 row affected*).

Query **INSERT . . . VALUES** juga bisa digunakan untuk menginput banyak data sekaligus, seperti contoh berikut:

```
INSERT INTO contoh_insert_1 VALUES
(20, 'putih', 19900817),
(300, 'hitam', 20001201),
(-17, 'kuning', 20210101)
;
Query OK, 3 rows affected (0.05 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM contoh_insert_1;
```

```
+-----+-----+-----+
| a    | b     | c      |
+-----+-----+-----+
| 10   | merah | 2018-01-01 |
| 11   | biru   | 2019-01-01 |
| 20   | putih  | 1990-08-17 |
| 300  | hitam  | 2000-12-01 |
| -17  | kuning | 2021-01-01 |
+-----+-----+-----+
```

Saya memisahkan penulisan data menjadi beberapa baris. Ini semata-mata agar query tersebut mudah dibaca. Anda tetap bisa menulisnya dalam 1 baris panjang.

Jika ada banyak data yang diinput sekaligus, *feedback* atau info dari MySQL akan menambah 1 baris keterangan, seperti: *Records: 3 Duplicates: 0 Warnings: 0*. Artinya, ada 3 baris yang diproses (*Records: 3*), tidak ditemukan duplikasi data (*Duplicates: 0*) dan tidak ada warning (*Warnings: 0*).

Selanjutnya, bagaimana jika saya hanya ingin mengisi sebagai data saja (tidak untuk seluruh kolom)? Untuk keperluan seperti ini, harus ditulis nama kolom yang akan diberikan nilai sebelum perintah **VALUES**.

Sebagai contoh, saya ingin mengisi hanya kolom **a** saja. Penulisan querinya adalah sebagai berikut:

```
INSERT INTO contoh_insert_1 (a) VALUES (10);
```

Disini kolom **a** diisi dengan angka 10. Bagaimana dengan kolom **b** dan **c**? Karena nilainya tidak ditulis, MySQL akan menggunakan nilai default dari kolom tersebut. Secara bawaan, nilai **NULL** akan menjadi nilai default:

```
SELECT * FROM contoh_insert_1;
+-----+-----+-----+
| a    | b     | c      |
+-----+-----+-----+
| 10   | merah | 2018-01-01 |
| 11   | biru   | 2019-01-01 |
| 20   | putih  | 1990-08-17 |
| 300  | hitam  | 2000-12-01 |
| -17  | kuning | 2021-01-01 |
| 10   | NULL  | NULL  |
+-----+-----+-----+
```

Jika saya hanya ingin menginput data untuk kolom **b** dan **c**, caranya juga sama, yakni menulis nama kolom (**b**, **c**) sebelum perintah **VALUES**:

```
INSERT INTO contoh_insert_1 (b, c) VALUES ('cokelat', 20180101);
```

```
SELECT * FROM contoh_insert_1;
+-----+-----+-----+
| a    | b     | c      |
+-----+-----+-----+
| 10   | merah | 2018-01-01 |
| 11   | biru   | 2019-01-01 |
| 20   | putih  | 1990-08-17 |
| 300  | hitam  | 2000-12-01 |
| -17  | kuning | 2021-01-01 |
| 10   | NULL   | NULL    |
| NULL | cokelat | 2018-01-01 |
+-----+-----+-----+
```

Jika kita tidak ingin nilai NULL tampil sebagai nilai default, ini bisa diubah dengan menambahkan atribut **DEFAULT** ketika pembuatan tabel. Atau bisa juga menggunakan perintah **ALTER** untuk memodifikasi atribut **DEFAULT** dari kolom tersebut.

Sebagai contoh, saya akan membuat tabel **contoh_insert_2** yang sudah menyertakan nilai default:

```
CREATE TABLE contoh_insert_2 (
  a INT DEFAULT 100,
  b VARCHAR(10) DEFAULT 'hijau',
  c DATE DEFAULT 20171231
);
```

```
DESC contoh_insert_2;
+-----+-----+-----+-----+-----+
| Field | Type       | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| a     | int(11)    | YES  |     | 100     |        |
| b     | varchar(10) | YES  |     | hijau   |        |
| c     | date        | YES  |     | 2017-12-31 |        |
+-----+-----+-----+-----+-----+
```

Setiap kolom dari tabel **contoh_insert_2** diberikan nilai default. Ketika salah satu kolom tidak memiliki data, nilai default inilah yang akan dipakai:

```
INSERT INTO contoh_insert_2 (a) VALUES (10);

SELECT * FROM contoh_insert_2;
+---+---+---+
| a | b | c |
+---+---+---+
| 10 | hijau | 2017-12-31 |
+---+---+---+

INSERT INTO contoh_insert_2 (b, c) VALUES ('cokelat',20180101);

SELECT * FROM contoh_insert_2;
+---+---+---+
| a | b | c |
+---+---+---+
| 10 | hijau | 2017-12-31 |
| 100 | cokelat | 2018-01-01 |
+---+---+---+
```

Lebih lanjut tentang nilai default dan cara penggunaan atribut `DEFAULT` sudah pernah kita bahas di bab tentang **atribut tipe data**.

MySQL juga menyediakan keyword khusus: `DEFAULT` untuk menandakan bahwa kolom tersebut akan menggunakan nilai default. Berikut contoh penggunaannya:

```
INSERT INTO contoh_insert_2 VALUES (DEFAULT,'jingga',20190201);

SELECT * FROM contoh_insert_2;
+---+---+---+
| a | b | c |
+---+---+---+
| 10 | hijau | 2017-12-31 |
| 100 | cokelat | 2018-01-01 |
| 100 | jingga | 2019-02-01 |
+---+---+---+
```

Perhatikan penulisan nilai `DEFAULT` dalam query `INSERT`. Artinya, saya menginstruksikan kepada MySQL untuk menggunakan nilai default pada kolom `a`.

Juga harap diperhatikan agar tidak keliru bahwa keyword `DEFAULT` ini tidak sama efeknya dengan nilai `NULL` atau `0`:

```
INSERT INTO contoh_insert_2 VALUES (NULL, 'jingga', 20190201);
INSERT INTO contoh_insert_2 VALUES (0, 'jingga', 20190201);
```

```
SELECT * FROM contoh_insert_2;
+-----+-----+-----+
| a    | b      | c      |
+-----+-----+-----+
| 10   | hijau  | 2017-12-31 |
| 100  | cokelat | 2018-01-01 |
| 100  | jingga  | 2019-02-01 |
| NULL | jingga  | 2019-02-01 |
| 0    | jingga  | 2019-02-01 |
+-----+-----+-----+
```

Nilai **NULL** dan **0** baru akan berganti menjadi nilai lain ketika kolom tersebut di set sebagai **AUTO_INCREMENT**.

12.2 INSERT ... SET

Bentuk kedua dari query **INSERT** adalah **INSERT ... SET**. Disini, nilai untuk setiap kolom ditulis satu per satu dengan format sebagai berikut:

```
INSERT [INTO] nama_tabel SET nama_kolom1 = nilai1, nama_kolom2 = nilai2, ...
```

Sebagai contoh praktek, saya akan menggunakan kembali tabel **contoh_insert_2**, namun saya akan mengosongkan isi tabel menggunakan query **TRUNCATE**:

```
SELECT * FROM contoh_insert_2;
+-----+-----+-----+
| a    | b      | c      |
+-----+-----+-----+
| 10   | hijau  | 2017-12-31 |
| 100  | cokelat | 2018-01-01 |
| 100  | jingga  | 2019-02-01 |
| NULL | jingga  | 2019-02-01 |
| 0    | jingga  | 2019-02-01 |
+-----+-----+-----+
```

```
TRUNCATE contoh_insert_2;
Query OK, 0 rows affected (0.30 sec)
```

```
SELECT * FROM contoh_insert_2;
Empty set (0.00 sec)
```

Query TRUNCATE bertujuan untuk menghapus seluruh data yang ada di dalam tabel. Secara internal, query ini akan menghapus dan membuat ulang tabel. Cara ini jauh lebih cepat daripada menghapus satu-satu data yang ada.

Baik, berikut cara penggunaan query INSERT . . . SET:

```
INSERT INTO contoh_insert_2 SET a = 20, b = 'merah', c = 20180101;
```

```
SELECT * FROM contoh_insert_2;
+-----+-----+-----+
| a    | b    | c      |
+-----+-----+-----+
| 20   | merah | 2018-01-01 |
+-----+-----+-----+
```

Untuk query INSERT . . . SET, penulisan nama kolom harus sesuai dengan nilai yang ingin diinput. Kita juga tidak harus mengisi semua kolom, sebagaimana contoh berikut:

```
INSERT INTO contoh_insert_2 SET a = 50, b = 'cokelat';
```

```
SELECT * FROM contoh_insert_2;
+-----+-----+-----+
| a    | b    | c      |
+-----+-----+-----+
| 20   | merah | 2018-01-01 |
| 50   | cokelat | 2017-12-31 |
+-----+-----+-----+
```

Jika terdapat kolom yang tidak diinput, akan dipakai nilai default sebagaimana dalam query INSERT . . . VALUES.

Nilai DEFAULT juga bisa dipakai untuk query INSERT . . . SET, seperti contoh berikut:

```
INSERT INTO contoh_insert_2 SET a = 300, b = DEFAULT;
```

```
SELECT * FROM contoh_insert_2;
+-----+-----+-----+
| a    | b    | c      |
+-----+-----+-----+
| 20   | merah | 2018-01-01 |
| 50   | cokelat | 2017-12-31 |
| 300  | hijau  | 2017-12-31 |
+-----+-----+-----+
```

Dalam perintah query diatas, kolom b akan menggunakan nilai default, yakni 'hijau'. Begitu juga halnya dengan kolom c.

12.3 INSERT ... SELECT

Bentuk ketiga dari query `INSERT` adalah `INSERT ... SELECT`. Query ini merupakan gabungan dua buah perintah: `INSERT` dan `SELECT`. Dengan menggunakan query `INSERT ... SELECT`, kita bisa mengisi data tabel yang nilainya diambil dari tabel lain.

Penggabungan query juga dikenal dengan istilah **subquery**. Karena disini kita menggabungkan 2 buah query, perintahnya bisa jadi cukup kompleks.

Kunci dari penggunaan query `INSERT ... SELECT` terletak di bagian `SELECT`-nya, yakni bagaimana cara mencari data dari tabel lain. Materi tentang query `SELECT` baru akan kita bahas dalam bab berikutnya, oleh karena itu kita akan fokus ke query `INSERT` saja.

Sebagai contoh praktik, saya akan membuat tabel `contoh_insert_3` yang strukturnya dicopy dari tabel `contoh_insert_2`:

```
CREATE TABLE contoh_insert_3 LIKE contoh_insert_2;
```

```
SELECT * FROM contoh_insert_3;
Empty set (0.00 sec)
```

Sekarang, kita akan pakai query `INSERT ... SELECT` untuk mengambil seluruh data dari tabel `contoh_insert_2` ke tabel `contoh_insert_3`:

```
INSERT INTO contoh_insert_3 SELECT * FROM contoh_insert_2;
```

```
SELECT * FROM contoh_insert_3;
+-----+-----+-----+
| a    | b    | c    |
+-----+-----+-----+
| 20   | merah | 2018-01-01 |
| 50   | cokelat | 2017-12-31 |
| 300  | hijau  | 2017-12-31 |
+-----+-----+-----+
```

Perintah diatas sudah pernah saya pakai di materi tentang cara men-cloning data tabel.

Isi tabel yang diambil juga tidak harus semua, tapi bisa di filter tergantung struktur tabel dan perintah `SELECT` yang dipakai.

Sebagai contoh, saya ingin menginput isi kolom `b` dan `c` dari tabel `contoh_insert_2` ke dalam kolom `b` dan `c` dari tabel `contoh_insert_3`:

```
TRUNCATE contoh_insert_3;

INSERT INTO contoh_insert_3 (b,c) SELECT b,c FROM contoh_insert_2;

SELECT * FROM contoh_insert_3;
+-----+-----+-----+
| a    | b      | c      |
+-----+-----+-----+
| 100  | merah  | 2018-01-01 |
| 100  | cokelat | 2017-12-31 |
| 100  | hijau   | 2017-12-31 |
+-----+-----+-----+

DESC contoh_insert_3;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| a     | int(11)   | YES  |     | 100     |       |
| b     | varchar(10)| YES  |     | hijau   |       |
| c     | date       | YES  |     | 2017-12-31|       |
+-----+-----+-----+-----+-----+
```

Pertanyaannya, bagaimana dengan isi kolom a? Karena tidak ada data yang diinput, akan dipakai nilai default, yakni 100. Ini bisa dilihat dari hasil query DESC.

Fitur lain dari query **INSERT ... SELECT** adalah nama kolom dan tipe data dari tabel asal tidak harus sama dengan tabel baru. Selama data tersebut valid dan bisa diinput ke tipe data baru, hal tersebut bisa dilakukan. Berikut contoh prakteknya:

```
CREATE TABLE contoh_insert_4 (
  x TIMESTAMP,
  y SMALLINT,
  z CHAR(10)
);

INSERT INTO contoh_insert_4 (x,y,z) SELECT c,a,b FROM contoh_insert_2;

SELECT * FROM contoh_insert_4;
+-----+-----+-----+
| x           | y   | z      |
+-----+-----+-----+
| 2018-01-01 00:00:00 | 20 | merah  |
| 2017-12-31 00:00:00 | 50 | cokelat |
| 2017-12-31 00:00:00 | 300 | hijau  |
+-----+-----+-----+
```

Saya membuat tabel contoh_insert_4 dengan 3 kolom: x dengan tipe data TIMESTAMP, y dengan tipe data SMALLINT, dan z dengan tipe data CHAR(10).

Selanjutnya saya menggunakan query INSERT ... SELECT untuk menginput tabel contoh_insert_4 dari data yang tersedia dalam tabel contoh_insert_2 .

Perhatikan bahwa secara struktur, seluruh kolom dalam kedua tabel ini tidak ada yang sama. Namun datanya masih dalam satu jenis:

- Kolom x = dari DATE ke TIMESTAMP.
- Kolom y = dari INT ke SMALLINT.
- Kolom z = dari VARCHAR(10) ke CHAR(10).

Urutan nama kolom juga harus diubah karena struktur tabel contoh_insert_4 tidak sama dengan contoh_insert_2.

Apabila saat proses input terdapat data yang tidak valid, akan menghasilkan error seperti contoh berikut:

```
CREATE TABLE contoh_insert_5 (
    x TIMESTAMP,
    y TINYINT,
    z CHAR(10)
);

INSERT INTO contoh_insert_5 (x,y,z) SELECT c,a,b FROM contoh_insert_2;
ERROR 1264 (22003): Out of range value for column 'y' at row 3
```

Perhatikan tipe data kolom y dari tabel contoh_insert_5. Kolom ini bertipe TINYINT, sedangkan isi data yang diambil dari kolom a tabel contoh_insert_2 ada yang bernilai 300. Angka 300 ini berada di luar jangkauan tipe data TINYINT.

Error diatas sebenarnya terjadi karena saya menggunakan MySQL yang berjalan di **strict mode**. Jika strict mode dihapus, query diatas tidak akan berjalan dan tidak menghasilkan error, (hanya warning). Dimana angka 300 akan di konversi menjadi 127 sebagai nilai maksimum dari tipe data TINYINT.

12.4 ON DUPLICATE KEY UPDATE

ON DUPLICATE KEY UPDATE adalah perintah tambahan untuk query INSERT. Perintah ini dipakai saat menangani error ketika kita mencoba menginput data yang sama ke kolom yang tidak memperbolehkan hal tersebut (seperti kolom dengan atribut PRIMARY KEY).

Seperti yang pernah dibahas, jika sebuah kolom memiliki atribut PRIMARY KEY atau atribut UNIQUE, data di dalam kolom tersebut tidak boleh ada yang sama. Jika tetap diinput, akan menghasilkan error. Berikut contoh kasusnya:

```

CREATE TABLE contoh_insert_6 (
    a INT PRIMARY KEY,
    b VARCHAR(10),
    c DATE
);

INSERT contoh_insert_6 VALUES
(2, 'merah', 20180101),
(11, 'biru', 20190101),
(20, 'putih', 19900817)
;

INSERT contoh_insert_6 VALUES (2, 'jingga', 20190201);
ERROR 1062 (23000): Duplicate entry '2' for key 'PRIMARY'

```

Saya membuat tabel contoh_insert_6 dimana kolom a di set sebagai INT PRIMARY KEY. Artinya, kolom a akan menjadi kolom utama dan tidak boleh terdapat angka yang berulang.

Ketika saya mencoba menginput angka 2 ke dalam kolom a, hasilnya: *ERROR 1062 (23000): Duplicate entry '2' for key 'PRIMARY'*. Hal ini terjadi karena di dalam kolom a sudah terdapat angka 2.

Perintah ON DUPLICATE KEY UPDATE bisa dipakai dalam situasi seperti ini, yakni ketika terdapat error yang menyatakan terdapat duplikasi data, kita bisa menjalankan sesuatu.

Sebagai contoh, dalam query berikut ini saya mencoba menginput kembali data (2, 'jingga', 20190201) ke tabel contoh_insert_6. Jika terdeteksi ada data yang berulang (*duplicate*), saya ingin agar di baris tersebut kolom b diubah menjadi 'jingga'. Berikut perintahnya:

```

INSERT contoh_insert_6 VALUES (2, 'jingga', 20190201)
ON DUPLICATE KEY UPDATE b = 'jingga';
Query OK, 2 rows affected (0.04 sec)

```

```

SELECT * FROM contoh_insert_6;
+---+-----+-----+
| a | b      | c          |
+---+-----+-----+
| 2 | jingga | 2018-01-01 |
| 11 | biru   | 2019-01-01 |
| 20 | putih  | 1990-08-17 |
+---+-----+-----+

```

Query INSERT diatas tidak akan menambah data baru karena angka 2 sudah ada di dalam kolom a. Namun yang akan diproses adalah bagian ON DUPLICATE KEY UPDATE b = 'jingga'.

Perintah INSERT... ON DUPLICATE KEY UPDATE sebenarnya perpaduan 2 query, yakni INSERT dan UPDATE. Jika INSERT gagal (karena ditemukan duplikasi data), jalankan bagian UPDATE.

Perintah untuk proses UPDATE juga bisa sedikit kompleks, seperti contoh berikut:

```
INSERT contoh_insert_6 VALUES (11, 'kuning', 20190201)
ON DUPLICATE KEY UPDATE a = 12, b = 'kuning', c = 20190201;
```

```
SELECT * FROM contoh_insert_6;
+---+---+---+
| a | b      | c          |
+---+---+---+
| 2 | jingga | 2018-01-01 |
| 12 | kuning | 2019-02-01 |
| 20 | putih  | 1990-08-17 |
+---+---+---+
```

Query diatas bisa dibaca:

“Input data (11, 'kuning', 20190201) ke dalam tabel contoh_insert_6. Jika terjadi duplikasi data, update data di baris tersebut dengan nilai a = 12, b = 'kuning' dan c = 20190201”.

Karena di dalam tabel contoh_insert_6 sudah ada nilai 11 di kolom a, bagian UPDATE-lah yang akan diproses.

12.5 Latihan Menginput Data Tabel

Untuk menguji pemahaman tentang query INSERT, dan juga berbagai materi yang sudah kita pelajari dari bab sebelumnya, saya sudah menyiapkan sebuah latihan. Latihan ini tidak hanya untuk menginput data, tapi juga membuat tabel, mengubah tabel, hingga mengcopy tabel.

Seluruh query yang diperlukan sudah kita dipelajari. Jika anda ragu, bisa membuka kembali bab-bab sebelumnya.

Baik, berikut tabel yang akan menjadi bahan latihan kita:

Nama Kota	Kecamatan	Kelurahan	Luas Wilayah (km ²)	Jumlah Penduduk
Jakarta	44	267	664,01	9.988.495
Surabaya	31	154	350,54	2.805.906
Medan	21	151	265,00	2.465.469
Bekasi	12	56	206,61	2.381.053
Bandung	30	151	167,67	2.339.463
Makassar	14	143	199,26	1.651.146
Depok	11	63	200,29	1.631.951
Semarang	16	177	373,78	1.621.384
Tangerang	13	104	153,93	1.566.190
Palembang	14	107	369,22	1.548.064

Tabel diatas berisi 10 besar kota dengan jumlah penduduk terbesar di Indonesia. Data ini saya dapat dari situs [kemendagri.go.id](http://www.kemendagri.go.id)¹.

¹http://www.kemendagri.go.id/media/documents/2015/02/25/l/a/lampiran_i.pdf

Berikut soal latihan:

1. Buatlah struktur tabel yang akan menampung data-data diatas (CREATE TABLE). Kali ini saya tidak akan membatasi nama tabel, nama kolom maupun tipe datanya. Silahkan anda ‘berkreasi’ sendiri.

Misalnya, tidak masalah memakai tipe data TINYINT atau BIGINT untuk kolom jumlah penduduk. Tapi saya yakin anda sudah paham apa beda dari TINYINT dan BIGINT, dan tipe data mana yang sebaiknya dipakai. Boleh juga menambahkan beberapa atribut jika dirasa perlu.

Instruksi pertama ini hanya untuk membuat tabel saja. Proses input data akan menjadi soal no. 2.

2. Input 8 kota pertama menggunakan query INSERT . . . VALUES. Anda bisa menginputnya satu per satu (satu query INSERT untuk setiap baris), atau sekaligus dalam satu query INSERT.

Untuk penulisan query yang panjang dan berulang seperti ini, akan lebih mudah jika perintah tersebut diketik dulu di dalam teks editor seperti **Notepad++**, kemudian baru di copy ke **cmd Windows**.

Harap teliti dengan nilai yang diinput. Dalam tabel diatas, kolom luas wilayah menggunakan tanda koma sebagai pemisah desimal. Di dalam MySQL, kita menggunakan tanda titik sebagai pemisah desimal. Begitu juga data untuk kolom jumlah penduduk yang memiliki tanda titik sebagai pemisah ribuan.

Nama kolom yang mengandung spasi juga bisa menjadi masalah. Solusinya, anda bisa menggunakan karakter *backtick* atau menggunakan nama kolom yang lain.

3. Input 2 kota terakhir menggunakan query INSERT . . . SET.
4. Buat tabel kedua dengan menggunakan struktur yang sama dengan tabel pertama. Artinya, akan ada 2 tabel: Tabel pertama yang sudah berisi data (yang kita buat berdasarkan soal 1 - 3), dan tabel kedua yang belum berisi data.

Struktur tabel pertama dan kedua ini sama persis, dimana sama-sama terdiri dari 5 kolom.

5. Ubah tabel kedua, hapus kolom Kecamatan, Kelurahan dan Luas Wilayah. Sehingga tabel kedua hanya berisi 2 kolom saja: Nama Kota dan Jumlah Penduduk.
6. Input tabel kedua dengan data yang diambil dari tabel pertama. Disini anda akan menggunakan query INSERT . . . SELECT. Perhatikan bahwa jumlah kolom di tabel kedua hanya tinggal 2 buah: Nama Kota dan Jumlah Penduduk. Struktur kolom ini sudah tidak sama dengan jumlah kolom dari tabel pertama.

Hasil akhirnya dari query INSERT . . . SELECT, tabel kedua akan berisi 10 nama kota beserta jumlah penduduk, seperti tampilan berikut:

kota	penduduk
Jakarta	9988495
Surabaya	2805906
Medan	2465469
Bekasi	2381053
Bandung	2339463
Makassar	1651146
Depok	1631951
Semarang	1621384
Tangerang	1566190
Palembang	1548064

Silahkan anda coba kerjakan latihan diatas secara berurutan. Tidak ada aturan harus menggunakan tipe data apa atau memberi nama kolom apa. Selama data bisa diinput semua, hal tersebut tidak menjadi masalah.

12.6 Jawaban Menginput Data Tabel

Untuk menyamakan jawaban, atau mungkin anda menyerah dengan latihan diatas (semoga tidak), berikut query yang saya pakai.

1. Saya memutuskan memberi nama tabel populasi, yang dibuat dengan query sebagai berikut:

```
CREATE TABLE populasi (
    kota VARCHAR(50),
    kec TINYINT UNSIGNED,
    kel SMALLINT UNSIGNED,
    luas DECIMAL(6,2),
    penduduk INTEGER UNSIGNED
);
```

Pilihan tipe data diatas tidak harus sama persis. Jika anda menggunakan VARCHAR(15) untuk kolom kota juga tidak masalah, karena nama kota di dalam tabel maksimal memang hanya terdiri dari 12 karakter. Hanya saja jika tabel ini nantinya menampung nama kota lain, memberikan nilai yang terlalu kecil berpotensi jadi masalah.

Saya menggunakan atribut UNSIGNED untuk setiap tipe data INTEGER agar jangkauannya menjadi lebih besar. Inipun sebenarnya tidak harus.

2. Berikut query untuk menginput 8 data pertama menggunakan INSERT . . . VALUES:

```
INSERT INTO populasi VALUES
('Jakarta', 44, 267, 664.01, 9988495),
('Surabaya', 31, 154, 350.54, 2805906),
('Medan', 21, 151, 265.00, 2465469),
('Bekasi', 12, 56, 206.61, 2381053),
('Bandung', 30, 151, 167.67, 2339463),
('Makassar', 14, 143, 199.26, 1651146),
('Depok', 11, 63, 200.29, 1631951),
('Semarang', 16, 177, 373.78, 1621384)
;
```

Saya menggunakan 1 perintah query untuk menginput banyak data sekaligus. Untuk mengetik query seperti ini akan lebih praktis membuatnya di dalam text editor seperti **Notepad++**, kemudian baru di paste ke **cmd Windows**.

Pastikan hasilnya sudah sesuai dengan menjalankan query **SELECT * FROM populasi**.

3. Berikut query untuk menginput 2 data terakhir menggunakan **INSERT ... SET**:

```
INSERT INTO populasi SET
kota = 'Tangerang', kec = 13, kel = 104, luas = 153.93, penduduk = 1566190;
```

```
INSERT INTO populasi SET
kota = 'Palembang', kec = 14, kel = 107, luas = 369.22, penduduk = 1548064;
```

Karena query **INSERT ... SET** tidak mendukung banyak data sekaligus, saya harus menginputnya satu persatu.

4. Untuk membuat tabel kedua dengan struktur yang sama seperti tabel pertama, akan lebih praktis menggunakan query **CREATE ... LIKE**:

```
CREATE TABLE populasi_saja LIKE populasi;
```

Sekarang, tabel **populasi_saja** sudah memiliki struktur yang sama seperti tabel **populasi**. Alternatif lain tentu saja membuat ulang tabel **populasi_saja** secara manual menggunakan query **CREATE TABLE**.

5. Proses penghapusan kolom akan ditangani oleh query **ALTER ... DROP**:

```
ALTER TABLE populasi_saja DROP kec, DROP kel, DROP luas;
```

Untuk memastikan struktur tabel, bisa di cek menggunakan query **DESC populasi_saja**.

6. Proses input data dari tabel populasi bisa dilakukan menggunakan query berikut:

```
INSERT INTO populasi_saja SELECT kota, penduduk FROM populasi;
```

Karena tabel **populasi_saja** hanya memiliki 2 kolom, kita harus menuliskan kolom apa saja yang akan diambil dari tabel **populasi**. Kolom tersebut adalah **kota** dan **penduduk**.

Selamat, jika anda berhasil mengerjakan seluruh latihan dengan sukses. Disini anda juga belajar menebak apa tipe data yang sesuai. Karena tentu saja dalam kerjaan sehari-hari kita juga hanya berhadapan dengan data mentah seperti tabel diatas.

Kitalah sebagai programmer yang akan menentukan “Oh, untuk kolom ini akan lebih pas menggunakan tipe data `BIGINT`, sedangkan kolom ini lebih bagus menggunakan tipe data `VARCHAR(100)`”. Seiring dengan banyaknya latihan, anda akan lebih mahir dalam proses analisis seperti ini.

Berikutnya, kita akan masuk ke materi yang mungkin paling ditunggu-tunggu, yakni menampilkan data tabel menggunakan query `SELECT`.

13. Menampilkan Data Tabel

Menampilkan data tabel bisa dianggap sebagai salah satu materi paling penting di dalam buku ini dan (mungkin) sekaligus yang paling rumit.

Query yang kita pakai untuk menampilkan data tabel hanya 1, yakni SELECT. Akan tetapi perintah tambahannya sangat beragam, mulai dari AS, ORDER BY, LIMIT, WHERE, GROUP BY, HAVING, LIKE, UNION hingga JOIN.

Banyaknya perintah tambahan ini karena proses menampilkan tabel butuh penyeleksian data yang detail. Sebagai contoh, berdasarkan tabel populasi yang kita buat dari bab sebelumnya, tampilkan 3 provinsi teratas dengan jumlah kecamatan paling banyak dan syaratnya luas wilayah tidak lebih dari 300 km².

Query yang dibutuhkan adalah sebagai berikut:

```
SELECT * FROM populasi WHERE luas < 300 ORDER BY kec DESC LIMIT 3;
+-----+-----+-----+-----+
| kota | kec | kel | luas | penduduk |
+-----+-----+-----+-----+
| Bandung | 30 | 151 | 167.67 | 2339463 |
| Medan | 21 | 151 | 265.00 | 2465469 |
| Makassar | 14 | 143 | 199.26 | 1651146 |
+-----+-----+-----+-----+
```

Selain perintah seperti ini, kita bisa melakukan banyak hal lain untuk mendapatkan informasi sesuai keinginan. Kuncinya, ada di penguasaan query SELECT. Inilah yang akan kita pelajari dalam bab ini.



Kecuali dinyatakan lain, seluruh contoh tabel di input ke dalam database belajar. Saya berasumsi anda sudah menjalankan query USE DATABASE belajar agar bisa mengikuti materi dalam bab ini.

13.1 SELECT ... select_expression

Bentuk yang paling sederhana dari query SELECT adalah SELECT...select_expression. select_expression berisi perintah dasar SQL yang bisa menghasilkan sebuah nilai, dan itu tanpa perlu menuliskan nama tabel apapun.

Berikut beberapa contoh penggunaan dari query SELECT...select_expression:

```
SELECT 2;
+---+
| 2 |
+---+
| 2 |
+---+  
  
SELECT 10 + 5;
+-----+
| 10 + 5 |
+-----+
|      15 |
+-----+  
  
SELECT 'DuniaIlkom';
+-----+
| DuniaIlkom |
+-----+
| DuniaIlkom |
+-----+
```

Salah satu bentuk dari `select_expression` ini juga bisa berupa fungsi bawaan MySQL:

```
SELECT SQRT(49);
+-----+
| SQRT(49) |
+-----+
|      7 |
+-----+  
  
SELECT UPPER('DuniaIlkom');
+-----+
| UPPER('DuniaIlkom') |
+-----+
| DUNIAILKOM          |
+-----+
```

Disini fungsi `SQRT()` digunakan untuk mencari akar kuadrat dari sebuah angka. Sedangkan fungsi `UPPER()` dipakai untuk mengkonversi string menjadi huruf besar. Lebih lanjut tentang fungsi-fungsi bawaan MySQL akan saya bahas dalam bab tersendiri.

Penulisan `select_expression` juga bisa digabung dalam satu perintah `SELECT`, dimana setiap `expression` dipisah dengan tanda koma:

```
SELECT LOWER('DuNiAILkom'), POW(3,2), 'Belajar MySQL';
+-----+-----+-----+
| LOWER('DuNiAILkom') | POW(3,2) | Belajar MySQL |
+-----+-----+-----+
| duniailkom          |         9 | Belajar MySQL |
+-----+-----+-----+
```

Pada query diatas, saya menggunakan 2 buah fungsi bawaan MySQL: Fungsi LOWER() untuk mengkonversi string menjadi huruf kecil dan fungsi POW() untuk mencari pangkat dari sebuah angka, dalam contoh ini $\text{POW}(3,2)$ berarti 3^2 .

Dalam standar bahasa SQL, perintah SELECT sebenarnya tidak boleh ditulis seperti ini. Untuk setiap query SELECT, harus ada setidaknya 1 tabel. Agar bisa memenuhi syarat tersebut, MySQL menyediakan tabel khusus: DUAL.

Berikut contoh penggunaannya:

```
SELECT 'DuniaIlkom' FROM DUAL;
+-----+
| DuniaIlkom |
+-----+
| DuniaIlkom |
+-----+
```

Tabel DUAL tidak memiliki makna apapun, hanya sebagai tabel *dummy* supaya perintah SELECT MySQL bisa ditulis lengkap dengan nama tabel.

Membuat tabel populasi

Sebagai persiapan untuk materi selanjutnya, saya akan memakai tabel populasi yang pernah kita buat dalam latihan bab Menginput Data Tabel. Silahkan anda periksa apakah tabel sudah terbentuk atau gunakan query berikut untuk membuat tabel populasi:

```
CREATE TABLE IF NOT EXISTS populasi (
    kota VARCHAR(50),
    kec TINYINT UNSIGNED,
    kel SMALLINT UNSIGNED,
    luas DECIMAL(6,2),
    penduduk INTEGER UNSIGNED
);

INSERT INTO populasi VALUES
    ('Jakarta', 44, 267, 664.01, 9988495),
    ('Surabaya', 31, 154, 350.54, 2805906),
    ('Medan', 21, 151, 265.00, 2465469),
    ('Bekasi', 12, 56, 206.61, 2381053),
```

```

('Bandung', 30, 151, 167.67, 2339463),
('Makassar', 14, 143, 199.26, 1651146),
('Depok', 11, 63, 200.29, 1631951),
('Semarang', 16, 177, 373.78, 1621384),
('Tangerang', 13, 104, 153.93, 1566190),
('Palembang', 14, 107, 369.22, 1548064)
;

SELECT * FROM populasi;
+-----+-----+-----+-----+
| kota | kec | kel | luas | penduduk |
+-----+-----+-----+-----+
| Jakarta | 44 | 267 | 664.01 | 9988495 |
| Surabaya | 31 | 154 | 350.54 | 2805906 |
| Medan | 21 | 151 | 265.00 | 2465469 |
| Bekasi | 12 | 56 | 206.61 | 2381053 |
| Bandung | 30 | 151 | 167.67 | 2339463 |
| Makassar | 14 | 143 | 199.26 | 1651146 |
| Depok | 11 | 63 | 200.29 | 1631951 |
| Semarang | 16 | 177 | 373.78 | 1621384 |
| Tangerang | 13 | 104 | 153.93 | 1566190 |
| Palembang | 14 | 107 | 369.22 | 1548064 |
+-----+-----+-----+-----+

```

Jika tabel populasi sudah berhasil dibuat dan datanya sesuai dengan tampilan diatas, mari lanjut membahas bentuk lainnya dari query SELECT.

13.2 SELECT ... FROM

Bentuk SELECT...FROM merupakan bentuk perintah SELECT yang paling umum. Format dasar penulisannya adalah sebagai berikut:

```
SELECT nama_kolom FROM nama_tabel;
```

Sebagai contoh, saya ingin menampilkan isi kolom kota dari tabel populasi:

```
SELECT kota FROM populasi;
```

kota
Jakarta
Surabaya
Medan
Bekasi
....



Karena isi tabel populasi cukup banyak (10 baris), saya mengganti beberapa baris terakhir dengan tanda titik tiga (...), ini semata-mata agar tampilannya tidak terlalu panjang. Jika anda menjalankan query tersebut, seharusnya akan tampil 10 baris. Penulisan seperti ini akan saya pakai untuk tabel-tabel lainnya.

Untuk menampilkan kolom lain, cukup ditulis nama kolom yang ingin ditampilkan dan dipisah dengan tanda koma:

```
SELECT penduduk, kel, kec FROM populasi;
```

penduduk	kel	kec
9988495	267	44
2805906	154	31
2465469	151	21
...

Nama kolom yang dipilih juga tidak harus satu, tapi bisa berulang tergantung keinginan, seperti contoh berikut:

```
SELECT kota, kec, kota, penduduk, kota FROM populasi;
```

kota	kec	kota	penduduk	kota
Jakarta	44	Jakarta	9988495	Jakarta
Surabaya	31	Surabaya	2805906	Surabaya
Medan	21	Medan	2465469	Medan
...

Dalam query diatas, saya mengulang kolom kota beberapa kali.

Jika kita ingin menampilkan seluruh kolom yang ada di dalam tabel, bisa menulis satu persatu kolom tersebut, atau bisa diganti menggunakan karakter khusus berupa tanda bintang (*) yang maksudnya: *ambil seluruh kolom yang ada*. Seperti contoh berikut:

```
SELECT * FROM populasi;
+-----+-----+-----+-----+
| kota | kec | kel | luas | penduduk |
+-----+-----+-----+-----+
| Jakarta | 44 | 267 | 664.01 | 9988495 |
| Surabaya | 31 | 154 | 350.54 | 2805906 |
| Medan | 21 | 151 | 265.00 | 2465469 |
| ... | ... | ... | ... | ... |
+-----+-----+-----+-----+
```

Penulisan query **SELECT** seperti ini sudah sering kita pakai dalam bab-bab sebelumnya.

13.3 SELECT ... AS

Perintah **AS** sebenarnya tidak hanya dipakai pada query **SELECT**, tapi juga berbagai query lain di dalam MySQL. Hanya saja penggunaannya memang lebih banyak digabung dengan query **SELECT**.

Query **SELECT...AS** berfungsi untuk membuat **alias** atau nama pengganti dari sebuah kolom. Berikut contoh penggunaannya:

```
SELECT kota AS nama_kota FROM populasi;
+-----+
| nama_kota |
+-----+
| Jakarta |
| Surabaya |
| Medan |
| ... |
+-----+
```

Bagian perintah **kota AS nama_kota** artinya, saya ingin membuat *alias* dari kolom **kota** menjadi **nama_kota**. Dari tampilan query **SELECT**, dapat terlihat bahwa judul kolom bukan lagi **kota**, tapi berganti menjadi **nama_kota**.

Kita juga bisa membuat alias (AS) yang memiliki karakter khusus seperti spasi:

```
SELECT kota AS 'Nama Kota' FROM populasi;
+-----+
| Nama Kota |
+-----+
| Jakarta   |
| Surabaya  |
| Medan     |
| ...        |
+-----+
```

Dengan query seperti ini, kita bisa mengganti judul kolom agar lebih cantik. Yang juga bisa dilakukan untuk kolom-kolom lainnya:

```
SELECT
    kota AS 'Nama Kota',
    kec AS 'Jumlah Kecamatan',
    penduduk AS 'Jumlah Penduduk'
FROM populasi;
```

```
+-----+-----+-----+
| Nama Kota | Jumlah Kecamatan | Jumlah Penduduk |
+-----+-----+-----+
| Jakarta   |          44 |      9988495 |
| Surabaya  |          31 |      2805906 |
| Medan     |          21 |      2465469 |
| Bekasi    |          12 |      2381053 |
| ...       |          .. |       ...   |
+-----+-----+-----+
```

Dengan menggunakan query `SELECT ... AS`, kita tidak perlu khawatir memberi nama kolom yang agak aneh, seperti `kec` atau `ke1`. Sebab, ketika tabel tersebut ingin ditampilkan, nama kolom bisa diganti dengan mudah.

Penggunaan `SELECT ... AS` menjadi penting ketika kita menggabungkan beberapa kolom atau menggunakan fungsi bawaan MySQL. Judul kolom hasil proses ini terkesan kurang rapi, seperti contoh berikut:

```
SELECT UPPER(kota), kec + kel FROM populasi;
```

UPPER(kota)	kec + kel
JAKARTA	311
SURABAYA	185
MEDAN	172
BEKASI	68
BANDUNG	181
...	...

Untuk kolom pertama, saya menggunakan fungsi `UPPER(kota)` agar seluruh nama kota ditampilkan dengan huruf besar. Untuk kolom kedua, di dapat dari penjumlahan kolom `kec` dengan `kel`. Seperti yang terlihat, judul setiap kolom akan mencantumkan fungsi-fungsi ini.

Agar lebih rapi, kita bisa memanfaatkan query `SELECT...AS`:

```
SELECT
  UPPER(kota) AS 'Nama Kota',
  kec + kel AS 'Total Kecamatan dan Kelurahan'
FROM populasi;
```

Nama Kota	Total Kecamatan dan Kelurahan
JAKARTA	311
SURABAYA	185
MEDAN	172
BEKASI	68
...	...

Khusus untuk penggunaan fungsi bawaan MySQL dan cara penggabungan kolom seperti ini akan kita bahas dalam bagian khusus nantinya.

13.4 SELECT ... ORDER BY

Ketika kita menginput data ke dalam tabel, secara bawaan data yang diinput paling akhir akan berada di urutan paling bawah. Sebenarnya, urutan baris ini tidak bermakna apa-apa selain secara fisik data tabel memang disimpan seperti itu.

Dengan menggunakan berbagai perintah `SELECT`, kita bisa menampilkan data tabel sesuai kebutuhan dan tidak terikat dengan urutan fisik ini. Bisa saja baris paling akhir ditampilkan paling atas, tergantung query yang dijalankan. Salah satu query yang bisa dipakai adalah `SELECT...ORDER BY`.

Query SELECT...ORDER BY berfungsi untuk mengurutkan tampilan tabel berdasarkan kolom tertentu. Berikut format dasarnya:

```
SELECT nama_tabel FROM nama_tabel ORDER BY nama_kolom_urut [ASC | DESC]
```

Pada saat pembuatan tabel *populasi*, *kebetulan* saya menginput data secara berurutan mulai dari Jakarta dengan jumlah penduduk yang paling banyak, hingga kota Palembang dengan jumlah penduduk paling sedikit.

Saya sebut *kebetulan*, karena jika baris tabel diinput secara acak (tidak langsung tersusun dari jumlah penduduk yang paling besar), kita hanya perlu menambahkan query ORDER BY penduduk DESC ke dalam perintah SELECT. Dan MySQL akan menampilkan data tabel mulai dari populasi yang paling banyak hingga yang paling sedikit.

Bagaimana jika urutannya dibalik? Sekarang saya ingin menampilkan data mulai dari populasi yang paling sedikit hingga yang paling besar. Tidak masalah, berikut perintahnya:

```
SELECT * FROM populasi ORDER BY penduduk ASC;
```

kota	kec	kel	luas	penduduk
Palembang	14	107	369.22	1548064
Tangerang	13	104	153.93	1566190
Semarang	16	177	373.78	1621384
Depok	11	63	200.29	1631951
Makassar	14	143	199.26	1651146
Bandung	30	151	167.67	2339463
Bekasi	12	56	206.61	2381053
Medan	21	151	265.00	2465469
Surabaya	31	154	350.54	2805906
Jakarta	44	267	664.01	9988495

Perintah tambahan ORDER BY penduduk ASC menginstruksikan kepada MySQL agar tabel ditampilkan dengan diurutkan secara menaik (*ascending*) berdasarkan kolom penduduk. Sedangkan jika ditulis sebagai ORDER BY penduduk DESC artinya data akan ditampilkan dengan urutan menurun (*descending*) berdasarkan kolom penduduk.

Perintah tambahan ASC bersifat opsional. Jika query SELECT...ORDER BY dijalankan tanpa menulis ASC atau DESC, dianggap sebagai ASC.

Contoh lain, saya ingin menampilkan tabel *populasi* yang diurutkan berdasarkan kolom *kota*:

```
SELECT * FROM populasi ORDER BY kota;
+-----+-----+-----+-----+
| kota      | kec   | kel   | luas     | penduduk |
+-----+-----+-----+-----+
| Bandung   | 30    | 151   | 167.67   | 2339463  |
| Bekasi    | 12    | 56    | 206.61   | 2381053  |
| Depok     | 11    | 63    | 200.29   | 1631951  |
| Jakarta   | 44    | 267   | 664.01   | 9988495  |
| Makassar  | 14    | 143   | 199.26   | 1651146  |
| Medan     | 21    | 151   | 265.00   | 2465469  |
| Palembang | 14    | 107   | 369.22   | 1548064  |
| Semarang  | 16    | 177   | 373.78   | 1621384  |
| Surabaya  | 31    | 154   | 350.54   | 2805906  |
| Tangerang | 13    | 104   | 153.93   | 1566190  |
+-----+-----+-----+-----+
```

Kali ini tabel populasi ditampilkan secara berurutan berdasarkan kolom kota. Karena tidak ada instruksi ASC maupun DESC, proses pengurutan dianggap sebagai ASC. Hasilnya, pengurutan dilakukan secara menaik mulai dari huruf a hingga z. Jika kolom itu berisi angka, urutannya mulai dari yang paling kecil hingga yang paling besar.

Kolom yang digunakan sebagai patokan urutan juga tidak harus ditampilkan, seperti contoh berikut:

```
SELECT kota, luas FROM populasi ORDER BY kec DESC;
+-----+-----+
| kota      | luas     |
+-----+-----+
| Jakarta  | 664.01  |
| Surabaya | 350.54  |
| Bandung  | 167.67  |
| Medan    | 265.00  |
| ...       | ...      |
+-----+-----+
```

Disini saya hanya menampilkan kolom kota dan luas, namun urutannya berdasarkan kolom kec DESC. Artinya, tampilan diatas diurutkan mulai dari kota yang punya kecamatan paling banyak hingga paling kecil, meskipun kolom kecamatan sendiri tidak ditampilkan.

Proses pengurutan ini juga bisa dibuat berdasarkan kolom alias, yakni hasil dari query AS:

```
SELECT kota AS 'Nama Kota', penduduk FROM populasi ORDER BY `Nama Kota` DESC;
+-----+-----+
| Nama Kota | penduduk |
+-----+-----+
| Tangerang | 1566190 |
| Surabaya | 2805906 |
| Semarang | 1621384 |
| Palembang | 1548064 |
| ... | ... |
+-----+-----+
```

Perhatikan penulisan perintah ORDER BY, saya memakai tanda backtick ketika menulis Nama Kota. Ini karena nama alias tersebut mengandung karakter spasi (yang tidak memenuhi syarat *identifier*) sehingga tidak bisa ditulis langsung tanpa backtick.

Untuk proses pengurutan yang lebih detail, terutama jika di dalam tersebut punya banyak kolom dengan data yang sama, kita bisa menulis perintah ORDER BY beberapa kali. Jika ditulis seperti ini, proses pengurutan memiliki level atau tingkatan.

Sebagai tabel sample, saya akan membuat tabel contoh_select_1 dengan query sebagai berikut:

```
CREATE TABLE contoh_select_1 (
    a INT ,
    b VARCHAR(10),
    c TIME
);

INSERT contoh_select_1 VALUES
(1, 'merah', 085010), (1, 'biru', 123030), (2, 'putih', 181045),
(2, 'kuning', 003059), (1, 'jingga', 203030), (2, 'hitam', 194530),
(1, 'merah', 033510), (1, 'merah', 231530), (2, 'hitam', 101515)
;
```

```
SELECT * FROM contoh_select_1;
```

```
+-----+-----+-----+
| a     | b      | c      |
+-----+-----+-----+
| 1     | merah  | 08:50:10 |
| 1     | biru   | 12:30:30 |
| 2     | putih  | 18:10:45 |
| 2     | kuning  | 00:30:59 |
| 1     | jingga  | 20:30:30 |
| 2     | hitam  | 19:45:30 |
| 1     | merah  | 03:35:10 |
| 1     | merah  | 23:15:30 |
| 2     | hitam  | 10:15:15 |
+-----+-----+-----+
```

Sebagai latihan, bisakah anda mengurutkan tabel diatas berdasarkan kolom a secara menaik? Berikut query yang bisa dipakai:

```
SELECT * FROM contoh_select_1 ORDER BY a ASC;
```

a	b	c
1	merah	08:50:10
1	biru	12:30:30
1	jingga	20:30:30
1	merah	03:35:10
1	merah	23:15:30
2	putih	18:10:45
2	kuning	00:30:59
2	hitam	19:45:30
2	hitam	10:15:15

Meskipun ASC merupakan pengurutan default dari perintah ORDER BY, saya tetap menulisnya agar lebih jelas bahwa kolom tersebut diurutkan secara menaik.

Dari hasil yang tampil, terlihat bahwa kolom b sebenarnya juga bisa diurutkan lebih lanjut. Kita bisa mengurutkan isi kolom b setelah pengurutan kolom a dengan query sebagai berikut:

```
SELECT * FROM contoh_select_1 ORDER BY a ASC, b ASC;
```

a	b	c
1	biru	12:30:30
1	jingga	20:30:30
1	merah	08:50:10
1	merah	03:35:10
1	merah	23:15:30
2	hitam	19:45:30
2	hitam	10:15:15
2	kuning	00:30:59
2	putih	18:10:45

Sekarang, tabel contoh_select_1 memiliki 2 tingkat pengurutan, yakni setelah diurutkan berdasarkan kolom a, lanjut dengan kolom b.

Bagaimana dengan pengurutan 3 tingkat? Tidak masalah:

```
SELECT * FROM contoh_select_1 ORDER BY a ASC, b ASC, c ASC;
```

a	b	c
1	biru	12:30:30
1	jingga	20:30:30
1	merah	03:35:10
1	merah	08:50:10
1	merah	23:15:30
2	hitam	10:15:15
2	hitam	19:45:30
2	kuning	00:30:59
2	putih	18:10:45

Lebih jauh lagi, urutan kolom tampilan juga tidak harus secara berurutan dari kolom a, b dan c. Tapi juga bisa dibuat seperti contoh berikut:

```
SELECT b,c,a FROM contoh_select_1 ORDER BY b ASC, c ASC, a ASC;
```

b	c	a
biru	12:30:30	1
hitam	10:15:15	2
hitam	19:45:30	2
jingga	20:30:30	1
kuning	00:30:59	2
merah	03:35:10	1
merah	08:50:10	1
merah	23:15:30	1
putih	18:10:45	2

Pengurutan bertingkat seperti diatas tidak diperlukan untuk setiap tabel. Situasi ini hanya berlaku untuk tabel yang memiliki banyak data berulang.

Di dalam tabel populasi tidak ada data yang berulang, sehingga proses pengurutan cukup di 1 tingkat saja. Meskipun ditulis untuk 2 tingkat (dua buah perintah ORDER BY), tidak akan ada pengaruh apa-apa karena proses pengurutan sudah selesai di ORDER BY pertama.

Sampai disini kita juga bisa melihat bahwa urutan fisik penyimpanan data menjadi tidak relevan. Maksudnya, bagaimana pun urutan data saat diinput (termasuk urutan kolom), tidak berpengaruh sama sekali. Kita bisa dengan mudah menampilkan data dalam bentuk lain.

13.5 SELECT ... LIMIT

Query SELECT...LIMIT bisa dipakai untuk membatasi jumlah baris yang akan ditampilkan. Query ini biasa digabung dengan perintah ORDER BY, karena pada umumnya kita ingin menampilkan

sedikit data yang sudah disusun berdasarkan urutan tertentu.

Query `SELECT...LIMIT` memiliki 2 bentuk penulisan:

```
SELECT nama_kolom FROM nama_tabel LIMIT jumlah_baris
SELECT nama_kolom FROM nama_tabel LIMIT offset, jumlah_baris
```

Jika perintah `LIMIT` diikuti oleh 1 angka saja, artinya tampilkan baris sebanyak angka tersebut, dihitung mulai dari baris pertama.

Sebagai contoh, saya ingin menampilkan 5 provinsi dengan jumlah kelurahan tertinggi dari tabel `populasi`. Querynya adalah sebagai berikut:

```
SELECT * FROM populasi ORDER BY kel DESC LIMIT 5;
+-----+-----+-----+-----+
| kota | kec | kel | luas | penduduk |
+-----+-----+-----+-----+
| Jakarta | 44 | 267 | 664.01 | 9988495 |
| Semarang | 16 | 177 | 373.78 | 1621384 |
| Surabaya | 31 | 154 | 350.54 | 2805906 |
| Medan | 21 | 151 | 265.00 | 2465469 |
| Bandung | 30 | 151 | 167.67 | 2339463 |
+-----+-----+-----+-----+
```

Jika perintah `LIMIT` diikuti oleh 2 angka, angka pertama berfungsi sebagai `offset`, yakni jumlah baris yang dilompati. Sedangkan angka kedua berfungsi sebagai jumlah baris.

Sebagai contoh, saya ingin menampilkan tabel `populasi` untuk posisi ke 6, 7 dan 8 berdasarkan jumlah kelurahan. Berikut perintahnya:

```
SELECT * FROM populasi ORDER BY kel DESC LIMIT 5, 3;
+-----+-----+-----+-----+
| kota | kec | kel | luas | penduduk |
+-----+-----+-----+-----+
| Makassar | 14 | 143 | 199.26 | 1651146 |
| Palembang | 14 | 107 | 369.22 | 1548064 |
| Tangerang | 13 | 104 | 153.93 | 1566190 |
+-----+-----+-----+-----+
```

Bagian `ORDER BY kel DESC LIMIT 5, 3` bisa dibaca: Urutkan berdasarkan kolom `kel`, lalu ambil 3 data setelah baris ke-5.

Yang juga harus diperhatikan, baris ke-5 itu sendiri tidak masuk hitungan 3 baris. Yang akan ditampilkan adalah baris ke 6, 7 dan 8.

Sebagai latihan, bisakah anda menampilkan provinsi urutan 5 dan 6 jika dihitung berdasarkan luas wilayah?

Berikut query yang saya coba:

```
SELECT * FROM populasi ORDER BY luas DESC LIMIT 4, 2;
+-----+-----+-----+-----+
| kota | kec | kel | luas | penduduk |
+-----+-----+-----+-----+
| Medan | 21 | 151 | 265.00 | 2465469 |
| Bekasi | 12 | 56 | 206.61 | 2381053 |
+-----+-----+-----+-----+
```

Untuk memastikan apakah kota Medan dan Bekasi memang berada di urutan ke 5 dan ke 6, kita bisa bandingkan hasilnya tanpa perintah `LIMIT`:

```
SELECT * FROM populasi ORDER BY luas DESC;
+-----+-----+-----+-----+
| kota | kec | kel | luas | penduduk |
+-----+-----+-----+-----+
| Jakarta | 44 | 267 | 664.01 | 9988495 |
| Semarang | 16 | 177 | 373.78 | 1621384 |
| Palembang | 14 | 107 | 369.22 | 1548064 |
| Surabaya | 31 | 154 | 350.54 | 2805906 |
| Medan | 21 | 151 | 265.00 | 2465469 |
| Bekasi | 12 | 56 | 206.61 | 2381053 |
| Depok | 11 | 63 | 200.29 | 1631951 |
| Makassar | 14 | 143 | 199.26 | 1651146 |
| Bandung | 30 | 151 | 167.67 | 2339463 |
| Tangerang | 13 | 104 | 153.93 | 1566190 |
+-----+-----+-----+-----+
```

Terlihat, kota Medan dan Bekasi memang berada di baris ke- 5 dan ke-6.

Membatasi baris seperti ini adalah dasar dari membuat fitur *pagination* di web programming. **Pagination** adalah sebutan untuk tampilan halaman web yang membagi tabel menjadi beberapa segmen. Biasanya di halaman tersebut terdapat tombol Next dan Prev, atau angka 1, 2, 3, dst.

Misalkan tabel populasi terdiri dari 200 baris. Agar lebih user friendly, saya bisa membuat pagination. Saat halaman diakses pertama kali, tampilkan hanya 10 baris saja. Jika user men-klik tombol next, tampilkan 10 data berikutnya.

Untuk membuat fitur seperti ini, digunakan perintah `LIMIT` dengan cara membagi tampilan data tabel setiap 10 baris. Mulai dari `LIMIT 0, 10` kemudian lanjut ke `LIMIT 11, 10` dan seterusnya.

Upgrade tabel populasi

Sebagai persiapan untuk materi berikutnya, saya ingin menambah 1 kolom lagi untuk tabel `populasi`. Kolom tersebut adalah `id_kota` yang berisi nomor id atau identitas dari sebuah kota.

Kolom `id_kota` ini akan di set sebagai `SMALLINT UNSIGNED PRIMARY KEY AUTO_INCREMENT` dan ditempatkan di kolom pertama. Untuk keperluan ini saya butuh menjalankan query `ALTER . . . ADD`. Tertarik untuk merancang perintahnya? Silahkan dicoba.

Selamat! jika anda berhasil menambahkan kolom tersebut. Namun sebelum menampilkan perintah yang dipakai, saya ingin melenceng sedikit dari pembahasan kita.

Seandainya tabel populasi ini adalah tabel yang sudah ‘live’ dan sedang diakses oleh banyak orang, sebaiknya kita tidak langsung mengedit tabel tersebut. Jika query yang diinput salah atau error, akibatnya bisa fatal. Terlebih lagi jika tidak ada backup untuk mengembalikan isi tabel seperti semula.

Untuk situasi seperti ini, idealnya buat backup dari tabel ‘online’, lalu utak-atik secara offline di komputer lain. Jika perintahnya sudah benar, baru implementasikan ke database online dengan menimpa tabel lama, atau bisa juga dengan mengetik langsung perintah query untuk mengupdate tabel yang sudah ada (yang sebelumnya sudah dicoba ke tabel offline).

Selama proses update, usahakan tidak ada yang mengakses tabel. Karena bisa jadi ada yang sedang menginput data selagi kita melakukan perubahan struktur tabel. Caranya bisa dengan mematikan website beberapa saat (maintenance mode).

Untuk men-simulasi situasi seperti ini, saya akan mengcopy tabel populasi ke tabel populasi_sementara, kemudian menambahkan kolom id_kota ke tabel populasi_sementara. Jika tidak ada masalah dan hasilnya susah sesuai, saya akan menghapus tabel populasi dan me-rename tabel populasi_sementara menjadi tabel populasi.

Berikut perintah yang digunakan:

```
CREATE TABLE populasi_sementara LIKE populasi;
```

```
DESC populasi_sementara;
```

Field	Type	Null	Key	Default	Extra
kota	varchar(50)	YES		NULL	
kec	tinyint(3) unsigned	YES		NULL	
kel	smallint(5) unsigned	YES		NULL	
luas	decimal(6,2)	YES		NULL	
penduduk	int(10) unsigned	YES		NULL	

```
INSERT INTO populasi_sementara SELECT * FROM populasi;
```

```
SELECT * FROM populasi_sementara;
```

kota	kec	kel	luas	penduduk
Jakarta	44	267	664.01	9988495
Surabaya	31	154	350.54	2805906
Medan	21	151	265.00	2465469
Bekasi	12	56	206.61	2381053
Bandung	30	151	167.67	2339463
Makassar	14	143	199.26	1651146

```
+-----+-----+-----+-----+
| Depok | 11 | 63 | 200.29 | 1631951 |
| Semarang | 16 | 177 | 373.78 | 1621384 |
| Tangerang | 13 | 104 | 153.93 | 1566190 |
| Palembang | 14 | 107 | 369.22 | 1548064 |
+-----+-----+-----+-----+
```

```
ALTER TABLE populasi_sementara
ADD id_kota SMALLINT UNSIGNED PRIMARY KEY AUTO_INCREMENT FIRST;
```

```
SELECT * FROM populasi_sementara;
```

```
+-----+-----+-----+-----+-----+
| id_kota | kota | kec | kel | luas | penduduk |
+-----+-----+-----+-----+-----+
| 1 | Jakarta | 44 | 267 | 664.01 | 9988495 |
| 2 | Surabaya | 31 | 154 | 350.54 | 2805906 |
| 3 | Medan | 21 | 151 | 265.00 | 2465469 |
| 4 | Bekasi | 12 | 56 | 206.61 | 2381053 |
| 5 | Bandung | 30 | 151 | 167.67 | 2339463 |
| 6 | Makassar | 14 | 143 | 199.26 | 1651146 |
| 7 | Depok | 11 | 63 | 200.29 | 1631951 |
| 8 | Semarang | 16 | 177 | 373.78 | 1621384 |
| 9 | Tangerang | 13 | 104 | 153.93 | 1566190 |
| 10 | Palembang | 14 | 107 | 369.22 | 1548064 |
+-----+-----+-----+-----+-----+
```

```
DROP TABLE populasi;
```

```
RENAME TABLE populasi_sementara TO populasi;
```

```
SELECT * FROM populasi;
```

```
+-----+-----+-----+-----+-----+
| id_kota | kota | kec | kel | luas | penduduk |
+-----+-----+-----+-----+-----+
| 1 | Jakarta | 44 | 267 | 664.01 | 9988495 |
| 2 | Surabaya | 31 | 154 | 350.54 | 2805906 |
| 3 | Medan | 21 | 151 | 265.00 | 2465469 |
| 4 | Bekasi | 12 | 56 | 206.61 | 2381053 |
| 5 | Bandung | 30 | 151 | 167.67 | 2339463 |
| 6 | Makassar | 14 | 143 | 199.26 | 1651146 |
| 7 | Depok | 11 | 63 | 200.29 | 1631951 |
| 8 | Semarang | 16 | 177 | 373.78 | 1621384 |
| 9 | Tangerang | 13 | 104 | 153.93 | 1566190 |
| 10 | Palembang | 14 | 107 | 369.22 | 1548064 |
+-----+-----+-----+-----+-----+
```

Sepanjang proses ini, saya menggunakan query SELECT dan DESC untuk memastikan struktur

tabel sudah benar atau belum.

Mungkin anda juga heran, kenapa kolom `id_kota` sudah langsung berisi angka. Ini karena kolom tersebut di set dengan atribut `AUTO_INCREMENT`. Tanpa atribut ini, kolom yang baru ditambahkan akan berisi `NULL`, atau berisi nilai default jika kolom tersebut memiliki atribut `DEFAULT`.

13.6 SELECT ... WHERE

Query `SELECT...WHERE` berguna untuk men-filter atau membatasi hasil tampilan berdasarkan kondisi tertentu. Sama seperti `AS`, perintah `WHERE` juga bisa dipakai oleh query lain, terutama query `UPDATE`.

Berikut format dasar penulisan query `SELECT...WHERE`:

```
SELECT nama_kolom FROM nama_tabel WHERE kondisi1 [[AND | OR] kondisi2, ... ]
```

Kondisi yang bisa diterapkan sangat beragam, bentuk paling sederhana adalah menggunakan operator sama dengan (`=`). Misalnya saya ingin menampilkan data tabel `populasi` yang memiliki `id_kota = 5`. Berikut perintahnya:

```
SELECT * FROM populasi WHERE id_kota = 5;
+-----+-----+-----+-----+-----+
| id_kota | kota     | kec     | kel     | luas     | penduduk |
+-----+-----+-----+-----+-----+
|      5 | Bandung |     30 |    151 | 167.67 | 2339463 |
+-----+-----+-----+-----+-----+
```

Contoh lain, menampilkan data tabel `populasi` dimana kolom `kota` berisi string `Jakarta`:

```
SELECT * FROM populasi WHERE kota = "Jakarta";
+-----+-----+-----+-----+-----+
| id_kota | kota     | kec     | kel     | luas     | penduduk |
+-----+-----+-----+-----+-----+
|      1 | Jakarta |     44 |    267 | 664.01 | 9988495 |
+-----+-----+-----+-----+-----+
```

Selain operator sama dengan (`=`), tersedia juga operator perbandingan lain. Daftar lengkapnya bisa dilihat dari tabel dibawah ini:

Operator	Penjelasan
<code>=</code>	Sama dengan
<code><></code> atau <code>!=</code>	Tidak sama dengan
<code><</code>	kurang dari
<code><=</code>	kurang dari atau sama dengan
<code>></code>	lebih besar dari
<code>>=</code>	lebih besar atau sama dengan

Sebagai contoh, saya ingin melihat daftar nama kota yang memiliki luas kurang dari 300 km²:

```
SELECT * FROM populasi WHERE luas < 300;
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     3 | Medan     | 21    | 151   | 265.00  | 2465469  |
|     4 | Bekasi     | 12    | 56    | 206.61  | 2381053  |
|     5 | Bandung    | 30    | 151   | 167.67  | 2339463  |
|     6 | Makassar   | 14    | 143   | 199.26  | 1651146  |
|     7 | Depok      | 11    | 63    | 200.29  | 1631951  |
|     9 | Tangerang | 13    | 104   | 153.93  | 1566190  |
+-----+-----+-----+-----+-----+
```

Atau daftar nama kota dengan jumlah kecamatan lebih dari 30:

```
SELECT * FROM populasi WHERE kec > 30;
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     1 | Jakarta   | 44    | 267   | 664.01  | 9988495  |
|     2 | Surabaya  | 31    | 154   | 350.54  | 2805906  |
+-----+-----+-----+-----+-----+
```

Untuk query diatas, kadang kita juga harus memastikan apakah kota dengan kecamatan yang persis berjumlah 30 diikutkan atau tidak. Jika iya, operator yang dipakai bukan besar dari (>), tapi besar dari atau sama dengan (>=).

Salah pengertian untuk hal-hal seperti ini sering menjadi masalah, karena hasil yang didapat juga akan berbeda:

```
SELECT * FROM populasi WHERE kec >= 30;
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     1 | Jakarta   | 44    | 267   | 664.01  | 9988495  |
|     2 | Surabaya  | 31    | 154   | 350.54  | 2805906  |
|     5 | Bandung    | 30    | 151   | 167.67  | 2339463  |
+-----+-----+-----+-----+-----+
```

Terlihat, kota bandung memiliki jumlah kecamatan yang pas 30. Kota bandung tidak akan tampil ketika menggunakan operator lebih besar dari (>).

Sebagai latihan, bisakah anda menampilkan data tabel populasi yang id_kota-nya selain 1? Berikut query yang saya gunakan:

```
SELECT * FROM populasi WHERE id_kota != 1;
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     2 | Surabaya | 31    | 154   | 350.54  | 2805906  |
|     3 | Medan    | 21    | 151   | 265.00  | 2465469  |
|     4 | Bekasi    | 12    | 56    | 206.61  | 2381053  |
|     5 | Bandung   | 30    | 151   | 167.67  | 2339463  |
|     6 | Makassar  | 14    | 143   | 199.26  | 1651146  |
|     7 | Depok    | 11    | 63    | 200.29  | 1631951  |
|     8 | Semarang  | 16    | 177   | 373.78  | 1621384  |
|     9 | Tangerang | 13    | 104   | 153.93  | 1566190  |
|    10 | Palembang | 14    | 107   | 369.22  | 1548064  |
+-----+-----+-----+-----+-----+
```

Untuk operator tidak sama dengan, MySQL mendukung 2 buah penulisan: != dan <>, anda bisa menggunakan yang mana saja.

Khusus untuk perbandingan dengan nilai NULL, hasil yang didapat bisa berbeda. Sebagai contoh praktek, saya akan membuat tabel contoh_select_2:

```
CREATE TABLE contoh_select_2 (
    a INT ,
    b VARCHAR(10)
);
```

```
INSERT contoh_select_2 VALUES
    (1, 'merah'),
    (NULL, 'biru'),
    (NULL, 'putih'),
    (20, NULL)
;
```

```
SELECT * FROM contoh_select_2;
+-----+-----+
| a      | b      |
+-----+-----+
| 1      | merah |
| NULL  | biru  |
| NULL  | putih |
| 20    | NULL  |
+-----+-----+
```

Mari kita coba menampilkan data dengan kondisi NULL:

```
SELECT * FROM contoh_select_2 WHERE a = NULL;
Empty set (0.00 sec)
```

```
SELECT * FROM contoh_select_2 WHERE a <> NULL;
Empty set (0.00 sec)
```

Kenapa hasilnya kosong? Hal ini terjadi karena di dalam MySQL melakukan operasi dengan nilai NULL akan menghasilkan NULL juga.

Oleh karena itu MySQL menyediakan 3 operator khusus untuk menangani nilai NULL:

Operator	Penjelasan
<code><=></code>	sama dengan (null safe)
<code>IS NULL</code>	Pengecekan apakah berisi NULL
<code>IS NOT NULL</code>	Pengecekan apakah bukan berisi NULL

Mari kita coba:

```
SELECT * FROM contoh_select_2 WHERE a <=> NULL;
+-----+-----+
| a    | b    |
+-----+-----+
| NULL | biru |
| NULL | putih |
+-----+-----+
```

```
SELECT * FROM contoh_select_2 WHERE a IS NULL;
+-----+-----+
| a    | b    |
+-----+-----+
| NULL | biru |
| NULL | putih |
+-----+-----+
```

```
SELECT * FROM contoh_select_2 WHERE a IS NOT NULL;
+-----+-----+
| a    | b    |
+-----+-----+
|    1 | merah |
|   20 | NULL |
+-----+-----+
```

Operator `<=>` juga bisa digunakan untuk data biasa (bukan NULL), dan akan berfungsi sama seperti operator sama dengan (`=`). Namun ketika memproses nilai NULL, operator `<=>` berprilaku layaknya operator `IS NULL`.

Kondisi WHERE juga bisa disambung menggunakan operasi logika AND dan OR. Keduanya berguna untuk proses penyeleksian yang butuh memeriksa beberapa kolom sekaligus.

Sebagai contoh, saya ingin menampilkan kota dengan jumlah kecamatan = 16 dan jumlah kelurahan = 177:

```
SELECT * FROM populasi WHERE kec = 16 AND kel = 177;
+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel    | luas     | penduduk |
+-----+-----+-----+-----+
|       8 | Semarang  |   16 | 177   | 373.78  | 1621384  |
+-----+-----+-----+-----+
```

Contoh lain, saya ingin menampilkan data tabel populasi yang memiliki id_kota 6 atau 9:

```
SELECT * FROM populasi WHERE id_kota = 6 OR id_kota = 9;
+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel    | luas     | penduduk |
+-----+-----+-----+-----+
|       6 | Makassar  |   14 | 143   | 199.26  | 1651146  |
|       9 | Tangerang |   13 | 104   | 153.93  | 1566190  |
+-----+-----+-----+-----+
```

Yang perlu diperhatikan, kadang kalimat instruksi tidak bisa langsung dikonversi menjadi query. Sebagai contoh, misalkan kita disuruh: “Tampilkan isi tabel populasi untuk kota Medan dan Jakarta”.

Maksud kata “dan” disini bukanlah operator AND, tapi kita diminta untuk menampilkan 2 buah data, yakni untuk kota Medan dan Jakarta. Operator yang dipakai seharusnya OR:

```
SELECT * FROM populasi WHERE kota = 'Medan' OR kota = 'Jakarta';
+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel    | luas     | penduduk |
+-----+-----+-----+-----+
|       1 | Jakarta  |   44 | 267   | 664.01  | 9988495  |
|       3 | Medan    |   21 | 151   | 265.00  | 2465469  |
+-----+-----+-----+-----+
```

Jika kita menggunakan query `kota = 'Medan' AND kota = 'Jakarta'`, tidak akan ada hasil yang tampil. Karena tentu saja tidak ada data tabel yang memiliki kedua nilai itu dalam 1 baris.

Untuk melatih pemahaman, bisakah anda merancang query untuk untuk menampilkan data berikut?

1. Tampilkan kota dengan jumlah kecamatan lebih dari 20 dengan luas kota kurang dari 300 km².

2. Tampilkan kota dengan jumlah kecamatan lebih dari 20 atau luas kota kurang dari 300 km².
3. Tampilkan kota dengan jumlah penduduk antara 1.000.000 hingga 2.000.000.

Untuk soal pertama, maksudnya adalah kota dengan jumlah kecamatan lebih dari 20 dan kota tersebut juga harus memiliki luas kurang dari 300 km². Maka querynya adalah sebagai berikut:

```
SELECT * FROM populasi WHERE kec > 20 AND luas < 300;
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec    | kel    | luas     | penduduk |
+-----+-----+-----+-----+-----+
|      3 | Medan     |   21   | 151    | 265.00   | 2465469  |
|      5 | Bandung   |   30   | 151    | 167.67   | 2339463  |
+-----+-----+-----+-----+-----+
```

Soal kedua, mirip seperti query pertama, tapi menggunakan OR:

```
SELECT * FROM populasi WHERE kec > 20 OR luas < 300;
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec    | kel    | luas     | penduduk |
+-----+-----+-----+-----+-----+
|      1 | Jakarta   |   44   | 267    | 664.01   | 9988495  |
|      2 | Surabaya  |   31   | 154    | 350.54   | 2805906  |
|      3 | Medan     |   21   | 151    | 265.00   | 2465469  |
|      4 | Bekasi     |   12   | 56     | 206.61   | 2381053  |
|      5 | Bandung   |   30   | 151    | 167.67   | 2339463  |
|      6 | Makassar  |   14   | 143    | 199.26   | 1651146  |
|      7 | Depok     |   11   | 63     | 200.29   | 1631951  |
|      9 | Tangerang |   13   | 104    | 153.93   | 1566190  |
+-----+-----+-----+-----+-----+
8 rows in set (0.00 sec)
```

Terlihat, hanya dengan mengubah logika AND menjadi OR, hasil yang ditampilkan akan berbeda jauh.

Dalam pembuatan aplikasi ‘real world’, jika memungkinkan kita bisa bertanya dengan lebih detail kepada client, seperti apa data yang diminta. Apakah syarat itu berlaku untuk kota yang sama (query AND), atau cukup memenuhi salah satu kondisi saja (query OR).

Untuk soal ketiga, disini kita akan menampilkan data dalam sebuah jangkauan (range). Querynya adalah sebagai berikut:

```
SELECT * FROM populasi WHERE penduduk > 1000000 AND penduduk < 2000000;
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     6 | Makassar  |    14 |   143 | 199.26 | 1651146 |
|     7 | Depok     |    11 |    63 | 200.29 | 1631951 |
|     8 | Semarang   |    16 |   177 | 373.78 | 1621384 |
|     9 | Tangerang |    13 |   104 | 153.93 | 1566190 |
|    10 | Palembang |    14 |   107 | 369.22 | 1548064 |
+-----+-----+-----+-----+-----+
```

Kembali, untuk situasi sebenarnya, kita bisa menanyakan ke client apakah data yang diminta termasuk juga kota yang berpenduduk persis 1.000.000 dan 2.000.000. Untuk query diatas, kota dengan kondisi tersebut tidak akan ditampilkan karena saya memakai operator `>` dan `<`.

Untuk data yang kompleks, kita juga bisa menyambung beberapa perintah AND dan OR dalam 1 query, misalnya untuk soal seperti ini:

“Tampilkan kota dengan jumlah kecamatan lebih besar dari 20, atau memiliki luas kurang dari 300 km², dimana setiap kota itu harus memiliki jumlah penduduk kurang dari 2.000.000”

Berikut query yang saya gunakan:

```
SELECT * FROM populasi WHERE (kec > 20 OR luas < 300) AND penduduk < 2000000;
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     6 | Makassar  |    14 |   143 | 199.26 | 1651146 |
|     7 | Depok     |    11 |    63 | 200.29 | 1631951 |
|     9 | Tangerang |    13 |   104 | 153.93 | 1566190 |
+-----+-----+-----+-----+-----+
```

Terdapat tambahan tanda kurung. Ini perlu ditulis untuk menegaskan logika kondisi. Tanpa tanda kurung, hasilnya menjadi berbeda:

```
SELECT * FROM populasi WHERE kec > 20 OR luas < 300 AND penduduk < 2000000;
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     1 | Jakarta  |    44 |   267 | 664.01 | 9988495 |
|     2 | Surabaya |    31 |   154 | 350.54 | 2805906 |
|     3 | Medan    |    21 |   151 | 265.00 | 2465469 |
|     5 | Bandung  |    30 |   151 | 167.67 | 2339463 |
|     6 | Makassar |    14 |   143 | 199.26 | 1651146 |
|     7 | Depok   |    11 |    63 | 200.29 | 1631951 |
|     9 | Tangerang |    13 |   104 | 153.93 | 1566190 |
+-----+-----+-----+-----+-----+
```

Kenapa bisa seperti ini?

Hal tersebut berkaitan dengan kondisi apa yang harus dijalankan terlebih dahulu, atau urutan prioritas operator.

Dalam MySQL, operator AND memiliki prioritas lebih tinggi daripada operator OR. Dengan demikian, yang sebenarnya dijalankan dari query diatas adalah sebagai berikut:

```
SELECT * FROM populasi WHERE kec > 20 OR (luas < 300 AND penduduk < 2000000);
```

Artinya, seluruh kota yang memiliki kecamatan lebih dari 20 akan ditampilkan, meskipun luasnya lebih dari 300 dan jumlah penduduknya lebih dari 2000000.

Agar tidak terjadi “ambigu” seperti ini, sebaiknya kita selalu menggunakan tanda kurung jika terdapat lebih dari 1 operator di dalam 1 query.

Berikut urutan “kekuatan” operator di dalam MySQL:

Urutan	Operator
1	INTERVAL
2	BINARY, COLLATE
3	!
4	- (unary minus), ~ (unary bit inversion)
5	^
6	*, /, DIV, %, MOD
7	- , +
8	<<, >>
9	&
10	
11	= (comparison), <=>, >=, >, <=, <, >>, !=, IS, LIKE, REGEXP, IN BETWEEN, CASE, WHEN, THEN, ELSE
13	NOT
14	AND, &&
15	XOR
16	OR,
17	= (assignment), :=

Operator yang berada di baris paling atas, memiliki prioritas lebih tinggi daripada operator yang ada di bawahnya. Sebagian besar dari operator ini memang belum kita pelajari, dan akan dibahas secara bertahap.

Dari tabel diatas juga bisa terlihat bahwa operator AND juga bisa ditulis menggunakan karakter && dan operator OR menggunakan karakter ||. Berikut contoh penggunannya:

```
SELECT * FROM populasi WHERE (kec > 20 || luas < 300) && penduduk < 2000000;
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|       6 | Makassar  |    14 |   143 | 199.26 | 1651146 |
|       7 | Depok     |    11 |    63 | 200.29 | 1631951 |
|       9 | Tangerang |    13 |   104 | 153.93 | 1566190 |
+-----+-----+-----+-----+-----+
```

Query diatas merupakan konversi dari query kita sebelumnya, kali ini saya mengganti operator AND dengan && dan operator OR dengan ||.

Membuat tabel daftar_provinsi

Sebagai persiapan untuk materi kita berikutnya, saya akan membuat tabel daftar_provinsi. Berikut query yang digunakan untuk membuat tabel ini:

```
CREATE TABLE daftar_provinsi (
    prov VARCHAR(50),
    ibukota VARCHAR(50),
    luas INT,
    tanggal_diresmikan DATE
);
```

```
INSERT INTO daftar_provinsi VALUES
    ('Sumatera Utara', 'Medan', 72981, '1956-11-29'),
    ('Sumatera Barat', 'Padang', 42297, '1957-08-09'),
    ('Jawa Barat', 'Bandung', 35245, '1950-07-04'),
    ('Jawa Tengah', 'Semarang', 33987, '1950-07-04'),
    ('Sulawesi Selatan', 'Makassar', 46116, '1960-12-13'),
    ('Bali', 'Denpasar', 5561, '1958-08-14'),
    ('Sumatera Selatan', 'Palembang', 85679, '1950-08-14'),
    ('Papua Barat', 'Manokwari', 114566, '1999-10-04')
;
```

```
SELECT * FROM daftar_provinsi;
+-----+-----+-----+-----+
| prov        | ibukota    | luas    | tanggal_diresmikan |
+-----+-----+-----+-----+
| Sumatera Utara | Medan    | 72981 | 1956-11-29 |
| Sumatera Barat | Padang   | 42297 | 1957-08-09 |
| Jawa Barat    | Bandung  | 35245 | 1950-07-04 |
| Jawa Tengah    | Semarang  | 33987 | 1950-07-04 |
| Sulawesi Selatan | Makassar | 46116 | 1960-12-13 |
```

Bali	Denpasar	5561	1958-08-14	
Sumatera Selatan	Palembang	85679	1950-08-14	
Papua Barat	Manokwari	114566	1999-10-04	

Tabel `daftar_provinsi` memiliki 4 kolom: `prov`, `ibukota`, `luas` dan `tanggal_diresmikan`.

Seperti yang terlihat, isi tabel diatas sepertinya berkaitan dengan tabel `populasi`. Dan memang inilah yang akan kita bahas dalam materi berikutnya.

13.7 SELECT ... WHERE table1 = table2

Materi kita kali ini masih berhubungan dengan query `SELECT ... WHERE`, tapi dengan pembahasan yang sedikit ‘njelimet’, yakni penggunaan perintah `WHERE` untuk menampilkan data dari 2 tabel.

Sebagai *sample*, saya akan memakai tabel `populasi` dan tabel `daftar_provinsi`. Berikut isi dari kedua tabel tersebut:

```
SELECT * FROM populasi;
```

id_kota	kota	kec	kel	luas	penduduk
1	Jakarta	44	267	664.01	9988495
2	Surabaya	31	154	350.54	2805906
3	Medan	21	151	265.00	2465469
4	Bekasi	12	56	206.61	2381053
5	Bandung	30	151	167.67	2339463
6	Makassar	14	143	199.26	1651146
7	Depok	11	63	200.29	1631951
8	Semarang	16	177	373.78	1621384
9	Tangerang	13	104	153.93	1566190
10	Palembang	14	107	369.22	1548064

```
SELECT * FROM daftar_provinsi;
```

prov	ibukota	luas	tanggal_diresmikan
Sumatera Utara	Medan	72981	1956-11-29
Sumatera Barat	Padang	42297	1957-08-09
Jawa Barat	Bandung	35245	1950-07-04
Jawa Tengah	Semarang	33987	1950-07-04
Sulawesi Selatan	Makassar	46116	1960-12-13
Bali	Denpasar	5561	1958-08-14
Sumatera Selatan	Palembang	85679	1950-08-14
Papua Barat	Manokwari	114566	1999-10-04

Silahkan anda pelajari sebentar struktur dan isi dari kedua tabel diatas.

Perhatikan bahwa baik tabel populasi maupun tabel daftar_provinsi memiliki 1 kolom yang isinya kemungkinan bisa sama, yakni kolom kota di dalam tabel populasi, dan kolom ibukota di tabel daftar_provinsi.

Sebagai contoh, kota ‘Medan’ bisa muncul di dalam kedua kolom tersebut. Ini adalah kunci dalam proses penyeleksian dari 2 buah tabel.

Sebelum kita sampai kesana, masih ingatkah anda dengan pembahasan tentang *identifier qualifiers*?

Identifier Qualifiers adalah cara penulisan nama kolom / tabel dengan menulis lengkap nama tabel / database-nya. Sebagai contoh, dalam query berikut saya menampilkan kolom kota dan penduduk dari tabel populasi dengan penulisan *identifier qualifiers*:

```
SELECT populasi.kota, populasi.penduduk FROM populasi;
+-----+-----+
| kota      | penduduk |
+-----+-----+
| Jakarta   | 9988495 |
| Surabaya  | 2805906 |
| Medan     | 2465469 |
| Bekasi    | 2381053 |
| ...       | ...      |
+-----+-----+
```

Disini saya menulis dalam format `nama_tabel.nama_kolom`. Tentu juga bisa menulis lengkap hingga ke nama database, seperti `belajar.populasi.kota`, namun kali ini kita cukup sampai ke nama tabel saja.

Contoh berikutnya, bisakah anda menulis query untuk menampilkan kolom prov dan ibukota dari tabel daftar_provinsi menggunakan penulisan *identifier qualifiers* ?

Berikut query yang saya gunakan:

```
SELECT daftar_provinsi.prov, daftar_provinsi.ibukota FROM daftar_provinsi;
+-----+-----+
| prov        | ibukota   |
+-----+-----+
| Sumatera Utara | Medan     |
| Sumatera Barat  | Padang    |
| Jawa Barat    | Bandung   |
| ...          | ...       |
+-----+-----+
```

Penulisan *identifier qualifiers* seperti ini diperlukan sebagai dasar untuk memahami query berikut ini:

```

SELECT
    daftar_provinsi.prov, populasi.kota, populasi.penduduk
FROM
    daftar_provinsi, populasi
WHERE
    daftar_provinsi.ibukota = populasi.kota;
+-----+-----+-----+
| prov      | kota      | penduduk |
+-----+-----+-----+
| Sumatera Utara | Medan      | 2465469  |
| Jawa Barat     | Bandung    | 2339463  |
| Sulawesi Selatan | Makassar   | 1651146  |
| Jawa Tengah     | Semarang   | 1621384  |
| Sumatera Selatan | Palembang | 1548064  |
+-----+-----+-----+

```

Bisakah anda memahami query diatas? Saya sengaja memecahnya menjadi beberapa baris agar lebih jelas, tapi query ini tetap bisa ditulis dalam 1 baris panjang.

Untuk bagian nama kolom, saya menulis 3 buah kolom: `SELECT daftar_provinsi.prov, populasi.kota, populasi.penduduk FROM daftar_provinsi, populasi.` Artinya, saya ingin menampilkan kolom yang berada di 2 buah tabel terpisah: tabel `daftar_provinsi` dan tabel `populasi`.

Ketika kita mencoba menampilkan kolom dari dua buah tabel atau lebih, harus ada 1 kolom yang bertindak sebagai penghubung. Dalam contoh diatas, kolom penghubung tersebut ditulis di dalam perintah WHERE, yakni `WHERE daftar_provinsi.ibukota = populasi.kota`.

Maksudnya, selama nilai di dalam kolom `ibukota` dari tabel `daftar_provinsi` sama dengan nilai kolom `kota` dari tabel `populasi`, tampilkan isi kolom `prov` dari tabel `daftar_provinsi` serta kolom `kota` dan kolom `penduduk` dari tabel `populasi`.

Silahkan anda pahami sejenak penjelasan diatas dan bandingkan dengan query serta hasil yang didapat.

Pertanyaan berikutnya, kenapa hanya 5 baris? Padahal tabel `populasi` berisi 10 baris, dan tabel `daftar_provinsi` berisi 8 baris.

Ini karena kondisi WHERE `daftar_provinsi.ibukota = populasi.kota` hanya bisa dipenuhi untuk kelima kota tersebut. Kelima kota itulah yang ada di kedua tabel, seperti gambar dibawah ini:

```

SELECT * FROM daftar_provinsi;
+-----+-----+-----+-----+
| prov | ibukota | luas | tanggal_diresmikan |
+-----+-----+-----+-----+
| Sumatera Utara | Medan | 72981 | 1956-11-29 |
| Sumatera Barat | Padang | 42297 | 1957-08-09 |
| Jawa Barat | Bandung | 35245 | 1950-07-04 |
| Jawa Tengah | Semarang | 33987 | 1950-07-04 |
| Sulawesi Selatan | Makassar | 46116 | 1960-12-13 |
| Bali | Denpasar | 5561 | 1958-08-14 |
| Sumatera Selatan | Palembang | 85679 | 1950-08-14 |
| Papua Barat | Manokwari | 114566 | 1999-10-04 |
+-----+-----+-----+-----+

```



```

SELECT * FROM populasi;
+-----+-----+-----+-----+-----+
| id_kota | kota | kec | kel | luas | penduduk |
+-----+-----+-----+-----+-----+
| 1 | Jakarta | 44 | 267 | 664.01 | 9988495 |
| 2 | Surabaya | 31 | 154 | 350.54 | 2805906 |
| 3 | Medan | 21 | 151 | 265.00 | 2465469 |
| 4 | Bekasi | 12 | 56 | 206.61 | 2381053 |
| 5 | Bandung | 30 | 151 | 167.67 | 2339463 |
| 6 | Makassar | 14 | 143 | 199.26 | 1651146 |
| 7 | Depok | 11 | 63 | 200.29 | 1631951 |
| 8 | Semarang | 16 | 177 | 373.78 | 1621384 |
| 9 | Tangerang | 13 | 104 | 153.93 | 1566190 |
| 10 | Palembang | 14 | 107 | 369.22 | 1548064 |
+-----+-----+-----+-----+-----+

```

Gambar: Lima kota yang ada di tabel populasi dan daftar_provinsi

Sebenarnya, kita juga bisa menjalankan query yang sama tanpa menggunakan penulisan *identifier qualifiers*:

```

SELECT
    prov, kota, penduduk
FROM
    daftar_provinsi, populasi
WHERE
    ibukota = kota;

```

Jika ditulis seperti ini, kita tidak bisa membedakan kolom mana, kepunyaan tabel apa. Kecuali harus melihat satu-satu struktur kedua tabel.

Proses menampilkan gabungan tabel seperti diatas juga tidak berpengaruh ke urutan kolom maupun urutan nama tabel. Saya bisa saja menulis query diatas sebagai berikut:

```

SELECT
    prov, kota, penduduk
FROM
    populasi, daftar_provinsi
WHERE
    kota = ibukota;
+-----+-----+-----+
| prov | kota | penduduk |
+-----+-----+-----+
| Sumatera Utara | Medan | 2465469 |
| Jawa Barat | Bandung | 2339463 |
| Sulawesi Selatan | Makassar | 1651146 |
| Jawa Tengah | Semarang | 1621384 |
| Sumatera Selatan | Palembang | 1548064 |
+-----+-----+-----+

```

Disini saya mengganti baris **FROM daftar_provinsi, populasi WHERE ibukota = kota** menjadi **FROM populasi, daftar_provinsi WHERE kota = ibukota**.

Penulisan singkat seperti dua query sebelumnya hanya bisa dipakai jika tidak ada bentrok pada nama kolom. Seandainya terdapat nama kolom yang sama di kedua kolom, akan tampil pesan error karena MySQL ‘bingung’ untuk memilih kolom mana untuk tabel yang mana.

Error seperti ini sering terjadi, karena itu saya akan perlihatkan contoh kasusnya. Mari kita ubah nama kolom `ibukota` di tabel `daftar_provinsi` menjadi `kota`:

```
ALTER TABLE daftar_provinsi CHANGE ibukota kota VARCHAR(10);
```

```
SELECT * FROM daftar_provinsi;
+-----+-----+-----+-----+
| prov | kota | luas | tanggal_diresmikan |
+-----+-----+-----+-----+
| Sumatera Utara | Medan | 72981 | 1956-11-29 |
| Sumatera Barat | Padang | 42297 | 1957-08-09 |
| ... | ... | ... | ... |
+-----+-----+-----+-----+
```

Sekarang, kolom `kota` ada di dua tabel: `daftar_provinsi` dan tabel `populasi`. Mari jalankan query yang sama dengan sebelumnya, hanya saja kali ini kita ganti nama kolom `ibukota` menjadi `kota`:

```
SELECT
    prov, kota, penduduk
FROM
    daftar_provinsi, populasi
WHERE
    kota = kota;
```

```
ERROR 1052 (23000): Column 'kota' in field list is ambiguous
```

Disini MySQL ‘bingung’ dengan kolom `kota` (*ambiguous*), apakah kolom `kota` ini maksudnya kepunyaan tabel `daftar_provinsi`, ataukah kolom `kota` dari tabel `populasi`?

Untuk menghindari hal ini, kita harus menulis nama tabelnya menggunakan penulisan *identifier qualifiers*:

```
SELECT
    prov, populasi.kota, penduduk
FROM
    daftar_provinsi, populasi
WHERE
    daftar_provinsi.kota = populasi.kota;
+-----+-----+-----+
| prov | kota | penduduk |
+-----+-----+-----+
```

Sumatera Utara Medan 2465469
Jawa Barat Bandung 2339463
Sulawesi Selatan Makassar 1651146
Jawa Tengah Semarang 1621384
Sumatera Selatan Palembang 1548064

Intinya, jika anda menemukan pesan error yang ada kata-kata ambigu (*ambiguous*), besar kemungkinan ada nama kolom yang sama di kedua tabel.

Mari kita ubah kembali nama tabel kota untuk tabel `daftar_populasi` menjadi `ibukota`:

```
ALTER TABLE daftar_provinsi CHANGE kota ibukota VARCHAR(10);
```

SELECT * FROM daftar_provinsi;			
prov	ibukota	luas	tanggal_diresmikan
Sumatera Utara Medan 72981 1956-11-29			
Sumatera Barat Padang 42297 1957-08-09			
... 			

Sebagai latihan, bisakah anda merancang query untuk soal berikut?

“Tampilkan kolom `prov` dan `tanggal_diresmikan` dari tabel `daftar_provinsi`, serta kolom `kota`, `penduduk`, `kec`, dan `kel` dari tabel `populasi`, dimana isi kolom `ibukota` dari tabel `daftar_provinsi` sama dengan nama kota dari tabel `populasi`”.

Berdasarkan soal diatas, kita perlu menampilkan 6 kolom, 2 dari tabel `daftar_provinsi` serta 4 dari tabel `populasi`. Berikut query yang saya gunakan:

SELECT					
prov	tanggal_diresmikan	kota	penduduk	kec	kel
FROM					
populasi, daftar_provinsi					
WHERE					
kota = ibukota;					

Lebih jauh lagi, saya akan modifikasi query tersebut agar diurutkan berdasarkan kolom prov, menggunakan penulisan *identifier qualifiers* serta mengganti judul kolom dengan perintah AS:

```
SELECT
    daftar_provinsi.prov AS 'Nama Provinsi',
    daftar_provinsi.tanggal_diresmikan AS 'Tanggal Diresmikan',
    populasi.kota AS 'Ibukota',
    populasi.penduduk AS 'Jumlah penduduk',
    populasi.kec AS 'Jumlah Kecamatan',
    populasi.kel AS 'Jumlah Kelurahan'

FROM
    populasi, daftar_provinsi

WHERE
    kota = ibukota

ORDER BY
    prov ASC

;
```

```
MariaDB [belajar]> SELECT
    -> daftar_provinsi.prov AS 'Nama Provinsi',
    -> daftar_provinsi.tanggal_diresmikan AS 'Tanggal Diresmikan',
    -> populasi.kota AS 'Ibukota',
    -> populasi.penduduk AS 'Jumlah penduduk',
    -> populasi.kec AS 'Jumlah Kecamatan',
    -> populasi.kel AS 'Jumlah Kelurahan'
    -> FROM
    -> populasi, daftar_provinsi
    -> WHERE
    -> kota = ibukota
    -> ORDER BY
    -> prov ASC
    -> ;
+-----+-----+-----+-----+-----+-----+
| Nama Provinsi | Tanggal Diresmikan | Ibukota | Jumlah penduduk | Jumlah Kecamatan | Jumlah Kelurahan |
+-----+-----+-----+-----+-----+-----+
| Jawa Barat   | 1950-07-04       | Bandung |      2339463 |        30 |       151 |
| Jawa Tengah  | 1950-07-04       | Semarang |     1621384 |        16 |       177 |
| Sulawesi Selatan | 1960-12-13 | Makassar |     1651146 |        14 |       143 |
| Sumatera Selatan | 1950-08-14 | Palembang |    1548064 |        14 |       107 |
| Sumatera Utara | 1956-11-29       | Medan   |     2465469 |        21 |       151 |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

MariaDB [belajar]> .
```

Gambar: Tampilan query WHERE dengan 2 tabel

Terlihat sangat panjang? Memang. Tapi sebenarnya semua materi untuk query diatas sudah kita bahas sejak awal bab.

Lebih lanjut tentang menampilkan 2 tabel seperti ini akan kembali kita singgung di materi tentang query `SELECT...JOIN`.

13.8 SELECT ... WHERE ... IN

Query `SELECT...WHERE...IN` digunakan untuk proses seleksi yang melibatkan himpunan. Berikut format dasar penulisannya:

```
SELECT nama_kolom FROM nama_tabel WHERE nama_kolom IN (nilai1, nilai2, ...)
```

Sebagai contoh, saya ingin menampilkan tabel populasi dimana kolom kota ada di dalam himpunan Jakarta, Bandung dan Medan:

```
SELECT * FROM populasi WHERE kota IN ('Jakarta', 'Bandung', 'Medan');
```

id_kota	kota	kec	kel	luas	penduduk
1	Jakarta	44	267	664.01	9988495
3	Medan	21	151	265.00	2465469
5	Bandung	30	151	167.67	2339463

Query **SELECT... WHERE... IN** sebenarnya mirip dengan operator **OR** yang digabung bersama. Contoh diatas juga bisa dikonversi menjadi berikut ini:

```
SELECT * FROM populasi WHERE
kota = 'Jakarta' OR kota = 'Bandung' OR kota = 'Medan';
```

id_kota	kota	kec	kel	luas	penduduk
1	Jakarta	44	267	664.01	9988495
3	Medan	21	151	265.00	2465469
5	Bandung	30	151	167.67	2339463

Di dalam MySQL, perintah **IN** termasuk ke dalam kelompok operator. Bisa juga dikatakan bahwa operator **IN** sama dengan penulisan singkat untuk banyak operator **OR**.

Kita juga bisa menggunakan perintah **NOT IN** untuk membalik logika. Sebagai contoh, saya ingin menampilkan tabel populasi dimana kolom kota selain yang ada di dalam himpunan Jakarta, Bandung dan Medan:

```
SELECT * FROM populasi WHERE kota NOT IN ('Jakarta', 'Bandung', 'Medan');
```

id_kota	kota	kec	kel	luas	penduduk
2	Surabaya	31	154	350.54	2805906
4	Bekasi	12	56	206.61	2381053
6	Makassar	14	143	199.26	1651146
7	Depok	11	63	200.29	1631951
8	Semarang	16	177	373.78	1621384
9	Tangerang	13	104	153.93	1566190
10	Palembang	14	107	369.22	1548064

Kali ini, yang ditampilkan adalah kota selain Jakarta, Bandung dan Medan.

Untuk query yang lebih rumit, isi himpunan untuk operator IN juga bisa berasal dari hasil query SELECT.

Sebagai contoh, query berikut akan menghasilkan nama-nama kota yang ada di dalam tabel daftar_provinsi:

```
SELECT ibukota FROM daftar_provinsi;
+-----+
| ibukota |
+-----+
| Medan   |
| Padang  |
| Bandung |
| Semarang|
| Makassar|
| Denpasar|
| Palembang|
| Manokwari|
+-----+
```

Hasil ini bisa menjadi isi dari himpunan IN:

```
SELECT * FROM populasi WHERE kota IN (SELECT ibukota FROM daftar_provinsi);
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec | kel  | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     3  | Medan     |  21 | 151  | 265.00  | 2465469  |
|     5  | Bandung   |  30 | 151  | 167.67  | 2339463  |
|     6  | Makassar  |  14 | 143  | 199.26  | 1651146  |
|     8  | Semarang  |  16 | 177  | 373.78  | 1621384  |
|    10  | Palembang|  14 | 107  | 369.22  | 1548064  |
+-----+-----+-----+-----+-----+
```

Query diatas juga sama artinya dengan query berikut:

```
SELECT * FROM populasi WHERE kota IN ('Medan', 'Padang', 'Bandung',
'Semarang', 'Makassar', 'Denpasar', 'Palembang', 'Manokwari');
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec | kel  | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     3  | Medan     |  21 | 151  | 265.00  | 2465469  |
|     5  | Bandung   |  30 | 151  | 167.67  | 2339463  |
|     6  | Makassar  |  14 | 143  | 199.26  | 1651146  |
|     8  | Semarang  |  16 | 177  | 373.78  | 1621384  |
|    10  | Palembang|  14 | 107  | 369.22  | 1548064  |
+-----+-----+-----+-----+-----+
```

Terlihat, meskipun isi di dalam himpunan IN ada 8 kota, yang ditampilkan hanya 5 kota. Karena 5 kota inilah yang terdapat di dalam tabel populasi.

13.9 SELECT ... ANY, SELECT ... SOME, SELECT ... ALL

Query SELECT...ANY, SELECT...SOME dan SELECT...ALL masih berhubungan dengan query SELECT...IN yang baru saja kita bahas. Ketiga query ini dipakai untuk operasi perbandingan dengan himpunan yang di dapat dari hasil **subquery** (sebuah query di dalam query lainnya).



Penjelasan tentang query ANY, SOME, dan ALL memang sedikit rumit. Anda boleh melewatkannya jika kurang paham. Query ini juga relatif jarang dipakai.

Sebelum masuk ke penjelasan tentang ANY, SOME, dan ALL, saya akan kembali membahas query SELECT...IN yang melibatkan *subquery*. Perhatikan contoh berikut:

```
SELECT * FROM populasi WHERE kota IN (SELECT ibukota FROM daftar_provinsi);
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec | kel | luas   | penduduk |
+-----+-----+-----+-----+-----+
|     3 | Medan    |  21 | 151 | 265.00 | 2465469 |
|     5 | Bandung   |  30 | 151 | 167.67 | 2339463 |
|     6 | Makassar  |  14 | 143 | 199.26 | 1651146 |
|     8 | Semarang  |  16 | 177 | 373.78 | 1621384 |
|    10 | Palembang |  14 | 107 | 369.22 | 1548064 |
+-----+-----+-----+-----+-----+
```

Perintah yang ada di dalam query IN merupakan contoh **subquery**, yakni `SELECT ibukota FROM daftar_provinsi`. Subquery ini membentuk himpunan seluruh nama ibukota yang ada di dalam tabel `daftar_provinsi`. Hasil himpunan ini selanjutnya dipakai oleh query IN.

Dalam tabel `populasi`, terdapat 10 nama kota. Akan tetapi tidak seluruh kota ini ada di dalam tabel `daftar_provinsi`. Karena itulah yang tampil dari query diatas hanya 5 kota saja. Yaitu kota yang ada di dalam tabel `populasi`, dan juga ada di dalam himpunan ibukota `daftar_provinsi`.

Secara keseluruhan, query diatas dipakai untuk menampilkan tabel `populasi`, dimana nama kotanya juga ada di dalam himpunan nama ibukota dari tabel `daftar_provinsi`.

Konsep query diatas sangat dibutuhkan untuk bisa memahami cara penggunaan query ANY, SOME dan ALL. Silahkan anda pahami sebentar sebelum kita lanjut ke penjelasannya. Query ANY, SOME dan ALL sebenarnya mirip seperti query `SELECT...IN`.

Tambahannya, di ketiga query ini kita bisa menggunakan berbagai operator perbandingan, seperti `=`, `<>`, `!=`, `>`, `>=`, `<`, dan `<=`. Semuanya dipakai untuk membuat sebuah kondisi pada himpunan.

Query ANY dan SOME dipakai untuk membandingkan **salah satu data** himpunan subquery. Sedangkan query ALL akan membandingkan **seluruh data** yang ada di himpunan subquery.

Berikut contoh penggunaannya:

```
SELECT * FROM populasi WHERE kota = ANY (SELECT ibukota FROM daftar_provinsi);
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec | kel | luas   | penduduk |
+-----+-----+-----+-----+-----+
|     3 | Medan    | 21  | 151 | 265.00 | 2465469 |
|     5 | Bandung  | 30  | 151 | 167.67 | 2339463 |
|     6 | Makassar | 14  | 143 | 199.26 | 1651146 |
|     8 | Semarang | 16  | 177 | 373.78 | 1621384 |
|    10 | Palembang | 14  | 107 | 369.22 | 1548064 |
+-----+-----+-----+-----+-----+
```

Query diatas bisa dibaca:

“Tampilkan seluruh data yang ada di tabel populasi, dimana nama kotanya sama dengan salah satu himpunan nama ibukota yang ada di dalam tabel daftar_provinsi”.

Sebagaimana yang bisa kita lihat, subquery SELECT ibukota FROM daftar_provinsi berisi 8 nama ibukota, yakni:

```
SELECT ibukota FROM daftar_provinsi;
+-----+
| ibukota |
+-----+
| Medan   |
| Padang  |
| Bandung |
| Semarang |
| Makassar |
| Denpasar |
| Palembang |
| Manokwari |
+-----+
```

Query `SELECT * FROM populasi WHERE kota = ANY (SELECT ibukota FROM daftar_provinsi)` akan mengecek satu per satu apakah nama kota di tabel populasi sama dengan salah satu himpunan ibukota di tabel daftar_provinsi. Jika ada, data tersebut ditampilkan.

Dalam MySQL, query `SELECT...SOME` adalah nama lain dari `SELECT...ANY`, jadi bisa saling ditukar:

```
SELECT * FROM populasi WHERE kota = SOME(SELECT ibukota FROM daftar_provinsi);
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec | kel | luas   | penduduk |
+-----+-----+-----+-----+-----+
|     3 | Medan    |  21 | 151 | 265.00 | 2465469 |
|     5 | Bandung   |  30 | 151 | 167.67 | 2339463 |
|     6 | Makassar  |  14 | 143 | 199.26 | 1651146 |
|     8 | Semarang  |  16 | 177 | 373.78 | 1621384 |
|    10 | Palembang |  14 | 107 | 369.22 | 1548064 |
+-----+-----+-----+-----+-----+
```

Terlihat juga bahwa query `SELECT... = ANY` maupun `SELECT... = SOME` menghasilkan tampilan yang sama dengan query `SELECT... IN`. Namun ini hanya berlaku untuk operator `(=) ANY` / `(=) SOME`.

Di lain pihak, query `SELECT... ALL` akan membandingkan **seluruh data** yang ada di himpunan *subquery*, bukan hanya salah satu saja seperti query `SELECT... ANY`. Berikut contohnya:

```
SELECT * FROM populasi WHERE kota != ALL(SELECT ibukota FROM daftar_provinsi);
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec | kel | luas   | penduduk |
+-----+-----+-----+-----+-----+
|     1 | Jakarta  |  44 | 267 | 664.01 | 9988495 |
|     2 | Surabaya |  31 | 154 | 350.54 | 2805906 |
|     4 | Bekasi    |  12 |  56 | 206.61 | 2381053 |
|     7 | Depok    |  11 |  63 | 200.29 | 1631951 |
|     9 | Tangerang |  13 | 104 | 153.93 | 1566190 |
+-----+-----+-----+-----+-----+
```

Query diatas bisa dibaca:

“Tampilkan seluruh data yang ada di tabel `populasi`, dimana nama kotanya tidak sama dengan **seluruh** himpunan nama ibukota yang ada di dalam tabel `daftar_provinsi`”.

Lima kota yang ada di tampilan diatas tidak ditemukan di dalam himpunan tabel ibukota di tabel `daftar_provinsi`. Jika ditulis seperti ini, query `!= ALL` sama pengertiannya dengan query `NOT IN`.

Keunggulan dari query `ALL` dibandingkan `IN` adalah, kita bisa membuat kondisi perbandingan, seperti contoh berikut:

```
SELECT * FROM populasi WHERE kel <= ALL (SELECT kel FROM populasi);
+-----+-----+-----+-----+
| id_kota | kota    | kec    | kel    | luas    | penduduk |
+-----+-----+-----+-----+
|      4 | Bekasi |   12 |    56 | 206.61 | 2381053 |
+-----+-----+-----+-----+
```

Query diatas bisa dibaca:

“Tampilkan seluruh data yang ada di tabel populasi, dimana jumlah kecamatannya lebih kecil atau sama dengan himpunan seluruh jumlah kecamatan dari tabel populasi”.

Kunci dari memahami query diatas adalah di kata “lebih kecil atau sama dengan”. Yang bisa memenuhi syarat ini hanyalah kota yang memiliki jumlah kecamatan terkecil (memenuhi unsur “sama dengan”).

Dengan logika yang sama, bisakah anda merancang query untuk mencari kota dengan jumlah penduduk terbesar?

Kali ini kita butuh operator `>= ALL`, yang querynya adalah sebagai berikut:

```
SELECT * FROM populasi WHERE penduduk >= ALL (SELECT penduduk FROM populasi);
+-----+-----+-----+-----+
| id_kota | kota    | kec    | kel    | luas    | penduduk |
+-----+-----+-----+-----+
|      1 | Jakarta |   44 |   267 | 664.01 | 9988495 |
+-----+-----+-----+-----+
```

Memahami penggunaan query ANY, SOME dan ALL memang cukup rumit. Dalam banyak situasi, penggunaannya bisa diganti dengan query yang lebih sederhana.

Misalnya jika saya ingin menampilkan kota dengan populasi terbanyak, bisa menggunakan perpaduan query ORDER BY dan LIMIT:

```
SELECT * FROM populasi ORDER BY penduduk DESC LIMIT 1;
+-----+-----+-----+-----+
| id_kota | kota    | kec    | kel    | luas    | penduduk |
+-----+-----+-----+-----+
|      1 | Jakarta |   44 |   267 | 664.01 | 9988495 |
+-----+-----+-----+-----+
```

Query diatas jauh lebih mudah dipahami dibandingkan query `>= ALL` kita sebelumnya. Namun setidaknya anda bisa memahami konsep dasar dari query SELECT...ANY, SELECT...SOME dan SELECT...ALL dalam MySQL.

13.10 SELECT ... EXISTS

Query SELECT ... EXISTS digunakan untuk mengecek apakah sebuah *subquery* ada atau tidak. Hasilnya **true** jika terdapat minimal 1 baris, atau **false** jika tidak ditemukan hasil apa-apa. Saat ditampilkan, nilai *true* akan dikonversi menjadi angka 1, dan nilai *false* dikonversi menjadi angka 0.

Berikut contoh penggunaannya:

```
SELECT EXISTS (SELECT * FROM populasi WHERE kota = 'Jakarta');
+-----+
| EXISTS (SELECT * FROM populasi WHERE kota = 'Jakarta') |
+-----+
|               1 |
+-----+  
  
SELECT EXISTS (SELECT * FROM populasi WHERE kota = 'Lampung');
+-----+
| EXISTS (SELECT * FROM populasi WHERE kota = 'Lampung') |
+-----+
|               0 |
+-----+
```

Di dalam tabel populasi tidak ada kota Lampung, karena itu query SELECT...EXISTS menghasilkan 0.

Contoh lain penerapannya seperti query berikut ini:

```
SELECT * FROM populasi WHERE
EXISTS (SELECT ibukota FROM daftar_provinsi WHERE ibukota = "Lampung");  
  
Empty set (0.00 sec)
```

Tidak ada tampilan apa-apa karena kondisi EXISTS akan menghasilkan nilai 0. Hal ini disebabkan tidak ada nama ibukota Lampung di tabel daftar_provinsi.

Jika kondisi subquery EXISTS menghasilkan nilai 1, seluruh data tabel populasi akan ditampilkan:

```
SELECT * FROM populasi WHERE
EXISTS (SELECT ibukota FROM daftar_provinsi WHERE ibukota = "Medan");
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec     | kel    | luas     | penduduk |
+-----+-----+-----+-----+-----+
| 1 | Jakarta | 44 | 267 | 664.01 | 9988495 |
| 2 | Surabaya | 31 | 154 | 350.54 | 2805906 |
| 3 | Medan   | 21 | 151 | 265.00 | 2465469 |
| ... | ... | ... | ... | ... | ... |
+-----+-----+-----+-----+-----+
```

Terlihat bahwa perintah EXISTS disini hanya digunakan untuk pengecekan kondisi.

Selain itu, kita bisa *menegasikan* atau membalik logika dengan query NOT EXIST. Dengan query ini, hasilnya 1 jika tidak ada data, dan 0 jika terdapat data yang bisa ditemukan:

```
SELECT NOT EXISTS (SELECT * FROM populasi WHERE kota = 'Jakarta');
+-----+
| NOT EXISTS (SELECT * FROM populasi WHERE kota = 'Jakarta') |
+-----+
| 0 |
+-----+
```

```
SELECT NOT EXISTS (SELECT * FROM populasi WHERE kota = 'Lampung');
+-----+
| NOT EXISTS (SELECT * FROM populasi WHERE kota = 'Lampung') |
+-----+
| 1 |
+-----+
```

Query EXISTS dan NOT EXISTS biasa dipakai untuk membuat query yang cukup kompleks.

13.11 SELECT ... WHERE ... BETWEEN

Query SELECT...WHERE...BETWEEN digunakan untuk membuat seleksi atau kondisi berdasarkan jangkauan (range). Di dalam MySQL, BETWEEN merupakan sebuah operator. Berikut format dasar penulisannya:

```
SELECT nama_kolom FROM nama_tabel WHERE nama_kolom BETWEEN nilai1 AND nilai2
```

Sebagai contoh, saya ingin menampilkan isi tabel populasi dimana nilai kolom kecamatan antara 20 dan 30:

```
SELECT * FROM populasi WHERE kec BETWEEN 20 AND 30;
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     3 | Medan     | 21    | 151   | 265.00  | 2465469  |
|     5 | Bandung   | 30    | 151   | 167.67  | 2339463  |
+-----+-----+-----+-----+-----+
```

Contoh lain, saya ingin menampilkan isi tabel populasi dimana nilai kolom penduduk antara 1.000.000 dan 2.000.000:

```
SELECT * FROM populasi WHERE penduduk BETWEEN 1000000 AND 2000000;
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     6 | Makassar  | 14    | 143   | 199.26  | 1651146  |
|     7 | Depok     | 11    | 63    | 200.29  | 1631951  |
|     8 | Semarang   | 16    | 177   | 373.78  | 1621384  |
|     9 | Tangerang | 13    | 104   | 153.93  | 1566190  |
|    10 | Palembang | 14    | 107   | 369.22  | 1548064  |
+-----+-----+-----+-----+-----+
```

Dengan menggunakan operator NOT BETWEEN, kita bisa membalik logika. Misalnya saya ingin menampilkan isi tabel populasi dimana nilai kolom penduduk bukan diantara 1.000.000 dan 2.000.000:

```
SELECT * FROM populasi WHERE penduduk NOT BETWEEN 1000000 AND 2000000;
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     1 | Jakarta   | 44    | 267   | 664.01  | 9988495  |
|     2 | Surabaya  | 31    | 154   | 350.54  | 2805906  |
|     3 | Medan     | 21    | 151   | 265.00  | 2465469  |
|     4 | Bekasi    | 12    | 56    | 206.61  | 2381053  |
|     5 | Bandung   | 30    | 151   | 167.67  | 2339463  |
+-----+-----+-----+-----+-----+
```

Yang cukup unik, jangkauan dari operator BETWEEN juga berlaku untuk kolom dengan tipe data string. Berikut contohnya:

```
SELECT * FROM populasi WHERE kota BETWEEN 'Bekasi' AND 'Medan';
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     1 | Jakarta   | 44    | 267   | 664.01  | 9988495  |
|     3 | Medan     | 21    | 151   | 265.00  | 2465469  |
|     4 | Bekasi     | 12    | 56    | 206.61  | 2381053  |
|     6 | Makassar   | 14    | 143   | 199.26  | 1651146  |
|     7 | Depok     | 11    | 63    | 200.29  | 1631951  |
+-----+-----+-----+-----+-----+
```

Jangkauan antara ‘Bekasi’ dan ‘Medan’ menggunakan aturan huruf alphabet. Artinya, seluruh kota dengan awalan antara B dan M masuk di dalam jangkauan ini.

Agar lebih mudah dilihat, kita bisa menambahkan perintah ORDER BY kota ASC:

```
SELECT * FROM populasi WHERE kota BETWEEN 'Bekasi' AND 'Medan'
ORDER BY kota ASC;
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     4 | Bekasi    | 12    | 56    | 206.61  | 2381053  |
|     7 | Depok    | 11    | 63    | 200.29  | 1631951  |
|     1 | Jakarta   | 44    | 267   | 664.01  | 9988495  |
|     6 | Makassar   | 14    | 143   | 199.26  | 1651146  |
|     3 | Medan     | 21    | 151   | 265.00  | 2465469  |
+-----+-----+-----+-----+-----+
```

Operator BETWEEN juga cocok dipakai untuk kolom dengan tipe data DATE.

Sebagai contoh, saya ingin menampilkan tabel daftar_provinsi dimana tanggal diresmikan antara 09-08-1957 hingga 01-01-2017:

```
SELECT * FROM daftar_provinsi WHERE tanggal_diresmikan
BETWEEN '1957-08-09' AND '2017-01-01';
+-----+-----+-----+-----+
| prov        | ibukota    | luas    | tanggal_diresmikan |
+-----+-----+-----+-----+
| Sumatera Barat | Padang    | 42297  | 1957-08-09          |
| Sulawesi Selatan | Makassar  | 46116  | 1960-12-13          |
| Bali         | Denpasar   | 5561   | 1958-08-14          |
| Papua Barat  | Manokwari | 114566 | 1999-10-04          |
+-----+-----+-----+-----+
```

Kita harus membalik penulisan tanggal agar sesuai dengan format dari MySQL: YYYY-MM-DD.

Untuk proses penyeleksian yang sedikit rumit, operator BETWEEN juga bisa digabung dengan operator lain, misalnya dengan NOT IN seperti contoh berikut:

```
SELECT * FROM
  daftar_provinsi
WHERE
  (tanggal_diresmikan BETWEEN '1957-08-09' AND '2017-01-01')
AND
  ibukota NOT IN ('Denpasar', 'Jakarta', 'Medan');
+-----+-----+-----+
| prov      | ibukota | luas    | tanggal_diresmikan |
+-----+-----+-----+
| Sumatera Barat | Padang   | 42297  | 1957-08-09        |
| Sulawesi Selatan | Makassar | 46116  | 1960-12-13        |
| Papua Barat     | Manokwari | 114566 | 1999-10-04        |
+-----+-----+-----+
```

Query diatas bisa dibaca:

“Tampilkan isi tabel daftar_provinsi dimana tanggal diresmikan antara 09-08-1957 hingga 01-01-2017, serta tidak termasuk salah satu dari kota ‘Denpasar’, ‘Jakarta’, dan ‘Medan’”.

Saya harus menulis tanda kurung untuk menegaskan operasi mana yang harus di jalankan duluan, karena terdapat lebih dari 1 operator AND.

Atau yang (sedikit) lebih rumit lagi:

```
SELECT * FROM
  daftar_provinsi
WHERE
  (tanggal_diresmikan BETWEEN '1957-08-09' AND '2017-01-01')
  AND ibukota NOT IN (SELECT kota from populasi);
+-----+-----+-----+
| prov      | ibukota | luas    | tanggal_diresmikan |
+-----+-----+-----+
| Sumatera Barat | Padang   | 42297  | 1957-08-09        |
| Bali       | Denpasar | 5561   | 1958-08-14        |
| Papua Barat     | Manokwari | 114566 | 1999-10-04        |
+-----+-----+-----+
```

Query diatas bisa dibaca:

“Tampilkan isi tabel daftar_provinsi dimana tanggal diresmikan antara 09-08-1957 hingga 01-01-2017, serta tidak termasuk semua kota yang ada di dalam tabel populasi”.

13.12 SELECT ... WHERE ... LIKE

Query `SELECT...WHERE...LIKE` digunakan untuk proses pencarian dengan pola (*pattern*). Berikut format dasar penulisannya:

```
SELECT nama_kolom FROM nama_tabel WHERE nama_kolom LIKE pola
```

MySQL menyediakan 2 karakter khusus untuk pola pencarian `LIKE`, yakni karakter underscore (`_`) dan karakter persen (%). Berikut penjelasannya:

- `_` : karakter ganti yang cocok untuk satu karakter apa saja.
- `%` : karakter ganti yang cocok untuk karakter apa saja dengan panjang karakter tidak terbatas, termasuk tidak ada karakter.

Sebagai contoh, saya ingin menampilkan tabel populasi dimana nama kota diawali dengan huruf B:

```
SELECT * FROM populasi WHERE kota LIKE 'B%';
+-----+-----+-----+-----+-----+
| id_kota | kota     | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|      4 | Bekasi   |    12 |    56 | 206.61 | 2381053 |
|      5 | Bandung  |    30 |   151 | 167.67 | 2339463 |
+-----+-----+-----+-----+-----+
```

Tergantung character set dan jenis tipe data, proses pencarian ini bisa bersifat **case sensitif** (membedakan huruf besar dan kecil), atau **case insensitif** (tidak membedakan huruf besar dan kecil).

Karena tabel populasi menggunakan character set default `latin1`, hasil pencarian bersifat case insensitif (huruf besar dan kecil dianggap sama):

```
SELECT * FROM populasi WHERE kota LIKE 'b%';
+-----+-----+-----+-----+-----+
| id_kota | kota     | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|      4 | Bekasi   |    12 |    56 | 206.61 | 2381053 |
|      5 | Bandung  |    30 |   151 | 167.67 | 2339463 |
+-----+-----+-----+-----+-----+
```

Pola kota `LIKE 'b%'` akan cocok dengan kota yang diawali huruf b dan juga huruf B. Contoh lain, saya ingin menampilkan tabel `daftar_provinsi` yang nama provinsinya diawali string Sumatera:

```
SELECT * FROM daftar_provinsi WHERE prov LIKE 'Sumatera%';
+-----+-----+-----+-----+
| prov      | ibukota | luas   | tanggal_diresmikan |
+-----+-----+-----+-----+
| Sumatera Utara | Medan     | 72981 | 1956-11-29       |
| Sumatera Barat | Padang    | 42297 | 1957-08-09       |
| Sumatera Selatan | Palembang | 85679 | 1950-08-14       |
+-----+-----+-----+-----+
```

Atau bagaimana dengan menampilkan nama provinsi yang di dalamnya terdapat string at, tidak peduli dimanapun string tersebut berada:

```
SELECT * FROM daftar_provinsi WHERE prov LIKE '%at%';
+-----+-----+-----+-----+
| prov      | ibukota | luas   | tanggal_diresmikan |
+-----+-----+-----+-----+
| Sumatera Utara | Medan     | 72981 | 1956-11-29       |
| Sumatera Barat | Padang    | 42297 | 1957-08-09       |
| Jawa Barat      | Bandung   | 35245 | 1950-07-04       |
| Sulawesi Selatan | Makassar  | 46116 | 1960-12-13       |
| Sumatera Selatan | Palembang | 85679 | 1950-08-14       |
| Papua Barat      | Manokwari | 114566 | 1999-10-04       |
+-----+-----+-----+-----+
```

Pola prov LIKE '%at%' akan cocok dengan Sum-at-era, Bar-at, maupun Sel-at-an. Selama ada string at, hasilnya akan ditampilkan.

Penulisan pola LIKE seperti ini umum digunakan dalam pencarian, karena biasanya kita perlu menampilkan teks tanpa mempedulikan posisinya.

Selain menggunakan pola % yang mewakili banyak karakter, terdapat juga pola _ untuk mewakili 1 karakter saja. Berikut contoh penggunaannya:

```
SELECT * FROM populasi WHERE kota LIKE '_akart_';
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec | kel  | luas   | penduduk |
+-----+-----+-----+-----+-----+
| 1        | Jakarta   | 44  | 267  | 664.01 | 9988495 |
+-----+-----+-----+-----+-----+
```

Pola kota LIKE '_akart_' akan cocok dengan kota yang diawali 1 karakter apa saja, diikuti dengan string akart, kemudian diikuti dengan 1 karakter apa saja. Dalam contoh diatas, pola tersebut akan cocok dengan Jakarta. Pencarian seperti ini sangat spesifik, karena hanya memperbolehkan 1 karakter pengganti.

Sebagai tambahan, berikut beberapa contoh pola dan hasil string yang cocok dengan pola tersebut:

- S% : Cocok dengan kata yang diawali dengan huruf S atau huruf s yang diikuti dengan karakter apa saja, contoh: S, Sa, Si, Saaaaaa, Susi, dan Sabrina Sari.
- S_ : Cocok dengan kata yang diawali dengan S dan diikuti dengan satu karakter apa saja, contoh: Si, Sa, Su, Se, dan St.
- A__i: Cocok dengan kata yang diawali dengan A, diikuti oleh 2 karakter bebas, namun diakhiri dengan huruf i, contoh: Andi, ardi, aaai.
- %e: Cocok dengan seluruh kata dengan panjang berapapun yang diakhiri dengan huruf e, contoh: Kece, Kue, dan mie.
- %dia%: Cocok dengan seluruh kata yang mengandung kata dia, contoh: media, kemudian, dia, dan diaaaa.

Untuk menguji pola LIKE, kita bisa menulis query SELECT sebagai berikut:

```
SELECT 'Saaaa' LIKE 'S%';
+-----+
| 'Saaaa' LIKE 'S%' |
+-----+
|          1 |
+-----+  
  

SELECT 'Belajar' LIKE 'S%';
+-----+
| 'Belajar' LIKE 'S%' |
+-----+
|          0 |
+-----+  
  

SELECT 'kemudian' LIKE '%dia%';
+-----+
| 'kemudian' LIKE '%dia%' |
+-----+
|          1 |
+-----+  
  

SELECT 'Bandung' LIKE '%du%';
+-----+
| 'Bandung' LIKE '%du%' |
+-----+
|          1 |
+-----+
```

Disini perintah LIKE berperan sebagai operator. Jika hasilnya 1, maka string yang diuji sesuai dengan pola. Jika hasilnya 0, string tidak sesuai dengan pola.

13.13 SELECT ... WHERE ... REGEXP

Untuk proses pencarian yang lebih rumit, MySQL mendukung pola **Regular Expression**. *Regular Expression* (sering juga disingkat sebagai **RexExp** atau hanya **RE**) adalah kumpulan huruf atau karakter yang digunakan untuk proses pencocokan pola (*pattern matching*). Regular Expression mirip seperti perintah **LIKE** tapi lebih powerfull.

Jika sebelumnya anda sudah mempelajari bahasa pemrograman PHP dan JavaScript (misalnya dari buku PHP Uncover dan JavaScript Uncover), tentunya tidak asing dengan Regular Expression. Dalam kedua buku tersebut saya membahas cukup detail tentang regular expression.

Pola karakter RegExp di MySQL mirip seperti yang ada di PHP dan JavaScript, karena regular expression ini bersifat umum dan biasanya tersedia di hampir semua bahasa pemrograman modern.

Berikut format dasar penulisan RegExp di MySQL:

```
SELECT nama_kolom FROM nama_tabel WHERE nama_kolom REGEXP pola_regexp
SELECT nama_kolom FROM nama_tabel WHERE nama_kolom RLIKE pola_regexp
```

Terdapat 2 perintah untuk menjalankan regular expression, yakni **REGEXP** dan **RLIKE**, keduanya bermakna sama.

Regular expression memiliki aturan penulisan pola yang cukup kompleks, dan kita akan bahas secara bertahap.

Pola RegExp Sebagai String

Dalam bentuk paling sederhana, pola RegExp bisa ditulis sebagai sebuah string atau kata biasa. MySQL akan menampilkan hasil selama kata tersebut ditemukan, tidak peduli dimana posisinya. Sebagai contoh, saya ingin menampilkan tabel *populasi* untuk nama kota yang memiliki string **Ba**:

```
SELECT * FROM populasi WHERE kota REGEXP 'Ba';
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel    | luas     | penduduk |
+-----+-----+-----+-----+-----+
|      2 | Surabaya  |   31 | 154   | 350.54  | 2805906  |
|      5 | Bandung   |   30 | 151   | 167.67  | 2339463  |
|     10 | Palembang |   14 | 107   | 369.22  | 1548064  |
+-----+-----+-----+-----+-----+
```

Atau contoh lain, saya ingin menampilkan tabel *daftar_provinsi* dimana nama provinsi memiliki kata Sumatera:

```
SELECT * FROM daftar_provinsi WHERE prov REGEXP 'sumatera';
+-----+-----+-----+-----+
| prov      | ibukota    | luas     | tanggal_diresmikan |
+-----+-----+-----+-----+
| Sumatera Utara | Medan       | 72981 | 1956-11-29        |
| Sumatera Barat | Padang       | 42297 | 1957-08-09        |
| Sumatera Selatan | Palembang | 85679 | 1950-08-14        |
+-----+-----+-----+-----+
```

Dari dua contoh diatas terlihat bahwa MySQL tidak membedakan huruf besar dan kecil untuk regular expression. Alasannya sama seperti pada query LIKE, dimana hal ini dipengaruhi oleh character set dan collation yang dipakai, serta apakah kolom tersebut bertipe *binary* atau tidak.

Menandakan Awal dan Akhir Pola

Untuk mengatur penempatan pola, misalnya harus berada di awal atau di akhir, regular expression menyediakan 2 karakter khusus: ^ dan \$. Berikut penjelasannya:

- ^ = Sebagai karakter penanda awal pola. Jika regular expression ditambahkan karakter ini, pola tersebut harus berada di awal string.
- \$ = Sebagai karakter penanda akhir pola. Jika regular expression ditambahkan karakter ini, pola tersebut harus berada di akhir string.

Sebagai contoh, saya ingin menampilkan tabel populasi dimana nama kota diawali dengan huruf B:

```
SELECT * FROM populasi WHERE kota REGEXP '^B';
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec     | kel     | luas     | penduduk |
+-----+-----+-----+-----+-----+
| 4 | Bekasi    | 12 | 56 | 206.61 | 2381053 |
| 5 | Bandung   | 30 | 151 | 167.67 | 2339463 |
+-----+-----+-----+-----+-----+
```

Contoh lain, saya ingin menampilkan tabel daftar_provinsi dimana nama ibukota diakhiri dengan huruf g:

```
SELECT * FROM daftar_provinsi WHERE ibukota REGEXP 'g$';
+-----+-----+-----+-----+
| prov      | ibukota | luas   | tanggal_diresmikan |
+-----+-----+-----+-----+
| Sumatera Barat | Padang    | 42297 | 1957-08-09      |
| Jawa Barat     | Bandung   | 35245 | 1950-07-04      |
| Jawa Tengah     | Semarang   | 33987 | 1950-07-04      |
| Sumatera Selatan | Palembang | 85679 | 1950-08-14      |
+-----+-----+-----+-----+
```

Wildcard

Wildcard adalah sebutan untuk karakter yang bisa mewakili karakter apa saja. Di dalam regular expression, karakter *wildcard* ditulis dengan tanda titik (.).

Penggunaan karakter *wildcard* mirip seperti karakter underscore (_) dalam perintah LIKE, yakni mewakili 1 karakter apa saja.

Sebagai contoh, saya ingin menampilkan tabel populasi dimana kolom kecamatan hanya terdiri dari 2 digit:

```
SELECT * FROM populasi WHERE kel REGEXP '^..$';
+-----+-----+-----+-----+-----+
| id_kota | kota    | kec   | kel    | luas   | penduduk |
+-----+-----+-----+-----+-----+
|       4 | Bekasi  | 12    | 56    | 206.61 | 2381053 |
|       7 | Depok   | 11    | 63    | 200.29 | 1631951 |
+-----+-----+-----+-----+-----+
```

Untuk contoh ini, saya harus menambahkan tanda ^ dan \$ sebagai penanda awal dan akhir pola. Karena jika tidak, jumlah kecamatan dengan 3 digit juga akan ditampilkan.

Ini terjadi karena pola regular expression akan menampilkan data meskipun hanya sebagian dari data tersebut yang sesuai dengan pola. Dengan menulis '^..\$' artinya dua digit tersebut harus berada di awal dan akhir, tidak boleh ada digit lain.

Kumpulan Karakter

Kita bisa membuat pola regular expression yang terdiri dari kumpulan karakter. Karakter-karakter ini ditulis di dalam tanda kurung siku ([]).

Sebagai contoh, saya ingin menampilkan tabel populasi dimana nama kota memiliki setidaknya satu huruf s dan diikuti oleh huruf vokal (a, i, u, e dan o):

```
SELECT * FROM populasi WHERE kota REGEXP 's[aiueo]';
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     2 | Surabaya |    31 |   154 | 350.54 | 2805906 |
|     4 | Bekasi    |    12 |    56 | 206.61 | 2381053 |
|     6 | Makassar  |    14 |   143 | 199.26 | 1651146 |
|     8 | Semarang  |    16 |   177 | 373.78 | 1621384 |
+-----+-----+-----+-----+-----+
```

Setiap kota yang tampil memiliki setidaknya salah satu dari string berikut: sa, si, su, se dan so. Pola kumpulan karakter ini juga bisa berbentuk jangkauan, seperti [a-e] yang sama artinya dengan [abcde]:

```
SELECT * FROM populasi WHERE kota REGEXP 's[a-e]';
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     6 | Makassar |    14 |   143 | 199.26 | 1651146 |
|     8 | Semarang |    16 |   177 | 373.78 | 1621384 |
+-----+-----+-----+-----+-----+
```

Pola 's[a-e]' cocok dengan kota Makassar dan Semarang karena terdapat karakter sa dan se.

Negasi Kumpulan Karakter

Negasi kumpulan karakter digunakan untuk membalik logika dari kumpulan karakter. Untuk membuat pola ini, tempatkan karakter ^ di dalam tanda kurung siku. Berikut contoh penggunaannya:

```
SELECT * FROM populasi WHERE kota REGEXP 's[^aiueo]';
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     6 | Makassar |    14 |   143 | 199.26 | 1651146 |
+-----+-----+-----+-----+-----+
```

Pola 's[^aiueo]' akan cocok dengan huruf s yang diikuti oleh **selain** huruf a, i, u, e dan o. Artinya, yang akan cocok dengan pola ini adalah: sb, sc, sd, dst.. tapi tidak dengan sa, si, su, se dan so.

Kota Makassar memiliki karakter ss yang cocok dengan syarat pola 's[^aiueo]'.

Yang perlu diperhatikan, tanda ^ memiliki 2 fungsi di dalam regular expression. Jika anda masih ingat, tanda ^ juga berfungsi untuk menandakan awal pola. Namun khusus jika ditempatkan di dalam tanda kurung siku, fungsinya berubah sebagai penanda negasi.

Sebagai contoh, bisakah anda membedakan kedua pola berikut?

- '^J[abcd]'
- 'J[^abcd]'

Pola '^J[abcd]' akan cocok dengan string yang diawali dengan huruf J, kemudian diikuti oleh satu dari huruf a, b, c dan d. String Jakarta akan cocok dengan pola ini.

'J[^abcd]' akan cocok dengan string yang memiliki huruf J dan diikuti oleh karakter **selain** huruf a, b, c dan d. Pola ini tidak harus ada di awal string, misalnya Mojokerto, dimana terdapat jo. Namun tidak akan cocok dengan Jakarta.

Menggunakan perintah REGEXP sebagai operator, kita bisa mengujinya tanpa melibatkan tabel MySQL:

```
SELECT 'Jakarta' REGEXP '^J[abcd]';
+-----+
| 'Jakarta' REGEXP '^J[abcd]' |
+-----+
|                         1 |
+-----+

SELECT 'Mojokerto' REGEXP 'J[^abcd]';
+-----+
| 'Mojokerto' REGEXP 'J[^abcd]' |
+-----+
|                         1 |
+-----+

SELECT 'Jambi' REGEXP 'J[^abcd]';
+-----+
| 'Jambi' REGEXP 'J[^abcd]' |
+-----+
|                         0 |
+-----+
```

Sama seperti query LIKE, jika hasilnya 1 maka string sesuai dengan pola. Jika hasilnya 0, maka string tidak cocok dengan pola regular expression.

Pola Logika OR

Pola Logika OR digunakan untuk membuat kondisi logika “atau” yang berfungsi sebagai alternatif pilihan pola. Untuk membuat logika OR ini, digunakan karakter pipe (|).

Sebagai contoh, saya ingin menampilkan tabel populasi dimana nama kota berisi string ta, atau ya atau ma:

```
SELECT * FROM populasi WHERE kota REGEXP 'ta|ya|ma';
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     1   | Jakarta   | 44    | 267   | 664.01  | 9988495  |
|     2   | Surabaya  | 31    | 154   | 350.54  | 2805906  |
|     6   | Makassar  | 14    | 143   | 199.26  | 1651146  |
|     8   | Semarang  | 16    | 177   | 373.78  | 1621384  |
|     9   | Tangerang | 13    | 104   | 153.93  | 1566190  |
+-----+-----+-----+-----+-----+
```

Untuk setiap kota yang tampil, setidaknya akan memiliki salah satu dari string ta, ya atau ma.

Contoh lain, saya ingin menampilkan tabel daftar_provinsi dimana nama provinsi berisi string ara atau esi:

```
SELECT * FROM daftar_provinsi WHERE prov REGEXP 'ara|esi';
+-----+-----+-----+-----+
| prov        | ibukota    | luas    | tanggal_diresmikan |
+-----+-----+-----+-----+
| Sumatera Utara | Medan      | 72981  | 1956-11-29       |
| Sumatera Barat | Padang     | 42297  | 1957-08-09       |
| Jawa Barat    | Bandung    | 35245  | 1950-07-04       |
| Sulawesi Selatan | Makassar  | 46116  | 1960-12-13       |
| Papua Barat   | Manokwari  | 114566 | 1999-10-04       |
+-----+-----+-----+-----+
```

Membatasi Jumlah Karakter

Regular expression juga mengizinkan kita membatasi jumlah karakter tertentu, misalnya huruf a harus ditulis 2 kali, atau digit angka harus tertulis sebanyak 8 - 12 kali (untuk nomor HP).

Karakter yang digunakan untuk membatasi jumlah ini adalah tanda kurung kurawal { dan }.

Sebagai contoh, saya ingin menampilkan tabel populasi dimana nama kota setidaknya memiliki 2 buah huruf s:

```
SELECT * FROM populasi WHERE kota REGEXP 's{2}';
+-----+-----+-----+-----+-----+
| id_kota | kota      | kec   | kel   | luas    | penduduk |
+-----+-----+-----+-----+-----+
|     6   | Makassar  | 14    | 143   | 199.26  | 1651146  |
+-----+-----+-----+-----+-----+
```

Contoh lain, pola 'a{3}9{2}' akan cocok dengan string yang di dalamnya ada huruf aaa99. Atau pola '.{3}z{2}' akan cocok dengan string yang diawali dengan 3 digit karakter apa saja, kemudian diikuti oleh dua buah huruf z, seperti abczz, aa123zz maupun zzzzz.

Berikut percobaannya:

```
SELECT 'aaa99' REGEXP 'a{3}9{2}' ;
+-----+
| 'aaa99' REGEXP 'a{3}9{2}' |
+-----+
|                         1 |
+-----+
```



```
SELECT 'aa123zz' REGEXP '.{3}z{2}' ;
+-----+
| 'aa123zz' REGEXP '.{3}z{2}' |
+-----+
|                         1 |
+-----+
```

Selain membatasi jumlah karakter, kita juga bisa membatasi *jangkauan* jumlah karakter.

Caranya dengan membuat 2 angka di dalam kurung kurawal. Angka pertama menentukan jumlah minimum, sedangkan angka kedua untuk menentukan nilai maksimum. Pola 'a{2,4}' artinya huruf a boleh ditulis 2 kali dan maksimal 4 kali.

Jika digit kedua tidak ditulis itu artinya digit maksimal tidak dibatasi. Misalnya pola 'a{2,}' akan cocok dengan string yang di dalamnya minimal ada 2 huruf a.

```
SELECT 'andikaa' REGEXP 'a{2,4}' ;
+-----+
| 'andikaa' REGEXP 'a{2,4}' |
+-----+
|                         1 |
+-----+
```



```
SELECT 'andika' REGEXP 'a{2,}' ;
+-----+
| 'andika' REGEXP 'a{2,}' |
+-----+
|                         0 |
+-----+
```



```
SELECT 'andikaa' REGEXP 'a{2,}' ;
+-----+
| 'andikaa' REGEXP 'a{2,}' |
+-----+
|                         1 |
+-----+
```

Selain menulis manual jumlah karakter, regular expression juga menyediakan beberapa karakter khusus untuk membatasi pola:

- * (tanda bintang), sama fungsinya dengan $\{0, \infty\}$. Artinya cocok dengan 0 atau lebih karakter (tidak dibatasi).
- + (tanda tambah), sama fungsinya dengan $\{1, \infty\}$. Artinya cocok dengan 1 karakter atau lebih (tidak dibatasi).
- ? (tanda tanya), sama fungsinya dengan $\{0, 1\}$. Artinya cocok dengan 0 atau 1 karakter (tidak boleh lebih).

Sebagai contoh, pola 'ab*c' akan cocok dengan string yang diawali huruf a, kemudian diikuti tanpa atau beberapa huruf b, dan diakhiri dengan 1 huruf c.

Pola 'ab+c' akan cocok dengan string yang diawali huruf a, kemudian diikuti minimal satu huruf b, dan diakhiri dengan 1 huruf c.

Berikut percobaannya:

```
SELECT 'abbc' REGEXP 'ab*c';
```

'abbc'	REGEXP 'ab*c'
+-----+	
	1
+-----+	

```
SELECT 'ac' REGEXP 'ab*c';
```

'ac'	REGEXP 'ab*c'
+-----+	
	1
+-----+	

```
SELECT 'abbc' REGEXP 'ab+c';
```

'abbc'	REGEXP 'ab+c'
+-----+	
	1
+-----+	

```
SELECT 'ac' REGEXP 'ab+c';
```

'ac'	REGEXP 'ab+c'
+-----+	
	0
+-----+	

Pola Karakter Khusus

Untuk pola yang sering digunakan seperti digit angka 0-9, huruf abjad A-Z dan a-z, atau karakter alfanumerik (huruf dan angka), MySQL memiliki karakter khusus sebagai pengganti pola-pola ini:

Pola Karakter	Penjelasan pola
alnum	Karakter alphanumeric (seluruh huruf dan angka)
alpha	Karakter alphabetic (seluruh huruf)
lower	Karakter lowercase alphabetic (huruf kecil)
upper	Karakter uppercase alphabetic (huruf besar)
digit	Karakter digit (seluruh angka)
punct	Karakter punctuation characters (pembatas kata, seperti ,!?:)
blank	Karakter whitespace (spasi, tab, enter, dll)
space	Karakter spasi, tab, newline, dan carriage return
xdigit	Karakter hexadecimal (semua angka + huruf a-f)
cntrl	Karakter control
graph	Karakter graphic
print	Karakter graphic atau spasi

Untuk menggunakan pola karakter diatas, harus ditulis dalam 2 buah tanda kurung siku dan tanda titik dua, yakni kurung pembuka [: dan kurung penutup :].

Sebagai contoh, pola ‘[:alpha:]’ akan mewakili 1 digit huruf. Pola [:alpha:][:space:][:digit:] akan mewakili 1 huruf, 1 spasi dan 1 angka, yang akan cocok dengan string ‘a 9’ atau belajar 1.

Berikut beberapa contoh penggunaannya:

```
SELECT 'a 9' REGEXP '[:alpha:][:space:][:digit:]';
+-----+
| 'a 9' REGEXP '[:alpha:][:space:][:digit:]' |
+-----+
|                               1 |
+-----+


SELECT 'belajar 9' REGEXP '[:alpha:][:space:][:digit:]';
+-----+
| 'belajar 9' REGEXP '[:alpha:][:space:][:digit:]' |
+-----+
|                               1 |
+-----+


SELECT 'belajar9' REGEXP '[:alpha:][:space:][:digit:]';
+-----+
| 'belajar9' REGEXP '[:alpha:][:space:][:digit:]' |
+-----+
|                               0 |
+-----+


SELECT 'belajar 9' REGEXP '^[:alpha:][:space:][:digit:]';
+-----+
| 'belajar 9' REGEXP '^[:alpha:][:space:][:digit:]' |
```

```
+-----+
|          0 |
+-----+
SELECT 'B 9120' REGEXP '^[[[:alpha:]][[:space:]][[:digit:]]';
+-----+
| 'B 9120' REGEXP '^[[[:alpha:]][[:space:]][[:digit:]]' |
+-----+
|          1 |
+-----+
```

Huff... terasa cukup rumit? Memang, saya harus mengakui bahwa regular expression bisa sangat kompleks. Apa yang kita coba disini baru materi dasar dari regular expression. Tapi setidaknya anda mengetahui bahwa regular expression juga bisa digunakan untuk proses pencarian di MySQL.

Dalam kebanyakan kasus, kita tidak perlu menggunakan REGEXP dan cukup dengan query LIKE. Regular expression baru pas dipakai untuk pencarian yang sangat spesifik. Selain itu dibutuhkan proses yang lebih lama untuk REGEXP dibandingkan LIKE.

13.14 SELECT ... UNION

Query SELECT...UNION bisa dipakai untuk menggabungkan hasil dua atau lebih query SELECT. Berikut format dasar penulisannya:

```
SELECT nama_kolom1, [nama_kolom2] FROM nama_tabel1
UNION
SELECT nama_kolom3, [nama_kolom4] FROM nama_tabel2
```

Syarat utama dari penggabungan ini adalah jumlah kolom di query SELECT pertama harus sama dengan jumlah kolom query SELECT kedua. Jika tidak, akan keluar pesan error.

Sebagai contoh saya ingin menggabungkan kolom kota dari tabel populasi dengan kolom ibukota dari tabel daftar_provinsi:

```
SELECT kota FROM populasi
UNION
SELECT ibukota FROM daftar_provinsi;
+-----+
| kota      |
+-----+
| Jakarta   |
| Surabaya  |
| Medan     |
| Bekasi    |
```

```

| Bandung    |
| Makassar   |
| Depok      |
| Semarang   |
| Tangerang  |
| Palembang  |
| Padang     |
| Denpasar   |
| Manokwari  |
+-----+
13 rows in set (0.09 sec)

```

Karena jumlah kolom dari kedua query SELECT hanya 1 buah, hasilnya nama-nama kota akan digabung ke dalam 1 kolom.

Nama kolom hasil UNION diambil dari nama kolom query SELECT pertama. Kita juga bisa menggunakan query AS untuk mengganti nama kolom ini :

```

SELECT kota AS 'Nama Kota' FROM populasi
UNION
SELECT ibukota FROM daftar_provinsi;
+-----+
| Nama Kota |
+-----+
| Jakarta   |
| Surabaya  |
| Medan     |
| ...       |
13 rows in set (0.09 sec)
+-----+

```

Hasil gabungan tabel terdapat 13 nama kota. Tapi bukankah tabel populasi terdiri dari 10 kota dan tabel daftar_provinsi terdiri dari 8 ibukota? Seharusnya total terdapat 18 nama kota, kemana sisanya?

Secara bawaan, jika di dalam hasil query UNION terdapat nilai yang sama, data tersebut hanya ditampilkan 1 kali saja.

Dari tabel populasi dan daftar_kota, terdapat 5 kota yang sama, yakni: Medan, Bandung, Makassar, Semarang dan Palembang. Kelima kota ini akan ditampilkan 1 kali saja, sehingga $18 - 5 = 13$ kota.

Jika kita ingin untuk tetap menampilkan kota yang sama, bisa menggunakan query UNION ALL:

```
SELECT kota FROM populasi
UNION ALL
SELECT ibukota FROM daftar_provinsi;
+-----+
| kota      |
+-----+
| Jakarta   |
| Surabaya  |
| Medan     |
| Bekasi    |
| Bandung   |
| Makassar  |
| Depok     |
| Semarang  |
| Tangerang |
| Palembang|
| Medan     |
| Padang    |
| Bandung   |
| Semarang  |
| Makassar  |
| Denpasar  |
| Palembang|
| Manokwari |
+-----+
18 rows in set (0.00 sec)
```

Sekarang, seluruh data dari kedua kolom akan ditampilkan, meskipun terdapat kota yang berulang.

Kita juga bisa menggunakan query UNION DISTINCT untuk secara tertulis memerintahkan MySQL agar tidak menampilkan data yang sama, meskipun sebagaimana yang kita lihat sebelumnya, ini merupakan prilaku bawaan:

```
SELECT kota FROM populasi
UNION DISTINCT
SELECT ibukota FROM daftar_provinsi;
+-----+
| kota      |
+-----+
| Jakarta   |
| Surabaya  |
| Medan     |
| ...       |
+-----+
13 rows in set (0.00 sec)
```

Secara default, kolom hasil query UNION akan diurutkan mulai dari kolom tabel pertama, kemudian dibawahnya akan diikuti data dari tabel kedua. Kita bisa menambahkan perintah ORDER BY agar tampilan datanya bisa berurutan:

```
SELECT kota FROM populasi
UNION
SELECT ibukota FROM daftar_provinsi
ORDER BY kota ASC;
+-----+
| kota      |
+-----+
| Bandung   |
| Bekasi    |
| Denpasar  |
| ...       |
| Semarang  |
| Surabaya  |
| Tangerang|
+-----+
```

Berbagai query yang telah kita pelajari hingga saat ini juga bisa diterapkan di masing-masing query SELECT, seperti contoh berikut:

```
(SELECT kota FROM populasi ORDER BY kota ASC LIMIT 5)
UNION
(SELECT ibukota FROM daftar_provinsi WHERE ibukota LIKE 'P%')
ORDER BY kota ASC;
+-----+
| kota      |
+-----+
| Bandung   |
| Bekasi    |
| Depok     |
| Jakarta   |
| Makassar  |
| Padang    |
| Palembang|
+-----+
7 rows in set (0.00 sec)
```

Disini saya menggabungkan 5 kota teratas dari tabel populasi yang diurutkan berdasarkan abjad dengan kolom ibukota dari tabel daftar_provinsi yang berawalan huruf P.

Tanda kurung diperlukan agar penulisan setiap query SELECT bisa saling terpisah. Jika tanda kurung ini dihapus, MySQL akan mengeluarkan error karena “bingung” perintah mana untuk query yang mana.

Dalam contoh sebelum ini saya hanya menggabungkan 1 kolom, namun tentu saja kita bisa menggabungkan 2 atau 3 kolom, misalnya:

```
SELECT kota, luas FROM populasi
UNION
SELECT prov, luas FROM daftar_provinsi;
+-----+-----+
| kota      | luas    |
+-----+-----+
| Jakarta   | 664.01  |
| Surabaya  | 350.54  |
| Medan     | 265.00  |
| ...        | ...     |
| Bali      | 5561.00 |
| Sumatera Selatan | 85679.00 |
| Papua Barat | 114566.00 |
+-----+
18 rows in set (0.04 sec)
```

Disini saya menggabungkan kolom kota dan luas dari tabel populasi, dengan kolom prov dan luas dari tabel daftar_kota.

Yang cukup unik, meskipun tidak lazim, MySQL membolehkan penggabungan kolom meskipun tidak bertipe data sama. Berikut contohnya:

```
SELECT kota, luas FROM populasi
UNION
SELECT luas, tanggal_diresmikan FROM daftar_provinsi;
+-----+-----+
| kota      | luas    |
+-----+-----+
| Jakarta   | 664.01  |
| Surabaya  | 350.54  |
| Medan     | 265.00  |
| Bekasi    | 206.61  |
| ...        | ...     |
| 85679     | 1950-08-14 |
| 114566    | 1999-10-04 |
+-----+
18 rows in set (0.00 sec)
```

Dalam query diatas, saya menggabungkan kolom kota yang bertipe VARCHAR, dengan kolom luas yang bertipe INT. Di sebelahnya, kolom luas yang bertipe DECIMAL digabung dengan kolom tanggal_diresmikan yang bertipe DATE.

Tipe data kolom hasil gabungan akan ditentukan dari kedua tipe data asal, selama tipe data hasil gabungan bisa menampung semua nilai yang ada. Misalnya gabungan tipe data INT dengan

DECIMAL akan menggunakan kolom DECIMAL karena angka integer tetap bisa disimpan ke dalam kolom DECIMAL.

Atau jika kedua tipe data tidak ada yang cocok, kolom akan menggunakan tipe data string (CHAR/VARCHAR). Misalnya gabungan kolom DECIMAL dengan DATE yang tidak bisa ditampung oleh salah satu tipe data seperti contoh kita diatas.

Hampir setiap kolom bisa digabung, selama jumlah kolomnya sama. Dalam contoh berikut ini saya menggabungkan 4 kolom dari setiap tabel:

```
SELECT kota, luas, kec, penduduk FROM populasi
UNION
SELECT * FROM daftar_provinsi;
+-----+-----+-----+-----+
| kota      | luas     | kec      | penduduk |
+-----+-----+-----+-----+
| Jakarta   | 664.01   | 44       | 9988495  |
| Surabaya  | 350.54   | 31       | 2805906  |
| Medan     | 265.00   | 21       | 2465469  |
| ...        | ....     | ..       | ...       |
| Bali       | Denpasar | 5561    | 1958-08-14 |
| Sumatera Selatan | Palembang | 85679 | 1950-08-14 |
| Papua Barat | Manokwari | 114566 | 1999-10-04 |
+-----+-----+-----+-----+
18 rows in set (0.00 sec)
```

Query SELECT...UNION cocok digunakan untuk membuat laporan, terutama jika data tersebar ke dalam beberapa tabel.

13.15 SELECT DISTINCT

Perintah DISTINCT digunakan untuk menghapus data yang sama dari hasil tampilan query SELECT.

Berikut format dasar penulisan query ini:

```
SELECT DISTINCT nama_kolom FROM nama_tabel
```

Membuat tabel mahasiswa

Sebagai tabel sample untuk pembahasan query DISTINCT dan juga untuk materi selanjutnya, saya akan membuat tabel `mahasiswa`. Berikut perintah pembuatan tabel `mahasiswa`:

```
CREATE TABLE mahasiswa (
    nama VARCHAR(50),
    asal VARCHAR(50),
    kel ENUM('L','P'),
    tinggi TINYINT UNSIGNED,
    jurusan VARCHAR (20),
    nilai_uan DEC(5,2)
);
Query OK, 0 rows affected (0.24 sec)

INSERT INTO mahasiswa VALUES
('Riana Putria', 'Padang', 'P', 155, 'Kimia', 339.20),
('Rudi Permana', 'Bandung', 'L', 163, 'Ilmu Komputer', 290.44),
('Sari Citra Lestari', 'Jakarta', 'P', 161, 'Manajemen', 310.60),
('Rina Kumala Sari', 'Jakarta', 'P', 158, 'Akuntansi', 337.99),
('James Situmorang', 'Medan', 'L', 168, 'Kedokteran Gigi', 341.10),
('Sandri Fatmala', 'Bandung', 'P', 165, 'Ilmu Komputer', 322.91),
('Husli Khairan', 'Jakarta', 'L', 170, 'Akuntansi', 288.55),
('Christine Wijaya', 'Medan', 'P', 157, 'Manajemen', 321.74),
('Ikhsan Prayoga', 'Jakarta', 'L', 172, 'Ilmu Komputer', 300.16),
('Bobby Permana', 'Medan', 'L', 161, 'Ilmu Komputer', 280.82);
Query OK, 10 rows affected (0.07 sec)
Records: 10 Duplicates: 0 Warnings: 0
```

```
SELECT * FROM mahasiswa;
```

nama	asal	kel	tinggi	jurusan	nilai_uan
Riana Putria	Padang	P	155	Kimia	339.20
Rudi Permana	Bandung	L	163	Ilmu Komputer	290.44
Sari Citra Lestari	Jakarta	P	161	Manajemen	310.60
Rina Kumala Sari	Jakarta	P	158	Akuntansi	337.99
James Situmorang	Medan	L	168	Kedokteran Gigi	341.10
Sandri Fatmala	Bandung	P	165	Ilmu Komputer	322.91
Husli Khairan	Jakarta	L	170	Akuntansi	288.55
Christine Wijaya	Medan	P	157	Manajemen	321.74
Ikhsan Prayoga	Jakarta	L	172	Ilmu Komputer	300.16
Bobby Permana	Medan	L	161	Ilmu Komputer	280.82

Tabel `mahasiswa` ini saya rancang untuk menampung 6 kolom: nama mahasiswa, kota asal, jenis kelamin, tinggi badan, jurusan serta nilai uan dari setiap mahasiswa. Karena terdapat kolom nilai uan, anda bisa menebak bahwa tabel ini berisi data mahasiswa baru.

Silahkan pelajari sejenak struktur serta isi dari tabel `mahasiswa` diatas, karena kita akan mulai menyeleksi tampilan dari tabel ini.

Baik, bagaimana jika saya ingin menampilkan daftar seluruh kota asal mahasiswa? Tidak ada masalah, querinya juga cukup sederhana:

```
SELECT asal FROM mahasiswa ORDER BY asal;  
+-----+  
| asal |  
+-----+  
| Bandung |  
| Bandung |  
| Jakarta |  
| Medan |  
| Medan |  
| Padang |  
+-----+
```

Query diatas menampilkan data sebagaimana mestinya. Karena terdapat 10 mahasiswa, juga akan tampil 10 nama kota.

Akan tetapi yang saya inginkan sebenarnya hanya daftar nama kota, kota yang sama tidak perlu ditampilkan secara berulang tapi cukup sekali saja. Dalam situasi inilah query DISTINCT bisa digunakan:

```
SELECT DISTINCT asal FROM mahasiswa ORDER BY asal;  
+-----+  
| asal |  
+-----+  
| Bandung |  
| Jakarta |  
| Medan |  
| Padang |  
+-----+
```

Sekarang, tiap data cukup ditampilkan 1 kali. Contoh lain, saya ingin melihat apa saja nama jurusan yang tersedia:

```
SELECT DISTINCT jurusan FROM mahasiswa ORDER BY jurusan;
+-----+
| jurusan |
+-----+
| Akuntansi |
| Ilmu Komputer |
| Kedokteran Gigi |
| Kimia |
| Manajemen |
+-----+
```

Tanpa menggunakan query DISTINCT, hasil diatas akan menampilkan 10 nama jurusan dari pilihan setiap mahasiswa, yang akan menampilkan jurusan yang sama beberapa kali.

Query DISTINCT hanya akan mengeliminasi data berulang di dalam satu baris. Jika data yang ditampilkan terdiri dari 2 kolom, penentuan data yang sama akan melibatkan kedua kolom tersebut.

Berikut contoh kasusnya:

```
SELECT DISTINCT asal, jurusan FROM mahasiswa;
+-----+-----+
| asal | jurusan |
+-----+-----+
| Padang | Kimia |
| Bandung | Ilmu Komputer |
| Jakarta | Manajemen |
| Jakarta | Akuntansi |
| Medan | Kedokteran Gigi |
| Medan | Manajemen |
| Jakarta | Ilmu Komputer |
| Medan | Ilmu Komputer |
+-----+-----+
8 rows in set (0.00 sec)
```

Meskipun saya sudah menggunakan query DISTINCT, di kolom asal tetap tampil 3 kali nama kota Jakarta dan Medan. Alasannya, jika dikombinasikan dengan kolom jurusan, setiap baris tetap unik, yakni tidak ada baris yang memiliki nilai kota asal dan jurusan yang sama ditampilkan lebih dari 1 kali.

Query DISTINCT cocok digunakan untuk membuat data rangkuman, dimana kita hanya perlu informasi global tanpa butuh detail setiap data.

13.16 SELECT COUNT(), MAX(), MIN(), AVG() dan SUM()

Perintah COUNT(), MAX(), MIN(), AVG() dan SUM() sebenarnya merupakan *function* bawaan MySQL.

Kita akan membahas function bawaan MySQL dalam 1 bab khusus, namun karena penggunaan ke-5 function ini sangat erat dengan query `SELECT...GROUP BY` (yang akan kita bahas setelah ini), saya butuh membahasnya terlebih dahulu.

Dalam bahasa pemrograman, **function** atau **fungsi** adalah perintah khusus yang menghasilkan nilai tertentu. Umumnya setiap function butuh data inputan sebagai nilai awal yang akan diproses oleh function tersebut. Function bisa dibuat sendiri, atau tersedia secara bawaan.

Di dalam MySQL, function `COUNT()`, `MAX()`, `MIN()`, `AVG()` dan `SUM()` termasuk ke dalam jenis **aggregate function**, yakni function yang nilai inputannya butuh berbagai data (sekumpulan data), dan mengembalikan satu nilai akhir.

Berikut penjelasan dari masing-masing function:

- `COUNT()`: Menghitung jumlah baris data.
- `MAX()`: Menampilkan nilai tertinggi dari kumpulan data.
- `MIN()`: Menampilkan nilai terendah dari kumpulan data.
- `AVG()`: Menampilkan nilai rata-rata dari kumpulan data.
- `SUM()`: Menampilkan nilai total (penjumlahan) dari kumpulan data.

Kita akan bahas function-function ini dengan berbagai contoh.

Pertama, jika saya ingin mencari informasi berapa jumlah baris yang ada di dalam tabel, bisa menggunakan function `COUNT()`. Cara penggunaannya adalah sebagai berikut:

```
SELECT COUNT(*) FROM mahasiswa;
+-----+
| COUNT(*) |
+-----+
|      10 |
+-----+
```

Dalam query diatas, saya men-input tanda bintang ke dalam function `COUNT()`, yang artinya function ini akan menghitung jumlah baris untuk seluruh kolom. Karena setiap kolom memiliki jumlah baris yang sama, kita juga bisa menggantinya dengan salah satu nama kolom:

```
SELECT COUNT(asal) FROM mahasiswa;
+-----+
| COUNT(asal) |
+-----+
|      10 |
+-----+
```

Hasilnya tetap 10 karena tentu saja kolom `asal` juga terdiri dari 10 baris data.

Selanjutnya, bagaimana jika saya ingin mencari informasi berapa jumlah mahasiswa yang berasal dari Jakarta? Cukup dengan menambahkan perintah `WHERE` untuk membuat kondisi tersebut:

```
SELECT COUNT(*) FROM mahasiswa WHERE asal = 'Jakarta';
+-----+
| COUNT(*) |
+-----+
|      4   |
+-----+
```

Untuk memeriksa apakah benar ada 4 mahasiswa yang berasal dari Jakarta, bisa dengan menjalankan query diatas tanpa function COUNT():

```
SELECT * FROM mahasiswa WHERE asal = 'Jakarta';
+-----+-----+-----+-----+-----+
| nama          | asal    | kel    | tinggi | jurusan        | nilai_uan |
+-----+-----+-----+-----+-----+
| Sari Citra Lestari | Jakarta | P     | 161   | Manajemen       | 310.60  |
| Rina Kumala Sari | Jakarta | P     | 158   | Akuntansi       | 337.99  |
| Husli Khairan   | Jakarta | L     | 170   | Akuntansi       | 288.55  |
| Ikhsan Prayoga   | Jakarta | L     | 172   | Ilmu Komputer   | 300.16  |
+-----+-----+-----+-----+-----+
```

Terlihat memang ada 4 mahasiswa yang berasal dari Jakarta.

Contoh lain, saya ingin mencari tahu jumlah mahasiswa perempuan yang ada di dalam tabel mahasiswa. Caranya juga sama, hanya saja kali ini kondisi yang diperlukan adalah `kel = 'P'`:

```
SELECT COUNT(kel) FROM mahasiswa WHERE kel = 'P';
+-----+
| COUNT(kel) |
+-----+
|      5   |
+-----+
```

Sama seperti sebelumnya, kita juga bisa memastikan hal ini dengan menjalankan query SELECT tanpa function COUNT().

Di dalam tabel `mahasiswa`, terdapat kolom `nilai_uan` yang berisi nilai ujian akhir nasional dari setiap mahasiswa. Kita bisa memanfaatkan function `MIN()` dan `MAX()` untuk mencari tahu berapa nilai tertinggi dan terendah dari seluruh nilai uan:

```
SELECT MIN(nilai_uan), MAX(nilai_uan) FROM mahasiswa;
+-----+
| MIN(nilai_uan) | MAX(nilai_uan) |
+-----+
|      280.82 |      341.10 |
+-----+
```

Terlihat bahwa nilai uan terendah adalah 280.82 dan nilai tertinggi 341.10.

Cara penggunaan function AVG() dan SUM() juga sama, misalnya saya ingin mencari rata-rata nilai uan serta total jumlah nilai uan dari ke-10 mahasiswa:

```
SELECT AVG(nilai_uan), SUM(nilai_uan) FROM mahasiswa;
+-----+
| AVG(nilai_uan) | SUM(nilai_uan) |
+-----+
| 313.351000 | 3133.51 |
+-----+
```

Hasil dari seluruh *aggregate function* ini bisa kita manfaatkan untuk membuat **subquery** (query SELECT di dalam query SELECT).

Misalnya, saya ingin mengetahui siapa nama mahasiswa yang memiliki nilai uan tertinggi. Untuk mencari informasi tersebut, kita bisa saja menggunakan 2 kali perintah query seperti contoh berikut:

```
SELECT MAX(nilai_uan) FROM mahasiswa;
+-----+
| MAX(nilai_uan) |
+-----+
|      341.10 |
+-----+

SELECT nama FROM mahasiswa WHERE nilai_uan = 341.10;
+-----+
| nama      |
+-----+
| James Situmorang |
+-----+
```

Query SELECT pertama dipakai untuk mencari nilai uan maksimum. Setelah itu nilainya saya gunakan untuk query SELECT kedua, dan terlihat bahwa nilai uan 341.10 dimiliki oleh James Situmorang.

Menggunakan subquery, kedua perintah SELECT ini bisa kita gabung dalam 1 perintah saja:

```
SELECT nama FROM mahasiswa WHERE nilai_uan =  
(SELECT MAX(nilai_uan) FROM mahasiswa);  
+-----+  
| nama |  
+-----+  
| James Situmorang |  
+-----+
```

Tanda kurung harus ditulis agar MySQL menjalankan subquery `SELECT MAX(nilai_uan)` terlebih dahulu. Setelah itu, nilainya akan dipakai untuk kondisi `WHERE` dari query `SELECT` utama. Cara ini memang lebih praktis, tapi butuh sedikit analisis dalam merangkai query-nya.

Masih menggunakan subquery, bisakah anda merancang query untuk menampilkan daftar mahasiswa yang memiliki nilai uan di atas rata-rata seluruh mahasiswa? Disini kita butuh function `AVG()`.

Baik, berikut query yang saya pakai:

```
SELECT nama FROM mahasiswa WHERE nilai_uan >=  
(SELECT AVG(nilai_uan) FROM mahasiswa);  
+-----+  
| nama |  
+-----+  
| Riana Putria |  
| Rina Kumala Sari |  
| James Situmorang |  
| Sandri Fatmala |  
| Christine Wijaya |  
+-----+
```

Dengan query ini, akan terlihat 5 mahasiswa yang memiliki nilai uan diatas rata-rata, dan disini kita malah tidak perlu tahu berapa sebenarnya nilai rata-rata tersebut.

Jika query seperti ini akan dipakai dalam pembuatan aplikasi, saya sangat sarankan untuk mengecek hasil diatas secara manual. Apakah 5 mahasiswa itu benar-benar memiliki nilai uan diatas rata-rata atau bukan. Jangan sampai kita membuat laporan yang salah.

Perintah `DISTINCT` juga bisa dikombinasikan dengan function `COUNT()`. Misalnya saya ingin mencari tahu berapa jumlah jurusan yang telah dipilih oleh mahasiswa.

Jika kita langsung menjalankan query `COUNT(jurusan)`, hasilnya adalah 10. Namun ini bukan berarti terdapat 10 jurusan yang bisa dipilih, karena di antaranya 10 ini terdapat nama jurusan yang berulang.

Untuk mendapatkan hasil yang sebenarnya, kita bisa menggunakan query berikut:

```
SELECT COUNT(DISTINCT jurusan) FROM mahasiswa;
+-----+
| COUNT(DISTINCT jurusan) |
+-----+
|      5      |
+-----+
```

Tambahan query DISTINCT akan menghapus nama jurusan yang sama.

Function COUNT(), MAX(), MIN(), AVG() dan SUM() yang kita pelajari disini bisa dikombinasikan lebih lanjut dengan query SELECT...GROUP BY. Inilah yang akan kita bahas berikutnya.

13.17 SELECT ... GROUP BY

Query SELECT...GROUP BY dipakai untuk mengelompokkan data berdasarkan kriteria tertentu. Umumnya query ini digabung dengan *aggregate function*, seperti COUNT(), MAX(), MIN(), AVG() dan SUM().

Berikut format dasar dari query SELECT...GROUP BY:

```
SELECT nama_kolom FROM nama_tabel WHERE kondisi GROUP BY nama kolom
```

Tidak semua tabel bisa menggunakan perintah GROUP BY. Query ini hanya bisa dipakai jika di dalam tabel terdapat data berulang yang bisa kita kelompokkan.

Saya akan tampilkan kembali seluruh isi tabel mahasiswa:

```
SELECT * FROM mahasiswa;
+-----+-----+-----+-----+-----+-----+
| nama       | asal     | kel    | tinggi | jurusan      | nilai_uan |
+-----+-----+-----+-----+-----+-----+
| Riana Putria | Padang   | P     | 155   | Kimia        | 339.20   |
| Rudi Permana | Bandung  | L     | 163   | Ilmu Komputer | 290.44   |
| Sari Citra Lestari | Jakarta | P     | 161   | Manajemen    | 310.60   |
| Rina Kumala Sari | Jakarta  | P     | 158   | Akuntansi    | 337.99   |
| James Situmorang | Medan    | L     | 168   | Kedokteran Gigi | 341.10   |
| Sandri Fatmala | Bandung  | P     | 165   | Ilmu Komputer | 322.91   |
| Husli Khairan | Jakarta  | L     | 170   | Akuntansi    | 288.55   |
| Christine Wijaya | Medan    | P     | 157   | Manajemen    | 321.74   |
| Ikhsan Prayoga | Jakarta  | L     | 172   | Ilmu Komputer | 300.16   |
| Bobby Permana | Medan    | L     | 161   | Ilmu Komputer | 280.82   |
+-----+-----+-----+-----+-----+-----+
```

Terlihat bahwa kolom asal, kel dan jurusan memiliki data yang berulang.

Query ORDER BY nantinya bisa dipakai untuk menjawab pertanyaannya seperti ini: "Berapa jumlah mahasiswa yang berasal dari setiap kota?"

Alternatif pertama, kita bisa menjalankan query SELECT COUNT() secara berulang untuk memeriksa setiap kota, seperti contoh berikut:

```

SELECT COUNT(asal) FROM mahasiswa WHERE asal = 'Bandung';
+-----+
| COUNT(asal) |
+-----+
|          2 |
+-----+

SELECT COUNT(asal) FROM mahasiswa WHERE asal = 'Jakarta';
+-----+
| COUNT(asal) |
+-----+
|          4 |
+-----+

SELECT COUNT(asal) FROM mahasiswa WHERE asal = 'Medan';
+-----+
| COUNT(asal) |
+-----+
|          3 |
+-----+

SELECT COUNT(asal) FROM mahasiswa WHERE asal = 'Padang';
+-----+
| COUNT(asal) |
+-----+
|          1 |
+-----+

```

Tidak ada yang salah dari query diatas, tapi cukup merepotkan karena kita harus memeriksanya satu persatu.

Alternatif kedua, saya bisa menambahkan perintah ORDER BY asal sebagai instruksi agar MySQL langsung mengelompokkan berdasarkan kolom asal jika nilainya ternyata sama:

```

SELECT asal, COUNT(asal) FROM mahasiswa GROUP BY asal;
+-----+-----+
| asal      | COUNT(asal) |
+-----+-----+
| Bandung   |          2 |
| Jakarta   |          4 |
| Medan     |          3 |
| Padang    |          1 |
+-----+-----+

```

Jauh lebih efisien, dan kita juga tidak harus menulis satu per satu nama kota yang ada.

Agar lebih informatif, data diatas bisa diolah lebih lanjut dengan diurutkan berdasarkan kota yang paling banyak ke kota yang paling sedikit. Betul, disini kita akan menggunakan query ORDER BY.

Akan tetapi, kolom apa yang dipakai sebagai proses pengurutan? Perintah ORDER BY `asal` tidak bisa dipakai, karena itu akan mengurutkan tabel berdasarkan abjad kota asal. Kita harus menguratkannya menggunakan kolom COUNT(`asal`):

```
SELECT asal, COUNT(asal) FROM mahasiswa GROUP BY asal
ORDER BY COUNT(asal) DESC;
+-----+-----+
| asal      | COUNT(asal) |
+-----+-----+
| Jakarta   |        4 |
| Medan     |        3 |
| Bandung   |        2 |
| Padang    |        1 |
+-----+-----+
```

Disini saya menggunakan dua kali function COUNT(`asal`), pertama untuk proses perhitungan dan kedua untuk proses pengurutan.

Solusi yang lebih rapi, kolom COUNT(`asal`) pertama bisa dibuat alias menggunakan perintah AS. Alias ini kemudian dipakai untuk proses pengurutan. Berikut modifikasinya:

```
SELECT asal, COUNT(asal) AS jumlah FROM mahasiswa GROUP BY asal
ORDER BY jumlah DESC;
+-----+-----+
| asal      | jumlah |
+-----+-----+
| Jakarta   |      4 |
| Medan     |      3 |
| Bandung   |      2 |
| Padang    |      1 |
+-----+-----+
```

Disini saya membuat alias `jumlah` sebagai nama pengganti dari kolom COUNT(`asal`). Alias `jumlah` kemudian dipakai kembali pada proses pengurutan dengan perintah ORDER BY `jumlah` DESC.

Sebagai latihan, bisakah anda merancang query untuk menjawab soal berikut: “Tampilkan nama jurusan terbanyak yang dipilih mahasiswa”.

Terdapat berbagai cara untuk mencari jawaban pertanyaan diatas. Yang paling sederhana, kita bisa mengurutkan tabel mahasiswa berdasarkan jurusan, kemudian menghitung mana jurusan yang paling banyak diambil. Ini adalah proses manual.

Cara lain, menggunakan query SELECT COUNT(jurusan) secara satu persatu dan mencari nilai yang paling besar. Inipun sebenarnya juga masih manual karena kita harus input query satu per satu.

Menggunakan perpaduan query GROUP BY, ORDER BY, dan LIMIT, hasil akhirnya bisa langsung didapat:

```
SELECT jurusan, COUNT(jurusan) AS jumlah FROM mahasiswa
GROUP BY jurusan ORDER BY jumlah DESC LIMIT 1;
+-----+-----+
| jurusan | jumlah |
+-----+-----+
| Ilmu Komputer | 4 |
+-----+-----+
```

Merancang query diatas (dan memahaminya) memang tidak mudah, karena disini dipadukan berbagai perintah query. Saya sendiri butuh beberapa kali percobaan untuk membuatnya. Seiring dengan banyaknya latihan dan pemecahan masalah, secara perlahan anda akan semakin mahir dalam merangkai perintah query MySQL.

Dalam query tersebut, tabel `mahasiswa` akan di kelompokkan berdasarkan `jurus`, function `COUNT(jurus)` dipakai untuk menghitung jumlah mahasiswa yang memilih setiap jurusan.

Selanjutnya, tabel diurutkan dengan jurusan yang paling banyak dipilih berada di urutan paling atas (menggunakan query `ORDER BY jumlah DESC`). Karena jurusan yang paling banyak akan berada di baris paling atas, query `LIMIT 1` akan menampilkan 1 baris ini saja.

Tidak berhenti sampai disini, kita juga bisa men-filter hasil query `ORDER BY` dengan membuat kondisi logika.

Sebagai contoh, saya ingin mencari tahu apa saja nama jurusan yang dipilih oleh setidaknya 2 atau lebih mahasiswa. Artinya, jurusan yang dipilih oleh 1 mahasiswa tidak perlu ditampilkan. Sebelum kita mulai merancang query untuk pertanyaan diatas, mari pastikan berapa sebenarnya jumlah mahasiswa yang memilih setiap jurusan:

```
SELECT jurusan, COUNT(jurusan) AS jumlah FROM mahasiswa GROUP BY jurusan
ORDER BY jumlah DESC;
+-----+-----+
| jurusan | jumlah |
+-----+-----+
| Ilmu Komputer | 4 |
| Manajemen | 2 |
| Akuntansi | 2 |
| Kimia | 1 |
| Kedokteran Gigi | 1 |
+-----+-----+
```

Terlihat bahwa jurusan Ilmu Komputer, Manajemen dan Akuntansi dipilih setidaknya oleh 2 mahasiswa. Inilah jawaban yang kita cari.

Mari coba membuat sebuah kondisi untuk hasil query GROUP BY diatas agar hanya menampilkan jurusan yang dipilih oleh lebih dari 1 mahasiswa:

```
SELECT jurusan, COUNT(jurusan) AS jumlah FROM mahasiswa GROUP BY jurusan
WHERE jumlah > 1;
```

ERROR 1064 (42000): You have an error **in** your **SQL** syntax; **check** the manual that corresponds **to** your MariaDB server **version for** the **right** syntax **to** use near '**WHERE** jumlah > 1' **at** line 1

Logika pembuatan query diatas sebenarnya sudah benar, dimana saya menggunakan kondisi WHERE jumlah > 1 agar yang ditampilkan hanya nama jurusan yang dipilih lebih dari 1 mahasiswa. Tapi kenapa tetap error?

Khusus untuk query yang menggunakan GROUP BY, pembuatan kondisi tidak bisa menggunakan perintah WHERE, tapi diganti menjadi HAVING:

```
SELECT jurusan, COUNT(jurusan) AS jumlah FROM mahasiswa GROUP BY jurusan
HAVING jumlah > 1;
+-----+-----+
| jurusan | jumlah |
+-----+-----+
| Akuntansi | 2 |
| Ilmu Komputer | 4 |
| Manajemen | 2 |
+-----+-----+
```

Sekarang, yang ditampilkan hanya jurusan yang hasil dari COUNT(jurusan) lebih dari 1. Anda bisa menambahkan lagi perintah ORDER BY agar hasilnya terurut dari jurusan yang paling banyak ke yang paling sedikit.

Masing ingat dengan *aggregate function*? Kita juga bisa menggunakan untuk query GROUP BY. Contoh kasusnya, saya ingin mengetahui berapa tinggi badan minimum, maksimum dan rata-rata dari semua mahasiswa yang berasal dari Jakarta. Berikut perintahnya tanpa query GROUP BY:

```
SELECT asal, MIN(tinggi), MAX(tinggi), AVG(tinggi) FROM mahasiswa
WHERE asal = 'Jakarta';
+-----+-----+-----+-----+
| asal | MIN(tinggi) | MAX(tinggi) | AVG(tinggi) |
+-----+-----+-----+
| Jakarta | 158 | 172 | 165.2500 |
+-----+-----+-----+
```

Sekarang, bagaimana jika saya ingin menampilkan hal yang sama untuk setiap kota? Kita bisa menggunakan query GROUP BY:

```
SELECT asal, MIN(tinggi), MAX(tinggi), AVG(tinggi) FROM mahasiswa
GROUP BY asal;
```

asal	MIN (tinggi)	MAX (tinggi)	AVG (tinggi)
Bandung	163	165	164.0000
Jakarta	158	172	165.2500
Medan	157	168	162.0000
Padang	155	155	155.0000

Hasil tampilan diatas berisi informasi mengenai tinggi minimum, tinggi maksimum, serta tinggi rata-rata mahasiswa berdasarkan asal kota.

Sebagai latihan, bisakah anda merancang query yang mirip seperti diatas untuk soal berikut ini: “Tampilkan data nilai uan minimum, nilai uan maksimum, serta nilai uan rata-rata untuk setiap jurusan yang dipilih mahasiswa”

Dari hasil query ini, nantinya akan terlihat berapa nilai uan maksimum untuk mahasiswa yang memilih jurusan Ilmu Komputer, berapa rata-rata nilai uan untuk mahasiswa yang memilih jurusan Akuntansi, dst.

Baik, berikut query yang saya gunakan:

```
SELECT jurusan, MIN(nilai_uan), MAX(nilai_uan), AVG(nilai_uan)
FROM mahasiswa GROUP BY jurusan;
```

jurusan	MIN (nilai_uan)	MAX (nilai_uan)	AVG (nilai_uan)
Akuntansi	288.55	337.99	313.270000
Ilmu Komputer	280.82	322.91	298.582500
Kedokteran Gigi	341.10	341.10	341.100000
Kimia	339.20	339.20	339.200000
Manajemen	310.60	321.74	316.170000

Seakan belum cukup, query **GROUP BY** masih punya tambahan perintah lain, yakni **WITH ROLLUP**. Perintah **WITH ROLLUP** berfungsi untuk membuat kesimpulan dari hasil query **GROUP BY**.

Agar lebih mudah dijelaskan, saya akan menambah perintah **WITH ROLLUP** ke dalam query kita sebelumnya:

```
SELECT jurusan, MIN(nilai_uan), MAX(nilai_uan), AVG(nilai_uan)
FROM mahasiswa GROUP BY jurusan WITH ROLLUP;
```

jurusan	MIN(nilai_uan)	MAX(nilai_uan)	AVG(nilai_uan)
Akuntansi	288.55	337.99	313.270000
Ilmu Komputer	280.82	322.91	298.582500
Kedokteran Gigi	341.10	341.10	341.100000
Kimia	339.20	339.20	339.200000
Manajemen	310.60	321.74	316.170000
NULL	280.82	341.10	313.351000

Perhatikan tambahan di baris terakhir. Terdapat satu baris baru dimana nama jurusan berisi nilai NULL. Inilah baris tambahan hasil dari perintah WITH ROLLUP. Sepanjang baris ini, akan berisi kesimpulan dari setiap kolom.

Di kolom MIN(nilai_uan), baris terakhir berisi nilai uan paling rendah dari seluruh jurusan, begitu juga untuk kolom MAX(nilai_uan), baris ini akan berisi nilai uan paling tinggi dari seluruh jurusan. Sedangkan untuk kolom AVG(nilai_uan), akan berisi total rata-rata nilai uan dari seluruh jurusan.

Query WITH ROLLUP ini cocok dipakai sebagai tambahan data akhir dari laporan.

Sampai disini anda bisa melihat bahwa dengan query SQL, kita bisa mengolah data sedemikian rupa. Database tidak hanya sekedar tempat menyimpan data, tapi juga untuk memproses data menjadi informasi yang dibutuhkan. Tentu saja diperlukan skill yang cukup untuk bisa merangkai berbagai perintah SQL ini.

13.18 SELECT ... JOIN

Query SELECT...JOIN dipakai untuk menggabungkan beberapa tabel. Query ini cukup terkenal karena dianggap sebagai satu-satunya cara menggabungkan tabel. Namun seperti yang telah kita pelajari, query WHERE juga bisa digunakan untuk keperluan ini.

Query JOIN hadir dengan beberapa “rasa”, yakni INNER JOIN, LEFT JOIN dan RIGHT JOIN. Cara penulisan ketiga perintah ini sama satu dengan yang lain, dengan format dasar sebagai berikut:

```
SELECT nama_kolom FROM tabel1 JOIN tabel2
WHERE tabel1.kolom1 = tabel2.kolom1
```

Membuat tabel universitas

Sebelum kita mulai membahas query JOIN, saya akan menyiapkan tabel universitas. Tabel ini nantinya berhubungan dengan tabel mahasiswa yang sudah beberapa kali kita pakai.

Berikut query pembuatan tabel universitas:

```
CREATE TABLE universitas (
    jurusan VARCHAR(20),
    tgl_berdiri DATE,
    nama_dekan VARCHAR(50),
    jum_mhs SMALLINT UNSIGNED,
    akr ENUM('A', 'B', 'C', 'N/A')
);

INSERT INTO universitas VALUES
('Kimia', '1987-07-12', 'Prof. Mulyono, M.Sc.', 662, 'B'),
('Ilmu Komputer', '2003-02-23', 'Dr. Syahrial, M.Kom.', 412, 'A'),
('Akuntansi', '1985-03-19', 'Maya Fitrianti, M.M.', 895, 'B'),
('Farmasi', '1997-05-30', 'Prof. Silvia Nst, M.Farm.', 312, 'C'),
('Fisika', '1989-12-10', 'Dr. Umar Agustinus, M.Sc.', 275, 'A'),
('Hukum', '1983-08-08', 'Prof. Gunarto, M.H.', 754, 'B');
```

```
SELECT * FROM universitas;
```

jurusan	tgl_berdiri	nama_dekan	jum_mhs	akr
Kimia	1987-07-12	Prof. Mulyono, M.Sc.	662	B
Ilmu Komputer	2003-02-23	Dr. Syahrial, M.Kom.	412	A
Akuntansi	1985-03-19	Maya Fitrianti, M.M.	895	B
Farmasi	1997-05-30	Prof. Silvia Nst, M.Farm.	312	C
Fisika	1989-12-10	Dr. Umar Agustinus, M.Sc.	275	A
Hukum	1983-08-08	Prof. Gunarto, M.H.	754	B

Tabel universitas berisi daftar nama jurusan, lengkap dengan kolom tanggal berdiri, nama dekan, jumlah mahasiswa dan akreditasi.



Sebagai info tambahan, **dekan** adalah istilah untuk jabatan kepala fakultas, sedangkan **akreditasi** adalah penilaian seberapa baik sebuah jurusan, nilainya mulai dari C, B dan A sebagai nilai paling tinggi.

Tabel universitas ini akan kita hubungkan dengan tabel mahasiswa:

```
SELECT * FROM mahasiswa;
+-----+-----+-----+-----+-----+
| nama | asal | kel | tinggi | jurusan | nilai_uan |
+-----+-----+-----+-----+-----+
| Riana Putria | Padang | P | 155 | Kimia | 339.20 |
| Rudi Permana | Bandung | L | 163 | Ilmu Komputer | 290.44 |
| Sari Citra Lestari | Jakarta | P | 161 | Manajemen | 310.60 |
| Rina Kumala Sari | Jakarta | P | 158 | Akuntansi | 337.99 |
| James Situmorang | Medan | L | 168 | Kedokteran Gigi | 341.10 |
| Sandri Fatmala | Bandung | P | 165 | Ilmu Komputer | 322.91 |
| Husli Khairan | Jakarta | L | 170 | Akuntansi | 288.55 |
| Christine Wijaya | Medan | P | 157 | Manajemen | 321.74 |
| Ikhsan Prayoga | Jakarta | L | 172 | Ilmu Komputer | 300.16 |
| Bobby Permana | Medan | L | 161 | Ilmu Komputer | 280.82 |
+-----+-----+-----+-----+-----+
```

Silahkan anda pelajari sejenak isi kedua tabel ini, terutama di kolom jurusan. Tidak semua jurusan dari tabel `mahasiswa` ada di tabel `universitas` dan begitu juga sebaliknya.

Di tabel `mahasiswa`, terdapat jurusan **Kedokteran Gigi** dan **Manajemen**. Jurusan ini tidak ada di tabel `universitas`. Sedangkan di tabel `universitas` terdapat jurusan **Farmasi**, **Fisika** dan **Hukum**. Ketiga jurusan ini juga tidak dipilih mahasiswa di tabel `mahasiswa`. Hubungan ini merupakan kunci kita agar bisa memahami berbagai query JOIN.

Seperti yang saya singgung di awal materi ini, JOIN bukan satu-satunya cara penggabungan tabel. Kita juga bisa menggunakan query `WHERE table1 = table2`.

Sebagai contoh, saya ingin menampilkan kolom `nama` dan `jurusan` dari tabel `mahasiswa`, serta kolom `dekan` dari tabel `universitas`. Kolom `jurusan` dari kedua tabel akan menjadi penghubung:

```
SELECT mahasiswa.nama, mahasiswa.jurusan, universitas.nama_dekan
FROM mahasiswa, universitas
WHERE mahasiswa.jurusan = universitas.jurusan;
+-----+-----+-----+
| nama | jurusan | nama_dekan |
+-----+-----+-----+
| Riana Putria | Kimia | Prof. Mulyono, M.Sc. |
| Rudi Permana | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Rina Kumala Sari | Akuntansi | Maya Fitrianti, M.M. |
| Sandri Fatmala | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Husli Khairan | Akuntansi | Maya Fitrianti, M.M. |
| Ikhsan Prayoga | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Bobby Permana | Ilmu Komputer | Dr. Syahrial, M.Kom. |
+-----+-----+-----+
```

Disini saya harus menggunakan penulisan *identifier qualifiers* karena nama kolom `jurusan` ada di kedua tabel.

Alur logika dari query diatas bisa dilihat dari gambar dibawah ini:

```

SELECT * FROM mahasiswa;
+-----+-----+-----+-----+-----+
| nama | asal | kel | tinggi | jurusan | nilai_uan |
+-----+-----+-----+-----+-----+
| Riana Putria | Padang | P | 155 | Kimia | 339.20 |
| Rudi Permana | Bandung | L | 163 | Ilmu Komputer | 290.44 |
| Sari Citra Lestari | Jakarta | P | 161 | Manajemen | 310.60 |
| Rina Kumala Sari | Jakarta | P | 158 | Akuntansi | 337.99 |
| James Situmorang | Medan | L | 168 | Kedokteran Gigi | 341.10 |
| Sandri Fatmala | Bandung | P | 165 | Ilmu Komputer | 322.91 |
| Husli Khairan | Jakarta | L | 170 | Akuntansi | 288.55 |
| Christine Wijaya | Medan | P | 157 | Manajemen | 321.74 |
| Ikhsan Prayoga | Jakarta | L | 172 | Ilmu Komputer | 300.16 |
| Bobby Permana | Medan | L | 161 | ilmu Komputer | 280.82 |
+-----+-----+-----+-----+-----+

```



```

SELECT * FROM universitas;
+-----+-----+-----+-----+-----+
| jurusan | tgl_berdiri | nama_dekan | jum_mhs | akr |
+-----+-----+-----+-----+-----+
| Kimia | 1987-07-12 | Prof. Mulyono, M.Sc. | 662 | B |
| Ilmu Komputer | 2003-02-23 | Dr. Syahrial, M.Kom. | 412 | A |
| Akuntansi | 1985-03-19 | Maya Fitrianti, M.M. | 895 | B |
| Farmasi | 1997-05-30 | Prof. Silvia Nst, M.Farm. | 312 | C |
| Fisika | 1989-12-10 | Dr. Umar Agustinus, M.Sc. | 275 | A |
| Hukum | 1983-08-08 | Prof. Gunarto, M.H. | 754 | B |
+-----+-----+-----+-----+-----+

```



```

SELECT mahasiswa.nama, mahasiswa.jurusan, universitas.nama_dekan
FROM mahasiswa, universitas
WHERE mahasiswa.jurusan = universitas.jurusan;
+-----+-----+-----+
| nama | jurusan | nama_dekan |
+-----+-----+-----+
| Riana Putria | Kimia | Prof. Mulyono, M.Sc. |
| Rudi Permana | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Rina Kumala Sari | Akuntansi | Maya Fitrianti, M.M. |
| Sandri Fatmala | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Husli Khairan | Akuntansi | Maya Fitrianti, M.M. |
| Ikhsan Prayoga | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Bobby Permana | Ilmu Komputer | Dr. Syahrial, M.Kom. |
+-----+-----+-----+

```

Gambar: Proses gabungan tabel mahasiswa dan universitas

Masing-masing wara dibedakan berdasarkan nama jurusan. Jurusan Kimia berwarna kuning, jurusan Akuntansi berwarna abu-abu, dan jurusan Ilmu Komputer berwarna hijau. Ketiga jurusan inilah yang digabung menjadi tampilan akhir.

Baris lainnya tidak akan diambil karena tidak memiliki jurusan yang cocok di kedua tabel. Di dalam tabel mahasiswa terdapat jurusan Kedokteran Gigi dan Manajemen, tapi pasangannya tidak ada di tabel universitas. Begitu juga dengan jurusan Farmasi, Fisika dan Hukum di tabel universitas yang juga tidak memiliki pasangan di tabel mahasiswa.

Prinsip yang sama juga dipakai untuk query INNER JOIN. Perintah diatas bisa dikonversi menjadi query berikut ini:

```

SELECT mahasiswa.nama, mahasiswa.jurusan, universitas.nama_dekan
FROM mahasiswa INNER JOIN universitas
ON mahasiswa.jurusan = universitas.jurusan;
+-----+-----+-----+
| nama | jurusan | nama_dekan |
+-----+-----+-----+
| Riana Putria | Kimia | Prof. Mulyono, M.Sc. |
| Rudi Permana | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Rina Kumala Sari | Akuntansi | Maya Fitrianti, M.M. |
| Sandri Fatmala | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Husli Khairan | Akuntansi | Maya Fitrianti, M.M. |
| Ikhsan Prayoga | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Bobby Permana | Ilmu Komputer | Dr. Syahrial, M.Kom. |
+-----+-----+-----+

```

Penjelasan dari query INNER JOIN diatas sama persis seperti query WHERE. Query INNER JOIN akan menampilkan gabungan data tabel, dimana nilainya ada di kedua tabel. Kata kunci disini adalah "nilainya harus tersedia di kedua tabel".

Di dalam MySQL, kata INNER boleh tidak ditulis, cukup JOIN saja. Dan ini akan dianggap sebagai INNER JOIN.

Karena kolom jurusan yang dipakai untuk proses penggabungan memiliki nama yang sama baik di tabel mahasiswa maupun tabel universitas, terdapat penulisan alternatif menggunakan function USING(). Berikut penulisannya:

```
SELECT mahasiswa.nama, mahasiswa.jurusan, universitas.nama_dekan
FROM mahasiswa JOIN universitas
USING (jurusan);
+-----+-----+-----+
| nama | jurusan | nama_dekan |
+-----+-----+-----+
| Riana Putria | Kimia | Prof. Mulyono, M.Sc. |
| Rudi Permana | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Rina Kumala Sari | Akuntansi | Maya Fitrianti, M.M. |
| Sandri Fatmala | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Husli Khairan | Akuntansi | Maya Fitrianti, M.M. |
| Ikhsan Prayoga | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Bobby Permana | Ilmu Komputer | Dr. Syahrial, M.Kom. |
+-----+-----+-----+
```

Perintah **FROM mahasiswa JOIN universitas USING (jurusan)** artinya kita ingin menggabungkan tabel mahasiswa dan universitas berdasarkan kolom jurusan. Syarat dari penulisan seperti ini, nama kolomnya harus sama di kedua tabel.

Prinsip utama dari **INNER JOIN** adalah nilai dari kolom jurusan harus tersedia di kedua tabel.

Di lain pihak, query **LEFT JOIN** dan **RIGHT JOIN** akan memaksa salah satu tabel untuk tetap ditampilkan nilainya (meskipun tidak memiliki pasangan).

Mari kita coba menjalankan query yang sama, tapi kali ini menggunakan **LEFT JOIN**:

```
SELECT mahasiswa.nama, mahasiswa.jurusan, universitas.nama_dekan
FROM mahasiswa LEFT JOIN universitas
ON mahasiswa.jurusan = universitas.jurusan;
+-----+-----+-----+
| nama | jurusan | nama_dekan |
+-----+-----+-----+
| Riana Putria | Kimia | Prof. Mulyono, M.Sc. |
| Rudi Permana | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Sandri Fatmala | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Ikhsan Prayoga | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Bobby Permana | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Rina Kumala Sari | Akuntansi | Maya Fitrianti, M.M. |
| Husli Khairan | Akuntansi | Maya Fitrianti, M.M. |
| Sari Citra Lestari | Manajemen | NULL |
| James Situmorang | Kedokteran Gigi | NULL |
| Christine Wijaya | Manajemen | NULL |
+-----+-----+-----+
```

Dengan query LEFT JOIN, seluruh nama mahasiswa akan ditampilkan, termasuk yang memiliki jurusan Manajemen dan Kedokteran Gigi. Informasi mengenai nama dekan untuk kedua jurusan ini tidak tersedia di tabel universitas, karena itulah nilainya diganti menjadi NULL.

Berikut ilustrasi yang menggambarkan proses yang terjadi:

```

SELECT * FROM mahasiswa;
+-----+-----+-----+-----+-----+
| nama | asal | kel | tinggi | jurusan | nilai_uan |
+-----+-----+-----+-----+-----+
| Riana Putria | Padang | P | 155 | Kimia | 339.20 |
| Rudi Permana | Bandung | L | 163 | Ilmu Komputer | 290.44 |
| Sari Citra Lestari | Jakarta | P | 161 | Manajemen | 310.60 |
| Rina Kumala Sari | Jakarta | P | 158 | Akuntansi | 327.99 |
| James Situmorang | Medan | L | 168 | Kedokteran Gigi | 341.10 |
| Sandri Fatmala | Bandung | P | 165 | Ilmu Komputer | 322.91 |
| Husli Khairan | Jakarta | L | 170 | Akuntansi | 288.55 |
| Christine Wijaya | Medan | P | 157 | Manajemen | 321.74 |
| Ikhsan Prayoga | Jakarta | L | 172 | Ilmu Komputer | 300.16 |
| Bobby Permana | Medan | L | 161 | Ilmu Komputer | 280.82 |
+-----+-----+-----+-----+-----+

```



```

SELECT * FROM universitas;
+-----+-----+-----+-----+
| jurusan | tgl_berdiri | nama_dekan | jum_mhs | akr |
+-----+-----+-----+-----+
| Kimia | 1987-07-12 | Prof. Mulyono, M.Sc. | 662 | B |
| Ilmu Komputer | 2003-02-23 | Dr. Syahrial, M.Kom. | 412 | A |
| Akuntansi | 1985-03-19 | Maya Fitrianti, M.M. | 895 | B |
| Farmasi | 1997-05-30 | Prof. Silvia Nst, M.Farm. | 312 | C |
| Fisika | 1989-12-10 | Dr. Umar Agustinus, M.Sc. | 275 | A |
| Hukum | 1983-08-08 | Prof. Gunarto, M.H. | 754 | B |
+-----+-----+-----+-----+

```



```

SELECT mahasiswa.nama, mahasiswa.jurusan, universitas.nama_dekan
FROM mahasiswa LEFT JOIN universitas
ON mahasiswa.jurusan = universitas.jurusan;
+-----+-----+-----+
| nama | jurusan | nama_dekan |
+-----+-----+-----+
| Riana Putria | Kimia | Prof. Mulyono, M.Sc. |
| Rudi Permana | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Sandri Fatmala | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Husli Khairan | Akuntansi | Maya Fitrianti, M.M. |
| Sari Citra Lestari | Manajemen | NULL |
| James Situmorang | Kedokteran Gigi | NULL |
| Christine Wijaya | Manajemen | NULL |
+-----+-----+-----+

```

Gambar: Proses LEFT JOIN dari tabel mahasiswa dengan tabel universitas

Query LEFT JOIN, juga berarti: "Ambil seluruh data dari tabel sebelah kiri, dan tampilkan nilainya meskipun tidak berpasangan". Tabel sebelah kiri ini adalah tabel mahasiswa, karena pada saat penulisan query, tabel mahasiswa-lah yang saya tempatkan di sisi kiri: FROM mahasiswa LEFT JOIN universitas

Sebaliknya, query RIGHT JOIN akan menampilkan seluruh tabel di sisi kanan, yakni tabel universitas, Meskipun tidak ada mahasiswa yang memilih jurusan tersebut:

```

SELECT mahasiswa.nama, mahasiswa.jurusan, universitas.nama_dekan
FROM mahasiswa RIGHT JOIN universitas
ON mahasiswa.jurusan = universitas.jurusan;
+-----+-----+-----+
| nama | jurusan | nama_dekan |
+-----+-----+-----+
| Riana Putria | Kimia | Prof. Mulyono, M.Sc. |
| Rudi Permana | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Rina Kumala Sari | Akuntansi | Maya Fitrianti, M.M. |
| Sandri Fatmala | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Husli Khairan | Akuntansi | Maya Fitrianti, M.M. |
| Ikhsan Prayoga | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Bobby Permana | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| NULL | NULL | Prof. Silvia Nst, M.Farm. |
| NULL | NULL | Dr. Umar Agustinus, M.Sc. |
| NULL | NULL | Prof. Gunarto, M.H. |
+-----+-----+-----+

```

Terlihat bahwa kolom nama dan jurusan berisi nilai NULL, karena kedua kolom ini tidak memiliki nilai untuk jurusan Farmasi, Fisika dan Hukum. Atau dengan kata lain, tidak ada mahasiswa yang memilih ketiga jurusan ini.

Gambar berikut mengilustrasikan proses penggabungan RIGHT JOIN:

```

SELECT * FROM mahasiswa;
+-----+-----+-----+-----+-----+
| nama | asal  | kel   | tinggi | jurusan  | nilai_uan |
+-----+-----+-----+-----+-----+
| Riana Putria | Padang | P    | 155   | Kimia     | 339.20  |
| Rudi Permana | Bandung | L   | 163   | Ilmu Komputer | 290.44  |
| Sari Citra Lestari | Jakarta | P    | 161   | Manajemen | 310.60  |
| Rina Kumala Sari | Jakarta | P    | 158   | Akuntansi | 337.99  |
| James Situmorang | Medan  | L    | 168   | Kedokteran Gigi | 341.10  |
| Sandri Fatmala | Bandung | P    | 165   | Ilmu Komputer | 322.91  |
| Husli Khairan | Jakarta | L    | 170   | Akuntansi | 288.55  |
| Christine Wijaya | Medan  | P    | 157   | Manajemen | 321.74  |
| Ikhwan Prayoga | Jakarta | L    | 172   | Ilmu Komputer | 300.16  |
| Bobby Permana | Medan  | L    | 161   | Ilmu Komputer | 280.82  |
+-----+-----+-----+-----+-----+

```



```

SELECT * FROM universitas;
+-----+-----+-----+-----+
| jurusan | tgl_berdiri | nama_dekan | jum_mhs | akr |
+-----+-----+-----+-----+
| Kimia | 1987-07-12 | Prof. Mulyono, M.Sc. | 662 | B |
| Ilmu Komputer | 2003-02-23 | Dr. Syahrial, M.Kom. | 412 | A |
| Akuntansi | 1985-03-19 | Maya Fitrianti, M.M. | 895 | B |
| Farmasi | 1997-05-30 | Prof. Silvia Nst, M.Farm. | 312 | C |
| Fisika | 1989-12-10 | Dr. Umar Agustinus, M.Sc. | 275 | A |
| Hukum | 1983-08-08 | Prof. Gunarto, M.H. | 754 | B |
+-----+-----+-----+-----+

```



```

SELECT mahasiswa.nama, mahasiswa.jurusan, universitas.nama_dekan
FROM mahasiswa RIGHT JOIN universitas
ON mahasiswa.jurusan = universitas.jurusan;
+-----+-----+-----+
| nama      | jurusan  | nama_dekan |
+-----+-----+-----+
| Riana Putria | Kimia     | Prof. Mulyono, M.Sc. |
| Rudi Permana | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Rina Kumala Sari | Akuntansi | Maya Fitrianti, M.M. |
| Sandri Fatmala | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Husli Khairan | Akuntansi | Maya Fitrianti, M.M. |
| Ikhwan Prayoga | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| Bobby Permana | Ilmu Komputer | Dr. Syahrial, M.Kom. |
| NULL        | NULL      | Prof. Silvia Nst, M.Farm. |
| NULL        | NULL      | Dr. Umar Agustinus, M.Sc. |
| NULL        | NULL      | Prof. Gunarto, M.H. |
+-----+-----+-----+

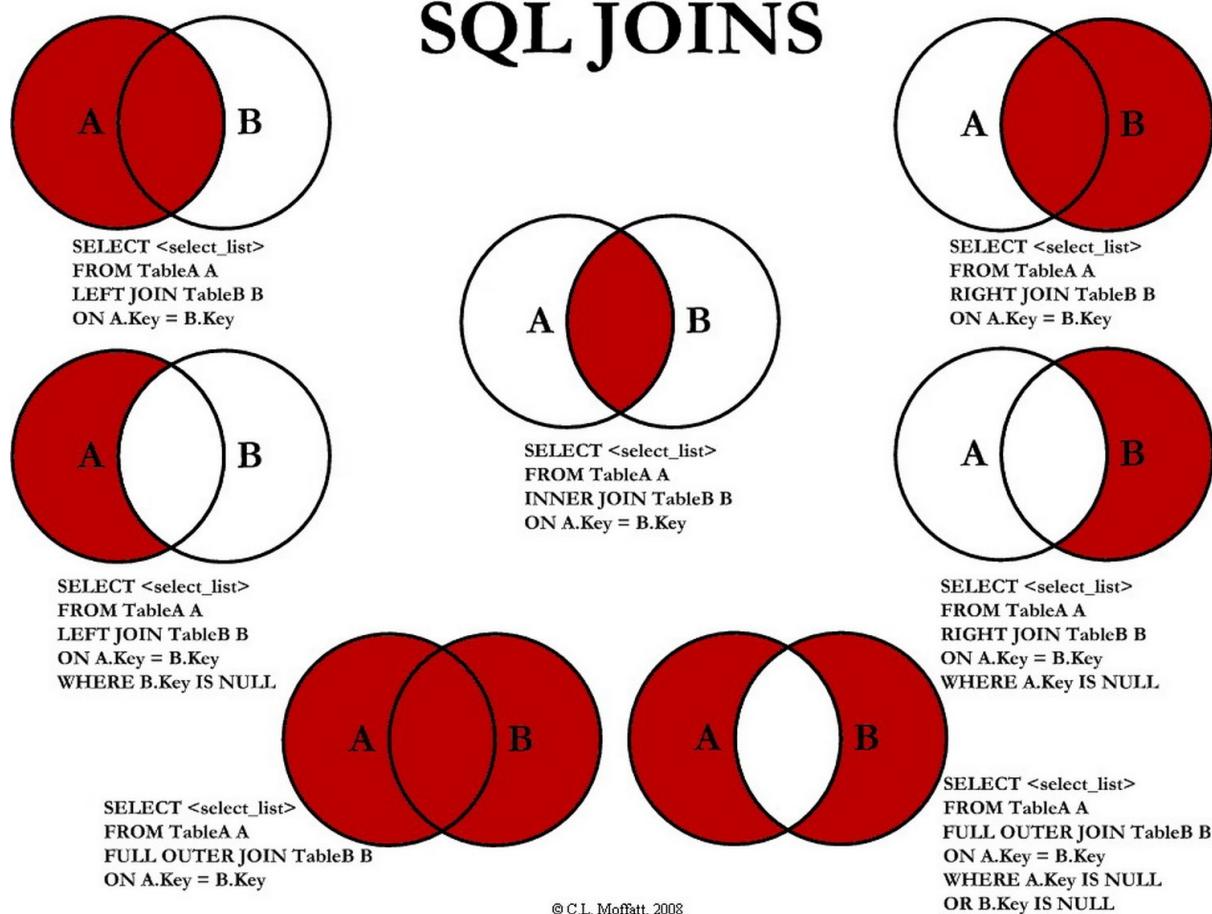
```

Gambar: Proses RIGHT JOIN dari tabel mahasiswa dengan tabel universitas

Sebenarnya, masih terdapat 1 lagi jenis JOIN, yakni FULL OUTER JOIN. Di dalam FULL OUTER JOIN, nilai yang ada di kedua tabel akan ditampilkan semua, meskipun tidak memiliki pasangan. Akan tetapi MySQL tidak mendukung query FULL OUTER JOIN.

Gambar himpunan berikut menyajikan ringkasan perbedaan dari INNER JOIN, LEFT JOIN, RIGHT JOIN serta FULL OUTER JOIN:

SQL JOINS



Gambar: Diagram perbedaan query JOIN, sumber: stackoverflow.com

Diagram yang ada di tengah adalah ilustrasi dari INNER JOIN, dimana bagian yang diarsir berada di irisan tabel A dan tabel B. Artinya, hasil dari INNER JOIN adalah data yang ada di tabel A, dan data yang sama juga harus ada di tabel B.

Diagram di kiri atas merupakan ilustrasi dari LEFT JOIN. Disini yang diarsir adalah seluruh isi tabel A, beserta irisan ke tabel B. Artinya, hasil dari LEFT JOIN adalah seluruh data yang ada di tabel A, plus data irisan dengan tabel B.

Diagram di kanan atas merupakan ilustrasi dari RIGHT JOIN. Disini yang diarsir adalah seluruh isi tabel B, beserta irisan ke tabel A. Artinya, hasil dari RIGHT JOIN adalah seluruh data yang ada di tabel B, plus data irisan dengan tabel A.

Diagram di kiri tengah adalah ilustrasi dari LEFT JOIN dengan sedikit modifikasi. Disini yang ditampilkan adalah seluruh data tabel A yang tidak memiliki pasangan dengan tabel B. Hasil ini didapat dengan penambahan kondisi WHERE kolom B.Key is NULL. Key disini merujuk kepada kolom yang dipakai untuk proses JOIN.

Berikut contohnya:

```
SELECT mahasiswa.nama, mahasiswa.jurusan, universitas.nama_dekan
FROM mahasiswa LEFT JOIN universitas
ON mahasiswa.jurusan = universitas.jurusan WHERE universitas.jurusan IS NULL;
+-----+-----+-----+
| nama | jurusan | nama_dekan |
+-----+-----+-----+
| Sari Citra Lestari | Manajemen | NULL |
| James Situmorang | Kedokteran Gigi | NULL |
| Christine Wijaya | Manajemen | NULL |
+-----+-----+-----+
```

Terlihat disini data yang ditampilkan adalah isi tabel mahasiswa yang tidak memiliki pasangan di tabel universitas.

Diagram di kanan tengah adalah ilustrasi dari **RIGHT JOIN** dengan pola yang sama. Kali ini yang ditampilkan adalah seluruh data tabel B yang tidak memiliki pasangan dengan tabel A. Hasil ini didapat dengan penambahan kondisi **WHERE** kolom A.key **is NULL**.

Berikut contohnya:

```
SELECT mahasiswa.nama, mahasiswa.jurusan, universitas.nama_dekan
FROM mahasiswa RIGHT JOIN universitas
ON mahasiswa.jurusan = universitas.jurusan WHERE mahasiswa.jurusan IS NULL;
+-----+-----+-----+
| nama | jurusan | nama_dekan |
+-----+-----+-----+
| NULL | NULL | Prof. Silvia Nst, M.Farm. |
| NULL | NULL | Dr. Umar Agustinus, M.Sc. |
| NULL | NULL | Prof. Gunarto, M.H. |
+-----+-----+-----+
```

Data diatas adalah hasil dari isi kolom universitas yang tidak memiliki pasangan di kolom mahasiswa.

Dua diagram paling bawah adalah ilustrasi dari **FULL OUTER JOIN**. MySQL tidak mendukung query ini, tapi kita bisa menghasilkan tampilan yang mirip dengan menggabungkan hasil **LEFT JOIN** dengan **RIGHT JOIN** menggunakan query **UNION**:

```
SELECT mahasiswa.nama, mahasiswa.jurusan, universitas.nama_dekan
FROM mahasiswa LEFT JOIN universitas
ON mahasiswa.jurusan = universitas.jurusan
UNION
SELECT mahasiswa.nama, mahasiswa.jurusan, universitas.nama_dekan
FROM mahasiswa RIGHT JOIN universitas
ON mahasiswa.jurusan = universitas.jurusan;
+-----+-----+-----+
| nama | jurusan | nama_dekan |
+-----+-----+-----+
```

Riana Putria	Kimia	Prof. Mulyono, M.Sc.	
Rudi Permana	Ilmu Komputer	Dr. Syahrial, M.Kom.	
Sandri Fatmala	Ilmu Komputer	Dr. Syahrial, M.Kom.	
Ikhsan Prayoga	Ilmu Komputer	Dr. Syahrial, M.Kom.	
Bobby Permana	Ilmu Komputer	Dr. Syahrial, M.Kom.	
Rina Kumala Sari	Akuntansi	Maya Fitrianti, M.M.	
Husli Khairan	Akuntansi	Maya Fitrianti, M.M.	
Sari Citra Lestari	Manajemen	NULL	
James Situmorang	Kedokteran Gigi	NULL	
Christine Wijaya	Manajemen	NULL	
NULL	NULL	Prof. Silvia Nst, M.Farm.	
NULL	NULL	Dr. Umar Agustinus, M.Sc.	
NULL	NULL	Prof. Gunarto, M.H.	

Hasil query ini terdiri dari 3 bagian: pasangan kolom yang ada di kedua tabel, kolom yang hanya ada di tabel mahasiswa saja, dan kolom yang ada di tabel universitas saja.

Query JOIN sangat penting, tapi memang perlu pemahaman yang cukup. Masih ingat dengan tabel populasi dan daftar_provinsi? Sebagai latihan, saya tantang anda untuk membuat query untuk soal berikut ini:

“Tampilkan kolom provinsi dan ibukota dari tabel daftar_provinsi serta kolom kelurahan dan kecamatan dari tabel populasi. Seluruh provinsi tetap ditampilkan meskipun tidak memiliki pasangan di tabel populasi”

Hasil akhir yang saya inginkan adalah sebagai berikut:

prov	ibukota	kec	kel
Sumatera Utara	Medan	21	151
Jawa Barat	Bandung	30	151
Sulawesi Selatan	Makassar	14	143
Jawa Tengah	Semarang	16	177
Sumatera Selatan	Palembang	14	107
Sumatera Barat	Padang	NULL	NULL
Bali	Denpasar	NULL	NULL
Papua Barat	Manokwari	NULL	NULL

Silahkan anda coba merancang querinya. Perhatikan terdapat nilai NULL di hasil akhir, artinya kita akan menggunakan apakah LEFT JOIN atau RIGHT JOIN. Baik, berikut query yang saya gunakan:

```
SELECT prov, ibukota, kec, kel
FROM daftar_provinsi LEFT JOIN populasi
ON ibukota = kota;
```

Disini saya menggunakan LEFT JOIN karena terdapat kolom yang tetap ditampilkan meskipun tidak memiliki pasangan.

Query ini saya buat tanpa penulisan identifier qualifiers karena di kedua kolom tidak terdapat kolom yang sama. Akan tetapi, penulisan identifier qualifiers akan membuat query diatas lebih mudah dibaca:

```
SELECT
daftar_provinsi.prov,
daftar_provinsi.ibukota,
populasi.kec,
populasi.kel
FROM daftar_provinsi LEFT JOIN populasi
ON daftar_provinsi.ibukota = populasi.kota;
```

Meskipun querynya lebih panjang, tapi jadi lebih mudah dipahami. Kita bisa langsung tahu kolom mana untuk tabel yang mana.

Sedikit tambahan, sebenarnya saya juga bisa menggunakan RIGHT JOIN, hanya saja kali ini urutan kolomnya harus dibalik:

```
SELECT
daftar_provinsi.prov,
daftar_provinsi.ibukota,
populasi.kec,
populasi.kel
FROM populasi RIGHT JOIN daftar_provinsi
ON daftar_provinsi.ibukota = populasi.kota;
```

Bisakah anda melihat perbedaan dari query kita sebelumnya? Bedanya ada di baris berikut:

- FROM daftar_provinsi LEFT JOIN populasi
- FROM populasi RIGHT JOIN daftar_provinsi

Keduanya akan menghasilkan tampilan yang sama.

Materi tentang JOIN menutup pembahasan cara menampilkan data tabel. Bab ini memang sangat panjang, karena banyak yang mesti dibahas.

Query SELECT akan menjadi query yang paling banyak kita gunakan sepanjang pembuatan program yang melibatkan database. Pemahaman tentang query ini sangat sangat penting. Selama mengelola dunia ilkom, mayoritas pertanyaan terkait database adalah cara menampilkan data tabel.

Query SELECT yang paling banyak dipakai nanti diantaranya ORDER BY, LIMIT, WHERE, LIKE, GROUP BY dan JOIN. Namun materi lain tetap penting untuk diketahui, bisa jadi anda akan menemukan kasus yang lebih mudah diselesaikan dengan query tertentu.

Saya sangat sarankan untuk latihan dengan berbagai tabel yang sudah kita buat. Karena latihan pemecahan masalah menjadi kunci untuk bisa merancang query, terutama yang cukup kompleks seperti GROUP BY dan JOIN.

14. Mengupdate dan Menghapus Data Tabel

Setelah kita mempelajari query untuk menginput dan menampilkan data tabel, materi kali ini akan melengkapi siklus CRUD data (Create, Read, Update, dan Delete). Dalam bab ini kita akan mempelajari cara mengupdate serta menghapus data tabel.

Query yang akan dipelajari adalah UPDATE, REPLACE, DELETE serta TRUNCATE.

Membuat tabel mahasiswa_baru

Sebagai tabel *sample*, saya akan membuat tabel `mahasiswa_baru`. Tabel ini mirip seperti tabel `mahasiswa` yang kita buat dalam bab sebelumnya namun dengan sedikit modifikasi. Berikut query untuk pembuatan tabel `mahasiswa_baru`:

```
CREATE TABLE mahasiswa_baru (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR (20),
    nilai_uan DEC(5,2)
);

INSERT INTO mahasiswa_baru VALUES
('17090113', 'Riana Putria', 'Padang', 'Kimia', 339.20),
('17140143', 'Rudi Permana', 'Bandung', 'Ilmu Komputer', 290.44),
('17090222', 'Sari Citra Lestari', 'Jakarta', 'Manajemen', 310.60),
('17080305', 'Rina Kumala Sari', 'Jakarta', 'Akuntansi', 337.99),
('17020217', 'James Situmorang', 'Medan', 'Kedokteran Gigi', 341.10),
('17140119', 'Sandri Fatmala', 'Bandung', 'Ilmu Komputer', 322.91),
('17080225', 'Husli Khairan', 'Jakarta', 'Akuntansi', 288.55),
('17090308', 'Christine Wijaya', 'Medan', 'Manajemen', 321.74),
('17140133', 'Ikhsan Prayoga', 'Jakarta', 'Ilmu Komputer', 300.16),
('17140120', 'Bobby Permana', 'Medan', 'Ilmu Komputer', 280.82);

SELECT * FROM mahasiswa_baru;
```

nim	nama	asal	jurusan	nilai_uan
17020217	James Situmorang	Medan	Kedokteran Gigi	341.10
17080225	Husli Khairan	Jakarta	Akuntansi	288.55

17080305 Rina Kumala Sari Jakarta Akuntansi 337.99
17090113 Riana Putria Padang Kimia 339.20
17090222 Sari Citra Lestari Jakarta Manajemen 310.60
17090308 Christine Wijaya Medan Manajemen 321.74
17140119 Sandri Fatmala Bandung Ilmu Komputer 322.91
17140120 Bobby Permana Medan Ilmu Komputer 280.82
17140133 Ikhsan Prayoga Jakarta Ilmu Komputer 300.16
17140143 Rudi Permana Bandung Ilmu Komputer 290.44

Saya membuat tabel `mahasiswa_baru` dengan 5 kolom.

Kolom `nim` berisi angka yang berfungsi sebagai identitas setiap mahasiswa. `nim` sendiri merupakan singkatan dari Nomor Induk Mahasiswa. Meskipun isinya angka, kolom ini lebih cocok menggunakan tipe data `CHAR` dibandingkan `INTEGER` karena kita tidak akan menggunakan nilai `nim` dalam operasi matematis.

Kolom `nim` juga menggunakan `CHAR(8)` dan bukan `VARCHAR(8)` karena kolom ini sudah pasti semuanya berisi 8 digit karakter sehingga akan lebih hemat jika menggunakan tipe data `CHAR`. Kolom `nim` ini saya set sebagai `PRIMARY KEY`.

Selanjutnya, kolom `nama` berisi nama mahasiswa. Kolom `asal` berisi kota asal dari mahasiswa. Kolom `jurusen` adalah pilihan jurusan dari setiap mahasiswa. Dan terakhir kolom `nilai_uan` berisi nilai ujian akhir sekolah dari setiap mahasiswa.

Dari hasil query `SELECT`, anda bisa melihat tampilan yang sedikit berbeda. Seluruh data `mahasiswa_baru` langsung diurutkan berdasarkan kolom `nim`, tidak lagi berdasarkan urutan pada saat proses input.

Ini merupakan fitur bawaan dari MySQL, yakni ketika sebuah tabel memiliki kolom *primary key*, data akan diurutkan berdasarkan kolom tersebut. Apabila tidak tedapat kolom primary key, data akan diurutkan berdasarkan proses input (data terbaru berada di urutan paling bawah).

Urutan ini sebenarnya tidak terlalu berpengaruh, karena seperti yang sudah di praktekkan dalam bab sebelumnya, kita bisa dengan mudah mengubah urutan ini menggunakan query `SELECT...ORDER BY`.

14.1 UPDATE

Sesuai dengan namanya, query `UPDATE` digunakan untuk mengupdate atau memperbarui data tabel. Bentuk dasar query ini adalah sebagai berikut:

```
UPDATE nama_tabel SET kolom1 = nilai1, kolom2 = nilai2 WHERE kondisi
```

Sebagai contoh, saya ingin mengupdate tabel `mahasiswa_baru` dengan mengubah nama mahasiswa yang memiliki `nim = 17020217`, menjadi `Jerry Fernando`. Berikut querynya:

```
SELECT * FROM mahasiswa_baru WHERE nim = '17020217';
+-----+-----+-----+-----+
| nim | nama | asal | jurusan | nilai_uan |
+-----+-----+-----+-----+
| 17020217 | James Situmorang | Medan | Kedokteran Gigi | 341.10 |
+-----+-----+-----+-----+
```

```
UPDATE mahasiswa_baru SET nama = 'Jerry Fernando' WHERE nim = '17020217';
Query OK, 1 row affected (0.08 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

```
SELECT * FROM mahasiswa_baru WHERE nim = '17020217';
+-----+-----+-----+-----+
| nim | nama | asal | jurusan | nilai_uan |
+-----+-----+-----+-----+
| 17020217 | Jerry Fernando | Medan | Kedokteran Gigi | 341.10 |
+-----+-----+-----+-----+
```

Sebelum proses update, saya menampilkan dulu mahasiswa yang memiliki nim 17020217. Terlihat nama mahasiswa dengan nomor nim tersebut adalah James Situmorang.

Selanjutnya query UPDATE akan mengubah nama mahasiswa ini menjadi Jerry Fernando. Dalam query diatas saya hanya mengubah 1 kolom saja, namun kita juga bisa mengubah banyak kolom sekaligus, seperti contoh berikut ini:

```
SELECT * FROM mahasiswa_baru WHERE nim = '17080305';
+-----+-----+-----+-----+
| nim | nama | asal | jurusan | nilai_uan |
+-----+-----+-----+-----+
| 17080305 | Rina Kumala Sari | Jakarta | Akuntansi | 337.99 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
UPDATE mahasiswa_baru
SET
  nama = 'Lidya Fitriana',
  asal = 'Makassar',
  nilai_uan = 329.67
WHERE nim = '17080305';
Query OK, 1 row affected (0.09 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

```
SELECT * FROM mahasiswa_baru WHERE nim = '17080305';
+-----+-----+-----+-----+
| nim | nama | asal | jurusan | nilai_uan |
+-----+-----+-----+-----+
```

```
| 17080305 | Lidya Fitriana | Makassar | Akuntansi | 329.67 |
+-----+-----+-----+-----+
```

Kali ini saya mengubah 3 kolom untuk data mahasiswa yang memiliki nim = '17080305', yakni nama = 'Lidya Fitriana', asal = 'Makassar', dan nilai_uan = 329.67.

Kunci dari query UPDATE ada di bagian **kondisi**. Di sinilah kita harus membuat aturan kolom mana yang akan diupdate. Jika kondisi ini menghasilkan 1 baris, maka nilai di baris itulah yang akan diubah. Jika kondisi menghasilkan lebih dari 1 baris, maka seluruh nilai pada baris itu akan ikut diupdate.

Sebagai contoh, saya ingin mengupdate seluruh kota asal mahasiswa dari Jakarta menjadi Semarang. Berikut penulisan querinya:

```
UPDATE mahasiswa_baru SET asal = 'Semarang' WHERE asal = 'Jakarta';
Query OK, 3 rows affected (0.08 sec)
Rows matched: 3  Changed: 3  Warnings: 0
```

```
SELECT * FROM mahasiswa_baru;
+-----+-----+-----+-----+
| nim      | nama          | asal      | jurusan        | nilai_uan |
+-----+-----+-----+-----+
| 17020217 | Jerry Fernando | Medan    | Kedokteran Gigi | 341.10 |
| 17080225 | Husli Khairan | Semarang | Akuntansi      | 288.55 |
| 17080305 | Lidya Fitriana | Makassar | Akuntansi      | 329.67 |
| 17090113 | Riana Putria   | Padang    | Kimia           | 339.20 |
| 17090222 | Sari Citra Lestari | Semarang | Manajemen       | 310.60 |
| 17090308 | Christine Wijaya | Medan    | Manajemen       | 321.74 |
| 17140119 | Sandri Fatmala | Bandung  | Ilmu Komputer  | 322.91 |
| 17140120 | Bobby Permana  | Medan    | Ilmu Komputer  | 280.82 |
| 17140133 | Ikhsan Prayoga  | Semarang | Ilmu Komputer  | 300.16 |
| 17140143 | Rudi Permana   | Bandung  | Ilmu Komputer  | 290.44 |
+-----+-----+-----+-----+
```

Terlihat bahwa seluruh kota asal Jakarta sudah berubah menjadi Semarang.

Semua kondisi pada query WHERE yang telah kita pelajari dalam bab sebelumnya (SELECT) juga bisa diterapkan pada query UPDATE, termasuk ORDER BY, LIMIT maupun yang cukup kompleks seperti JOIN.

Oh ya, sekedar mengingatkan, di dalam MySQL tidak terdapat tombol "undo" untuk mengembalikan nilai data yang sudah bertukar. Oleh karena itu anda harus hati-hati dalam perubahan data. Jangan sampai query yang kita sangka akan mengupdate baris A, malah mengupdate baris B. Atau yang lebih parah, query tersebut akan mengubah nilai seluruh kolom tabel.

Salah satu trik yang bisa dipakai adalah menggunakan query SELECT untuk memastikan kondisi apakah sudah benar atau tidak. Ini sangat berguna jika query UPDATE cukup kompleks.

Sebagai contoh, saya ingin mengubah pilihan jurusan untuk 2 orang mahasiswa Ilmu Komputer, nama jurusannya akan menjadi Teknik Informatika. Dua mahasiswa yang akan diubah adalah yang paling atas berdasarkan abjad nama.

Dari soal ini, kita bisa menyimpulkan bahwa akan dipakai query ORDER BY dan LIMIT untuk membuat kondisi ini, yakni membatasi 2 mahasiswa teratas dari jurusan Ilmu Komputer.

Pertama, saya akan rancang query SELECT untuk menampilkan 2 nama siswa ini:

```
SELECT * FROM mahasiswa_baru WHERE jurusan = 'Ilmu Komputer'
ORDER BY nama ASC;
+-----+-----+-----+-----+
| nim | nama | asal | jurusan | nilai_uan |
+-----+-----+-----+-----+
| 17140120 | Bobby Permana | Medan | Ilmu Komputer | 280.82 |
| 17140133 | Ikhsan Prayoga | Semarang | Ilmu Komputer | 300.16 |
| 17140143 | Rudi Permana | Bandung | Ilmu Komputer | 290.44 |
| 17140119 | Sandri Fatmala | Bandung | Ilmu Komputer | 322.91 |
+-----+-----+-----+-----+
```

```
SELECT * FROM mahasiswa_baru WHERE jurusan = 'Ilmu Komputer'
ORDER BY nama ASC LIMIT 2;
+-----+-----+-----+-----+
| nim | nama | asal | jurusan | nilai_uan |
+-----+-----+-----+-----+
| 17140120 | Bobby Permana | Medan | Ilmu Komputer | 280.82 |
| 17140133 | Ikhsan Prayoga | Semarang | Ilmu Komputer | 300.16 |
+-----+-----+-----+-----+
```

Pada query SELECT pertama, saya menampilkan seluruh mahasiswa yang memilih jurusan Ilmu Komputer dan diurutkan berdasarkan nama. Di query kedua, tambahan perintah LIMIT 2 akan men-filter lebih lanjut menjadi 2 mahasiswa teratas. Inilah mahasiswa yang akan kita ubah nilainya.

Hasil query diatas juga bisa dipakai sebagai pembuktian bahwa nantinya query UPDATE akan berjalan seperti yang diharapkan.

Setelah kondisi dalam query SELECT sesuai, kita tinggal memindahkan kondisi diatas ke dalam query UPDATE:

```
UPDATE mahasiswa_baru SET jurusan = 'Teknik Informatika'
WHERE jurusan = 'Ilmu Komputer' ORDER BY nama ASC LIMIT 2;
```

Query OK, 2 **rows** affected (0.06 sec)
Rows matched: 2 Changed: 2 Warnings: 0

```
SELECT * FROM mahasiswa_baru;
```

nim	nama	asal	jurusan	nilai_uan
...
17090308	Christine Wijaya	Medan	Manajemen	321.74
17140119	Sandri Fatmala	Bandung	Ilmu Komputer	322.91
17140120	Bobby Permana	Medan	Teknik Informatika	280.82
17140133	Ikhsan Prayoga	Semarang	Teknik Informatika	300.16
17140143	Rudi Permana	Bandung	Ilmu Komputer	290.44

Terlihat kolom jurusan untuk Bobby Permana dan Ikhsan Prayoga sudah berubah menjadi Teknik Informatika.

Sebagai contoh lain, saya ingin mengubah nilai_uan menjadi 100 untuk mahasiswa yang memiliki nilai_uan kurang dari 300. Berikut querynya:

```
SELECT * FROM mahasiswa_baru WHERE nilai_uan <= 300;
```

nim	nama	asal	jurusan	nilai_uan
17080225	Husli Khairan	Semarang	Akuntansi	288.55
17140120	Bobby Permana	Medan	Teknik Informatika	280.82
17140143	Rudi Permana	Bandung	Ilmu Komputer	290.44

```
UPDATE mahasiswa_baru SET nilai_uan = 200 WHERE nilai_uan <= 300;
```

Query OK, 3 **rows** affected (0.08 sec)
Rows matched: 3 Changed: 3 Warnings: 0

```
SELECT * FROM mahasiswa_baru;
```

nim	nama	asal	jurusan	nilai_uan
17020217	Jerry Fernando	Medan	Kedokteran Gigi	341.10
17080225	Husli Khairan	Semarang	Akuntansi	200.00
17080305	Lidya Fitriana	Makassar	Akuntansi	329.67
17090113	Riana Putria	Padang	Kimia	339.20

17090222 Sari Citra Lestari Semarang Manajemen 310.60
17090308 Christine Wijaya Medan Manajemen 321.74
17140119 Sandri Fatmala Bandung Ilmu Komputer 322.91
17140120 Bobby Permana Medan Teknik Informatika 200.00
17140133 Ikhsan Prayoga Semarang Teknik Informatika 300.16
17140143 Rudi Permana Bandung Ilmu Komputer 200.00

Disini saya menggunakan operator perbandingan “kurang lebih atau sama dengan” (\leq) untuk mencari nilai_uan yang kurang dari 300.

Untuk contoh yang lebih kompleks, saya ingin mengupdate tabel mahasiswa_baru yang kondisinya berada di tabel universitas.

Tabel universitas sendiri kita rancang di akhir pembahasan tentang query SELECT di bab sebelum ini. Mari cek kembali isi tabel universitas:

```
SELECT * FROM universitas;
```

jurusan	tgl_berdiri	nama_dekan	jumlah_mhs	akr
Kimia	1987-07-12	Prof. Mulyono, M.Sc.	662	B
Ilmu Komputer	2003-02-23	Dr. Syahrial, M.Kom.	412	A
Akuntansi	1985-03-19	Maya Fitrianti, M.M.	895	B
Farmasi	1997-05-30	Prof. Silvia Nst, M.Farm.	312	C
Fisika	1989-12-10	Dr. Umar Agustinus, M.Sc.	275	A
Hukum	1983-08-08	Prof. Gunarto, M.H.	754	B

Baik, soalnya adalah:

“Ubah kota asal di tabel mahasiswa_baru menjadi Pekanbaru untuk mahasiswa yang memilih jurusan dengan nama dekan Prof. Mulyono, M.Sc.”

Dari tampilan tabel universitas, kita bisa langsung melihat bahwa jurusan yang dimaksud adalah Kimia. Dan jika melihat isi tabel mahasiswa_baru, hanya terdapat 1 mahasiswa yang memilih jurusan Kimia, yakni Riana Putria.

Tentu saja kita bisa langsung mengubah nilai kota asal untuk Riana Putria dari informasi ini. Tapi saya ingin mengajak anda untuk ‘berpusing-pusing’ sedikit dengan membuat query JOIN untuk soal diatas.

Sama seperti tips sebelumnya, kita akan meracang sebuah query SELECT untuk mencari mahasiswa yang sesuai dengan kondisi soal. Silahkan anda coba sebentar.

Berikut query yang saya pakai:

```
SELECT mahasiswa_baru.nama, mahasiswa_baru.jurusan, universitas.nama_dekan
FROM mahasiswa_baru JOIN universitas
WHERE mahasiswa_baru.jurusan = universitas.jurusan
AND universitas.nama_dekan = 'Prof. Mulyono, M.Sc.';
```

nama	jurusan	nama_dekan
Riana Putria	Kimia	Prof. Mulyono, M.Sc.

Query ini cukup panjang karena saya harus menggunakan penulisan identifier qualifiers. Jika tidak, akan menghasilkan error ‘ambigu’ disebabkan kolom jurusan ada di kedua tabel yang di JOIN, yakni tabel mahasiswa_baru dan universitas.

Query diatas bisa dibaca:

“Tampilkan kolom nama dan jurusan dari tabel mahasiswa_baru, serta kolom nama_dekan dari tabel universitas, dimana kolom jurusan kedua tabel sama dan nama_dekan adalah Prof. Mulyono, M.Sc.”

Jika anda kurang paham maksud kalimat diatas, bisa dibaca secara perlahan dan kalau perlu secara berulang-ulang sembari melihat hasil query.

Untuk mengupdate data dengan kondisi ini, berikut querynya:

```
UPDATE mahasiswa_baru JOIN universitas SET asal = 'Pekanbaru'
WHERE mahasiswa_baru.jurusan = universitas.jurusan
AND universitas.nama_dekan = 'Prof. Mulyono, M.Sc.';
```

```
Query OK, 1 row affected (0.09 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

```
SELECT * FROM mahasiswa_baru;
```

nim	nama	asal	jurusan	nilai_uan
...
17080305	Lidya Fitriana	Makassar	Akuntansi	329.67
17090113	Riana Putria	Pekanbaru	Kimia	339.20
17090222	Sari Citra Lestari	Semarang	Manajemen	310.60
...

Perhatikan cara penulisan query UPDATE diatas. Karena kondisi update melibatkan 2 tabel, perintah JOIN ditulis di awal, yakni sebelum perintah SET.

Dari hasil query SELECT, terlihat bahwa kota asal dari Riana Putria sudah berubah menjadi Pekanbaru. Sekali lagi, proses penentuan kondisi menjadi faktor penting untuk mencari data yang akan diupdate.

Meskipun jarang, kita juga bisa mengupdate seluruh kolom dengan menulis perintah UPDATE tanpa (atau lupa) membuat kondisi, seperti contoh berikut:

```
UPDATE mahasiswa_baru SET asal = 'Yogyakarta';
```

```
Query OK, 10 rows affected (0.07 sec)
Rows matched: 10  Changed: 10  Warnings: 0
```

```
SELECT * FROM mahasiswa_baru;
```

nim	nama	asal	jurusan	nilai_uan
17020217	Jerry Fernando	Yogyakarta	Kedokteran Gigi	341.10
17080225	Husli Khairan	Yogyakarta	Akuntansi	200.00
17080305	Lidya Fitriana	Yogyakarta	Akuntansi	329.67
17090113	Riana Putria	Yogyakarta	Kimia	339.20
...

Dengan query diatas, seluruh kolom `asal` akan berubah menjadi Yogyakarta.

14.2 UPDATE IGNORE

Salah satu perintah tambahan yang bisa dipakai untuk query UPDATE adalah `IGNORE`.

Fungsinya query `IGNORE` ini sama seperti perintah pada query `INSERT`, yakni mengubah pesan *error* menjadi *warning*, serta ‘memaksa’ sebuah data agar bisa masuk ke dalam tabel meskipun tidak sesuai dengan tipe datanya. Untuk kasus yang terakhir ini hanya terjadi jika MySQL berjalan di `strict mode`.

Dalam bab tentang `SQL Mode`, saya sudah mengubah konfigurasi MariaDB bawaan XAMPP agar berjalan di `strict mode`. Akibatnya, query berikut akan menghasilkan error:

```
UPDATE mahasiswa_baru SET nilai_uan = '4000' WHERE nama = 'Bobby Permana';
```

```
ERROR 1264 (22003): Out of range value for column 'nilai_uan' at row 8
```

Error terjadi karena nilai `4000` tidak bisa ditampung oleh kolom `nilai_uan`. Kolom `nilai_uan` saya definisikan dengan tipe data `DEC(5,2)`, sehingga nilai maksimum yang bisa tampung hanya `999.99`.

Menggunakan query `INSERT IGNORE`, angka `4000` akan dipaksa masuk ke dalam kolom `nilai_uan` namun dikonversi menjadi `999.99`:

```
UPDATE IGNORE mahasiswa_baru SET nilai_uan = '4000'
WHERE nama = 'Bobby Permana';
```

Query OK, 1 **row** affected, 1 warning (0.09 sec)

Rows matched: 1 Changed: 1 Warnings: 1

```
SELECT * FROM mahasiswa_baru WHERE nama = 'Bobby Permana';
```

nim	nama	asal	jurusran	nilai_uan
17140120	Bobby Permana	Yogyakarta	Teknik Informatika	999.99

Jika MySQL berjalan **bukan di strict mode**, query pertama (tanpa IGNORE) akan berjalan seperti hasil diatas.

14.3 REPLACE

Query REPLACE merupakan perpaduan dari INSERT dan UPDATE. Query ini berprilaku layaknya query INSERT namun jika kolom primary key terdeteksi terdapat data yang sama, nilainya akan di-update.

Dalam materi tentang menginput data tabel (bab INSERT), terdapat tambahan perintah ON DUPLICATE KEY UPDATE. Fungsi perintah tersebut mirip dengan query REPLACE.

Sebelum kita mulai praktik, saya ingin mengembalikan data tabel `mahasiswa_baru` yang sudah kita utak-atik dalam pembahasan sebelumnya. Silahkan jalankan query berikut:

```
TRUNCATE mahasiswa_baru;
```

```
INSERT INTO mahasiswa_baru VALUES
('17090113', 'Riana Putria', 'Padang', 'Kimia', 339.20),
('17140143', 'Rudi Permana', 'Bandung', 'Ilmu Komputer', 290.44),
('17090222', 'Sari Citra Lestari', 'Jakarta', 'Manajemen', 310.60),
('17080305', 'Rina Kumala Sari', 'Jakarta', 'Akuntansi', 337.99),
('17020217', 'James Situmorang', 'Medan', 'Kedokteran Gigi', 341.10),
('17140119', 'Sandri Fatmala', 'Bandung', 'Ilmu Komputer', 322.91),
('17080225', 'Husli Khairan', 'Jakarta', 'Akuntansi', 288.55),
('17090308', 'Christine Wijaya', 'Medan', 'Manajemen', 321.74),
('17140133', 'Ikhsan Prayoga', 'Jakarta', 'Ilmu Komputer', 300.16),
('17140120', 'Bobby Permana', 'Medan', 'Ilmu Komputer', 280.82);
```

Penulisan query REPLACE tidak berbeda dengan query INSERT. Query INSERT memiliki beberapa cara penulisan, begitu juga dengan query REPLACE yang format dasarnya adalah sebagai berikut:

```
REPLACE [INTO] nama_tabel [nama_kolom] VALUES (nilai1, nilai2, ...)
REPLACE [INTO] nama tabel SET nama_kolom1 = nilai1, nama_kolom2 = nilai2, ...
```

Berikut contoh penggunaannya:

```
REPLACE INTO mahasiswa_baru
VALUES ('17090141', 'Lidya Fitriana', 'Surabaya', 'Kimia', 290.54);
Query OK, 1 row affected (0.06 sec)
```

```
SELECT * FROM mahasiswa_baru;
+-----+-----+-----+-----+-----+
| nim | nama | asal | jurusan | nilai_uan |
+-----+-----+-----+-----+-----+
| 17020217 | James Situmorang | Medan | Kedokteran Gigi | 341.10 |
| 17080225 | Husli Khairan | Jakarta | Akuntansi | 288.55 |
| 17080305 | Rina Kumala Sari | Jakarta | Akuntansi | 337.99 |
| 17090113 | Riana Putria | Padang | Kimia | 339.20 |
| 17090141 | Lidya Fitriana | Surabaya | Kimia | 290.54 |
| 17090222 | Sari Citra Lestari | Jakarta | Manajemen | 310.60 |
| ... | ... | ... | ... | ... |
+-----+-----+-----+-----+-----+
11 rows in set (0.00 sec)
```

Query REPLACE diatas akan menambah data baru ke dalam tabel `mahasiswa_baru`, karena data dengan nomor nim 17090141 belum terdapat di dalam tabel. Untuk kasus ini, query REPLACE berprilaku layaknya query `INSERT`.

Berikutnya, saya akan menjalankan query berikut:

```
REPLACE INTO mahasiswa_baru
VALUES ('17090141', 'Jerry Fernando', 'Pekanbaru', 'Kimia', 290.54);
Query OK, 2 rows affected (0.08 sec)
```

```
SELECT * FROM mahasiswa_baru;
+-----+-----+-----+-----+-----+
| nim | nama | asal | jurusan | nilai_uan |
+-----+-----+-----+-----+-----+
| 17020217 | James Situmorang | Medan | Kedokteran Gigi | 341.10 |
| 17080225 | Husli Khairan | Jakarta | Akuntansi | 288.55 |
| 17080305 | Rina Kumala Sari | Jakarta | Akuntansi | 337.99 |
| 17090113 | Riana Putria | Padang | Kimia | 339.20 |
| 17090141 | Jerry Fernando | Pekanbaru | Kimia | 290.54 |
| ... | ... | ... | ... | ... |
+-----+-----+-----+-----+-----+
11 rows in set (0.00 sec)
```

Kali ini, query REPLACE akan berprilaku seperti query UPDATE. Hal ini terjadi karena di dalam tabel mahasiswa_baru sudah terdapat data dengan nim 17090141, sehingga query REPLACE akan mengganti nilainya.

Data 17090141 tidak bisa diinput dua kali karena kolom nim sudah di set sebagai primary key. Hasilnya, nama mahasiswa dengan nim 17090141 akan diubah dari Lidya Fitriana menjadi Jerry Fernando, serta semua data yang ada di dalam query baru akan menimpa data lama.

Hasil *feedback* dari MySQL tertulis: *Query OK, 2 rows affected (0.08 sec)*. Secara internal, data lama akan dihapus terlebih dahulu baru di input data baru. Karena cara kerja inilah terdapat 2 baris yang terpengaruh (*2 rows affected*) meskipun tampak 1 baris data yang berubah.

Karena cara kerja query REPLACE yang seperti ini juga bisa menghasilkan efek yang tidak diduga, seperti contoh berikut:

```
SELECT * FROM mahasiswa_baru WHERE nim = 17090141;
+-----+-----+-----+-----+
| nim      | nama           | asal       | jurusan | nilai_uan |
+-----+-----+-----+-----+
| 17090141 | Jerry Fernando | Pekanbaru | Kimia    |     290.54 |
+-----+-----+-----+-----+
1 row in set (0.04 sec)

REPLACE INTO mahasiswa_baru (nim, asal) VALUES ('17090141', 'Yogyakarta');
Query OK, 2 rows affected (0.07 sec)
```

```
SELECT * FROM mahasiswa_baru WHERE nim = 17090141;
+-----+-----+-----+-----+
| nim      | nama | asal       | jurusan | nilai_uan |
+-----+-----+-----+-----+
| 17090141 | NULL | Yogyakarta | NULL    |      NULL |
+-----+-----+-----+-----+
```

Disini saya menggunakan cara penulisan REPLACE...SET, sehingga kita bisa untuk tidak menulis nilai seluruh kolom sebagaimana penulisan REPLACE...VALUE.

Akan tetapi efeknya kolom yang tidak diisi nilai turut terhapus (menjadi NULL, yakni nilai default dari kolom tersebut). Hasil seperti ini mirip seperti kita menggunakan query INSERT...SET.

Untuk menghindari hal diatas, sebaiknya kita hanya menggunakan query REPLACE jika ingin menginput atau mengubah data untuk seluruh baris. Jika tidak, gunakan query UPDATE jika yang diubah hanya 1 atau beberapa kolom saja.

14.4 DELETE

Sesuai dengan namanya, query DELETE digunakan untuk menghapus data. Berikut format dasar dari query ini:

```
DELETE FROM nama_tabel WHERE kondisi
```

Sama seperti query UPDATE, bagian **kondisi** akan menjadi patokan data apa yang akan dihapus.

Untuk menyamakan data sample, silahkan ‘restore’ kembali tabel mahasiswa_baru dengan menjalankan query berikut:

```
TRUNCATE mahasiswa_baru;
```

```
INSERT INTO mahasiswa_baru VALUES
('17090113', 'Riana Putria', 'Padang', 'Kimia', 339.20),
('17140143', 'Rudi Permana', 'Bandung', 'Ilmu Komputer', 290.44),
('17090222', 'Sari Citra Lestari', 'Jakarta', 'Manajemen', 310.60),
('17080305', 'Rina Kumala Sari', 'Jakarta', 'Akuntansi', 337.99),
('17020217', 'James Situmorang', 'Medan', 'Kedokteran Gigi', 341.10),
('17140119', 'Sandri Fatmala', 'Bandung', 'Ilmu Komputer', 322.91),
('17080225', 'Husli Khairan', 'Jakarta', 'Akuntansi', 288.55),
('17090308', 'Christine Wijaya', 'Medan', 'Manajemen', 321.74),
('17140133', 'Ikhsan Prayoga', 'Jakarta', 'Ilmu Komputer', 300.16),
('17140120', 'Bobby Permana', 'Medan', 'Ilmu Komputer', 280.82);
```

Sebagai contoh, saya ingin menghapus tabel mahasiswa_baru untuk data dengan nim 17090113:

```
DELETE FROM mahasiswa_baru WHERE nim = '17090113';
```

Query OK, 1 **row** affected (0.08 sec)

Feedback dari MySQL akan menginfokan jumlah baris data yang telah dihapus. Dalam query diatas, tertulis *Query OK, 1 row affected*, artinya terdapat 1 baris yang berhasil dihapus (*1 row affected*).

Kondisi yang dibuat juga bisa kompleks sebagaimana layaknya query SELECT dan UPDATE. Misalnya saya ingin menghapus 2 mahasiswa teratas jurusan Ilmu Komputer yang diurutkan berdasarkan nama:

```
DELETE FROM mahasiswa_baru WHERE jurusan = 'Ilmu Komputer'
ORDER BY nama ASC LIMIT 2;
```

Query OK, 2 **rows** affected (0.10 sec)

Atau saya juga bisa menghapus seluruh mahasiswa yang berasal dari Medan atau mahasiswa yang memilih jurusan Manajemen:

```
DELETE FROM mahasiswa_baru WHERE asal = 'Medan' OR jurusan = 'Manajemen';
```

Query OK, 3 rows affected (0.07 sec)

Terlihat terdapat 3 mahasiswa yang memenuhi kriteria ini (*3 rows affected*).

Kembali saya ingatkan bahwa tidak ada tombol undo di MySQL. Menjalankan query DELETE yang salah bisa berakibat fatal dalam aplikasi *live*. Tips dengan menjalankan query SELECT untuk menguji kondisi juga bisa diterapkan sebelum menjalankan perintah DELETE.

Query DELETE juga bisa digabung dengan kondisi yang melibatkan tabel lain (JOIN). Misalnya saya ingin menghapus seluruh mahasiswa yang memilih jurusan dengan nama dekan Maya Fitrianti, M.M.

Informasi mengenai nama dekan berada di tabel universitas, sehingga kita harus menggunakan query JOIN. Berikut contoh pencarian kondisi menggunakan query SELECT:

```
SELECT mahasiswa_baru.nama, mahasiswa_baru.jurusan, universitas.nama_dekan
FROM mahasiswa_baru JOIN universitas
WHERE mahasiswa_baru.jurusan = universitas.jurusan
AND universitas.nama_dekan = 'Maya Fitrianti, M.M.';
```

nama	jurusan	nama_dekan
Husli Khairan	Akuntansi	Maya Fitrianti, M.M.
Rina Kumala Sari	Akuntansi	Maya Fitrianti, M.M.

Terlihat bahwa ada 2 mahasiswa yang memilih jurusan yang dipimpin oleh Maya Fitrianti, M.M.. Dari tampilan diatas juga terlihat bahwa jurusan yang dimaksud adalah Akuntansi.

Setelah kondisi penghapusan didapat, kita bisa merangkainya menjadi query DELETE:

```
DELETE mahasiswa_baru FROM mahasiswa_baru JOIN universitas
WHERE mahasiswa_baru.jurusan = universitas.jurusan
AND universitas.nama_dekan = 'Maya Fitrianti, M.M.';
```

Query OK, 2 rows affected (0.05 sec)

Khusus untuk query DELETE yang melihatkan JOIN, perintah FROM ditulis diantara dua tabel yang akan digabung, seperti contoh query diatas.

Query DELETE juga bisa dijalankan tanpa kondisi WHERE, seperti contoh berikut:

```
DELETE FROM mahasiswa_baru;
```

Hasilnya, seluruh isi tabel mahasiswa_baru akan terhapus.

14.5 TRUNCATE

Query TRUNCATE sudah sering kita gunakan sepanjang buku ini. Fungsinya adalah untuk mengosongkan data tabel. Secara internal, query ini menghapus seluruh tabel (DROP TABLE), kemudian membuat ulang tabel menggunakan struktur yang sama (CREATE TABLE).

Dengan cara ini, query TRUNCATE lebih efisien dibandingkan query DELETE untuk mengosongkan tabel. Query DELETE perlu menghapus satu per satu data tabel, sedangkan query TRUNCATE langsung menghapus struktur tabel. Untuk tabel yang sederhana, perbedaan ini tidak akan terdeteksi.

Efek lain dari query TRUNCATE adalah akan me-reset kolom AUTO_INCREMENT. Sedangkan jika menggunakan query DELETE, nilai kolom AUTO_INCREMENT akan berlanjut.

Untuk melihat efek ini mari kita buat sebuah studi kasus. Saya akan membuat tabel contoh_truncate dengan query berikut ini:

```
CREATE TABLE contoh_truncate (
    a TINYINT PRIMARY KEY AUTO_INCREMENT,
    b VARCHAR(10)
);

INSERT INTO contoh_truncate (b)
VALUES ('merah'),('biru'),('kuning'),('putih');

SELECT * FROM contoh_truncate;
```

a	b
1	merah
2	biru
3	kuning
4	putih

Sekarang mari kita coba hapus semua data menggunakan query DELETE, kemudian menginput data baru:

```
DELETE FROM contoh_truncate;
Query OK, 4 rows affected (0.10 sec)

INSERT INTO contoh_truncate (b)
VALUES ('merah'),('biru'),('kuning'),('putih');

SELECT * FROM contoh_truncate;
+---+-----+
| a | b      |
+---+-----+
| 5 | merah  |
| 6 | biru   |
| 7 | kuning |
| 8 | putih  |
+---+-----+
```

Terlihat bahwa data baru akan menyambung angka kolom a yang di set sebagai AUTO_INCREMENT.

Mari kita coba hapus data menggunakan query TRUNCATE:

```
TRUNCATE contoh_truncate;
Query OK, 0 rows affected (0.28 sec)

INSERT INTO contoh_truncate (b)
VALUES ('merah'),('biru'),('kuning'),('putih');
Query OK, 4 rows affected (0.07 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM contoh_truncate;
+---+-----+
| a | b      |
+---+-----+
| 1 | merah  |
| 2 | biru   |
| 3 | kuning |
| 4 | putih  |
+---+-----+
```

Dengan menggunakan query TRUNCATE, kolom AUTO_INCREMENT akan di reset dan penomoran diulang kembali dari angka 1.

Perbedaan lain adalah, jika menggunakan query DELETE, kita bisa melihat berapa jumlah data yang telah dihapus, yakni dari *feedback* MySQL seperti *Query OK, 4 rows affected*. Sedangkan jika menggunakan query TRUNCATE, hasilnya akan selalu nol: *Query OK, 0 rows affected*.

Bab tentang mengupdate dan menghapus data tabel ini melengkapi materi seputar CRUD, yakni proses pembuatan data tabel:

- Query INSERT untuk proses pembuatan / input data.
- Query SELECT untuk proses pembacaan data.
- Query UPDATE untuk proses pembaharuan / update data.
- Query DELETE untuk proses menghapus data.

Disini saya merujuk ke **CRUD** ke proses pembuatan data tabel, bukan tabel itu sendiri. Untuk pembuatan tabel kita menggunakan query CREATE, sedangkan penghapusan tabel menggunakan query DROP. Semuanya juga sudah kita pelajari.

Selanjutnya, kita akan membahas tentang Index.

15. Index dan FullText Search

Membuat query yang bisa berjalan optimal dan dieksekusi dengan cepat merupakan hasil akhir yang diharapkan untuk setiap perancangan database. **Index** adalah salah satu metoda paling penting untuk mempercepat proses tersebut.

Dalam bab ini kita akan membahas lebih jauh tentang apa itu index, bagaimana membuatnya, serta mengenal jenis-jenis index. Di materi kedua akan dibahas tentang pencarian berbasis FullText yang juga merupakan salah satu jenis index.

15.1 Pengertian Index

Secara sederhana, **index** adalah sebuah mekanisme untuk mempercepat proses pencarian data. Index sendiri tidak menambah efek tampilan apapun ke dalam tabel. Ada atau tanpa index, tampilan tabel tidak akan berubah.

Prinsip kerja index mirip seperti index atau daftar isi di buku. Tabel tanpa index, ibarat buku tanpa daftar isi. Kita terpaksa harus menelusuri halaman satu persatu untuk mencari informasi yang dibutuhkan.

Fungsi index sebenarnya baru terasa untuk tabel dengan jumlah data yang banyak, misalnya terdiri dari 10.000 atau 1.000.000 baris data. Untuk tabel kecil seperti yang akan kita praktekkan, penambahan index tidak terlalu tampak pengaruhnya.

Meskipun demikian, materi tentang index tetap perlu dipahami karena sangat mungkin aplikasi yang kita kembangkan nantinya bisa berisi ribuan hingga jutaan data. Ketika tiba saatnya untuk optimasi database, penambahan index akan sangat bermanfaat dan hal pertama yang harus dimaksimalkan.

Sebagai tabel sample untuk pembahasan tentang index, saya kembali menggunakan tabel **mahasiswa** yang pernah kita buat di dalam bab tentang menampilkan data tabel. Berikut query pembuatan tabel tersebut:

```
DROP TABLE IF EXISTS mahasiswa;
```

```
CREATE TABLE mahasiswa (
    nama VARCHAR(50),
    asal VARCHAR(50),
    kel ENUM('L','P'),
    tinggi TINYINT UNSIGNED,
    jurusan VARCHAR (20),
    nilai_uan DEC(5,2)
);
```

```
INSERT INTO mahasiswa VALUES
('Riana Putria', 'Padang', 'P', 155, 'Kimia', 339.20),
('Rudi Permana', 'Bandung', 'L', 163, 'Ilmu Komputer', 290.44),
('Sari Citra Lestari', 'Jakarta', 'P', 161, 'Manajemen', 310.60),
('Rina Kumala Sari', 'Jakarta', 'P', 158, 'Akuntansi', 337.99),
('James Situmorang', 'Medan', 'L', 168, 'Kedokteran Gigi', 341.10),
('Sandri Fatmala', 'Bandung', 'P', 165, 'Ilmu Komputer', 322.91),
('Husli Khairan', 'Jakarta', 'L', 170, 'Akuntansi', 288.55),
('Christine Wijaya', 'Medan', 'P', 157, 'Manajemen', 321.74),
('Ikhsan Prayoga', 'Jakarta', 'L', 172, 'Ilmu Komputer', 300.16),
('Bobby Permana', 'Medan', 'L', 161, 'Ilmu Komputer', 280.82);
```

```
SELECT * FROM mahasiswa;
```

nama	asal	kel	tinggi	jurusan	nilai_uan
Riana Putria	Padang	P	155	Kimia	339.20
Rudi Permana	Bandung	L	163	Ilmu Komputer	290.44
Sari Citra Lestari	Jakarta	P	161	Manajemen	310.60
Rina Kumala Sari	Jakarta	P	158	Akuntansi	337.99
James Situmorang	Medan	L	168	Kedokteran Gigi	341.10
Sandri Fatmala	Bandung	P	165	Ilmu Komputer	322.91
Husli Khairan	Jakarta	L	170	Akuntansi	288.55
Christine Wijaya	Medan	P	157	Manajemen	321.74
Ikhsan Prayoga	Jakarta	L	172	Ilmu Komputer	300.16
Bobby Permana	Medan	L	161	Ilmu Komputer	280.82

Sebagai contoh, jika saya ingin menampilkan data mahasiswa dengan nama Sandri Fatmala, bisa menggunakan query berikut:

```
SELECT * FROM mahasiswa WHERE nama = 'Sandri Fatmala';
```

nama	asal	kel	tinggi	jurusan	nilai_uan
Sandri Fatmala	Bandung	P	165	Ilmu Komputer	322.91

Pertanyaannya, bagaimana cara MySQL memproses pencarian ini?

Logika sederhananya, MySQL akan menelusuri mulai dari baris pertama, kemudian di cek satu per satu ke bawah hingga menemukan nama = Sandri Fatmala. Proses ini setidaknya butuh 5 kali pemeriksaan, mulai dari baris pertama hingga baris ke-6 dimana Sandri Fatmala ditemukan.

Proses seperti ini sangat tidak efisien. Bayangkan jika ada 100.000 data mahasiswa dan Sandri Fatmala berada di urutan ke 98.731. Proses pencarian akan memakan waktu yang cukup lama.

Namun karena contoh tabel `mahasiswa` hanya terdiri dari 10 baris, proses pencarian tetap dieksekusi dengan cepat.

Sistem aplikasi **RDBMS** (dimana MySQL dan MariaDB merupakan salah satu diantaranya) memiliki fitur **index** untuk mempercepat proses pencarian seperti ini.

Kita bisa menginstruksikan MySQL untuk membuat index dari kolom `nama`. Index ini nantinya disimpan terpisah dari struktur tabel dan berisi kolom `nama` yang telah diurutkan. Secara internal, daftar index nantinya terhubung dengan setiap baris yang ada di dalam tabel.

Gambar berikut mengilustrasikan bentuk index dari kolom `nama`:

nama	asal	kel	tinggi	jurusan	nilai_uan
Riana Putria	Padang	P	155	Kimia	339.20
Rudi Permana	Bandung	L	163	Ilmu Komputer	290.44
Sari Citra Lestari	Jakarta	P	161	Manajemen	310.60
Rina Kumala Sari	Jakarta	P	158	Akuntansi	337.99
James Situmorang	Medan	L	168	Kedokteran Gigi	341.10
Sandri Fatmala	Bandung	P	165	Ilmu Komputer	322.91
Husli Khairan	Jakarta	L	170	Akuntansi	288.55
Christine Wijaya	Medan	P	157	Manajemen	321.74
Ikhsan Prayoga	Jakarta	L	172	Ilmu Komputer	300.16
Bobby Permana	Medan	L	161	Ilmu Komputer	280.82

Gambar: Index `nama` untuk tabel `mahasiswa`

Dengan adanya index, ketika kita ingin mencari nama Christine Wijaya, MySQL tinggal melihat dari daftar index `nama`, lalu mengakses tabel `mahasiswa`. Karena index sudah terurut, proses pencarian akan berjalan jauh lebih cepat daripada memeriksa satu-satu baris nama.

Ini ibarat mencari arti kata ‘prudent’ dari kamus bahasa Inggris. Karena kamus sudah terurut, kita bisa langsung lompat ke kata yang diawali ‘p’, kemudian mencari ‘pr’, demikian seterusnya sampai ketemu kata ‘prudent’. Proses ini jauh lebih cepat daripada mencari kata per kata mulai dari halaman pertama.



Secara internal, untuk tabel bertipe InnoDB atau MyISAM, MySQL menggunakan algoritma pencarian **BTREE** untuk index. Jika anda tertarik mempelajari bagaimana algoritma BTREE bekerja, bisa membacanya ke sini: [B-tree Algorithm¹](#)

Tapi bagaimana jika kolom `nama` sudah langsung terurut dengan membuatnya sebagai **PRIMARY KEY** (seperti contoh di tabel `mahasiswa_baru`)?

Sebenarnya proses pengurutan itu terjadi karena di dalam MySQL kolom yang diset sebagai PRIMARY KEY otomatis juga sudah ter-index. Lagi pula dalam satu tabel hanya bisa dibuat 1 buah kolom PRIMARY KEY, tidak bisa lebih. Sedangkan untuk index, kita bisa membuatnya lebih dari 1 kolom, bahkan untuk seluruh kolom yang ada di dalam tabel.

Misalkan ternyata saya banyak melakukan proses pencarian berdasarkan jurusan. Untuk men-goptimalkan waktu pencarian, saya juga bisa membuat index untuk kolom jurusan. Sehingga tabel `mahasiswa` akan memiliki 2 buah index: kolom `nama` dan kolom `jurusan`. Bagaimana dengan kolom lain? Juga bisa kita buatkan indexnya.

¹<https://en.wikipedia.org/wiki/B-tree>

Lalu timbul pertanyaan, jika index sangat bermanfaat dan mempercepat proses pencarian, kenapa tidak semua kolom saja dibuatkan indexnya?

Index memang mempercepat proses pencarian, tapi sedikit memperlama proses penambahan data, update data, dan proses menghapus data (proses yang mengharuskan perubahan isi tabel).

Setiap ada data baru yang ditambah/diubah/dihapus, index harus ditulis ulang. Misalkan saya menambah mahasiswa baru yang bernama Lidya Fitriana. Index nama harus disusun ulang agar Lidya Fitriana bisa berada setelah James Situmorang dan sebelum Riana Putria.

Selain itu, penyimpanan index juga butuh ruang harddisk, yakni ibarat kita membuat tabel kedua dari tabel mahasiswa.

Oleh karena alasan ini, sebaiknya index hanya dipakai untuk kolom yang benar-benar diperlukan saja.

Pertanyaan selanjutnya, bagaimana cara menentukan kolom yang pas dibuatkan index? Kolom yang sering berada di kondisi WHERE adalah kandidat utama kolom yang sebaiknya dibuatkan index. Perhatikan contoh query berikut:

```
SELECT nama, kel, tinggi FROM mahasiswa WHERE asal = 'Bandung' ;
```

Jika anda sering menjalankan query seperti ini, kolom yang akan diindex adalah kolom asal, bukan kolom nama, kel atau tinggi.

Proses JOIN yang kompleks dan melibatkan banyak data juga akan sangat diuntungkan dengan adanya index, karena JOIN juga melibatkan pencarian di kedua tabel.

Di dalam MySQL, terdapat fitur yang dinamakan **MySQL query optimizer**. Untuk menangani proses query yang kompleks, MySQL bisa menentukan index apa yang akan dipakai. Atau dalam kasus yang cukup jarang, MySQL query optimizer tidak akan menggunakan index jika dirasa malah akan memperlama proses pencarian.

15.2 Cara Pembuatan Index

Untuk membuat index, bisa melalui 3 buah query berikut:

- CREATE INDEX
- ALTER TABLE
- CREATE TABLE (saat pembuatan tabel)

Kita akan mulai bahas dari query CREATE INDEX yang format dasarnya adalah sebagai berikut:

```
CREATE [jenis_index] INDEX nama_index ON nama_tabel (nama_kolom);
```

Mengenai `jenis_index`, boleh diabaikan untuk sementara, kita akan belajar cara membuat index standar terlebih dahulu. Untuk membuat index standar, bagian `jenis_index` ini tidak perlu dituliskan.

Bagian `nama_index` boleh diisi dengan string apa saja. Nama index termasuk ke dalam kelompok *identifier*, seperti halnya nama kolom dan nama tabel. Nama index berfungsi sebagai identitas dari index, dan akan berguna seandainya kita ingin menghapus index tersebut.

Sebagai contoh, saya ingin membuat index `nama`, untuk kolom `nama` di dalam tabel `mahasiswa`:

```
CREATE INDEX nama ON mahasiswa (nama);
Query OK, 0 rows affected (0.19 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Menulis nama index yang sama dengan nama kolom cukup umum dipakai. Anda juga bisa menggunakan nama lain, misalnya `index_kolom_nama`.

Setelah perintah diatas, akan tercipta index dengan nama `nama`. Untuk memeriksanya, kita bisa menjalankan query `SHOW INDEX`:

```
SHOW INDEX FROM mahasiswa \G
*****
 1. row ****
    Table: mahasiswa
    Non_unique: 1
    Key_name: nama
    Seq_in_index: 1
    Column_name: nama
    Collation: A
    Cardinality: 10
    Sub_part: NULL
    Packed: NULL
    Null: YES
    Index_type: BTREE
    Comment:
Index_comment:
1 row in set (0.00 sec)
```

Dari hasil diatas, nama index dituliskan di kolom `Key_name`. Nama kolom tabel tertulis di bagian `Column_name`, serta beberapa informasi lain.

Satu informasi yang cukup penting untuk dibahas adalah `Cardinality`. `Cardinality` menunjukkan seberapa unik data yang ada di sebuah kolom. Angka 10 dari hasil diatas berarti terdapat 10 data unik pada kolom `nama`.

Karena tabel `mahasiswa` hanya terdiri dari 10 baris, artinya setiap nilai kolom `nama` adalah unik (tidak ada nama yang berulang).

Sebuah index akan lebih efisien untuk angka `cardinality` yang besar, dimana artinya banyak nilai yang harus diperiksa. Kolom dengan `cardinality` kecil tidak diuntungkan dengan penambahan index.

Untuk memeriksa nilai keunikan atau *cardinality* dari sebuah kolom, bisa menggunakan query berikut:

```
SELECT COUNT(DISTINCT nama) FROM mahasiswa;
+-----+
| COUNT(DISTINCT nama) |
+-----+
|          10          |
+-----+
```

Artinya kolom nama memiliki 10 data yang berbeda. Mari kita periksa kolom-kolom lain:

```
SELECT COUNT(DISTINCT asal) FROM mahasiswa;
+-----+
| COUNT(DISTINCT asal) |
+-----+
|          4           |
+-----+
1 row in set (0.00 sec)
```

```
SELECT COUNT(DISTINCT jurusan) FROM mahasiswa;
+-----+
| COUNT(DISTINCT jurusan) |
+-----+
|          5           |
+-----+
1 row in set (0.00 sec)
```

```
SELECT COUNT(DISTINCT kel) FROM mahasiswa;
+-----+
| COUNT(DISTINCT kel) |
+-----+
|          2           |
+-----+
```

Perhatikan nilai cardinality dari kolom `kel`. Kolom `kel` berisi data jenis kelamin mahasiswa, nilainya sudah pasti hanya salah satu dari 2: L untuk laki-laki atau P untuk perempuan. Untuk kolom seperti ini, penambahan index tidak akan banyak berpengaruh.

Kolom `asal` dan `jurusan` memiliki nilai kurang dari 10. Oleh karena data tabel mahasiswa ada 10 baris, artinya di kedua kolom ini terdapat beberapa data yang berulang (bernilai sama).

Selama nilai cardinality suatu kolom tidak terlalu kecil (dibandingkan dengan banyak data), kolom tersebut layak dibuatkan index-nya. Tentu saja kolom itu juga harus sering dipakai dalam kondisi WHERE.

Dari hasil query `SHOW INDEX` terdapat juga nilai `Index_type`: `BTREE`. `BTREE` adalah algoritma pencarian data yang dipakai untuk index. Algoritma `BTREE` menjadi algoritma default dan satu-satunya pilihan untuk jenis tabel InnoDB dan MyISAM.

Untuk tabel dengan tipe `MEMORY`, algoritma default adalah `HASH`, tetapi juga bisa diganti menjadi `BTREE`.

Perbedaan kedua algoritma ini adalah, algoritma `BTREE` cocok dipakai untuk semua kondisi perbandingan, yakni `=, >, <, <=, >=` dan `<>`. Sedangkan algoritma `HASH` sangat cepat untuk operasi perbandingan `=` dan `<>`, tetapi tidak terlalu bagus untuk operasi perbandingan yang melibatkan jangkauan: `>, <, <=,` dan `>=`.

Cara kedua untuk membuat index adalah dengan perintah `ALTER TABLE`. Berikut format dasarnya:

```
ALTER TABLE nama_tabel ADD INDEX nama_index (kolom_index);
```

Sebagai contoh, saya ingin membuat index `asal` untuk kolom `asal` dari tabel `mahasiswa`:

```
ALTER TABLE mahasiswa ADD INDEX asal (asal);
Query OK, 0 rows affected (0.25 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Khusus untuk pembuatan index menggunakan perintah `ALTER`, nama index boleh tidak ditulis. Jika ditulis tanda nama index, MySQL akan menggunakan nama index sesuai dengan judul kolom:

```
ALTER TABLE mahasiswa ADD INDEX (jurusan);
Query OK, 0 rows affected (0.16 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Jika kita menjalankan query `SHOW INDEX`, akan terlihat ada 3 index untuk tabel `mahasiswa`:

```
SHOW INDEX FROM mahasiswa \G
*****
1. row ****
  Table: mahasiswa
  Non_unique: 1
    Key_name: nama
  Seq_in_index: 1
  Column_name: nama
    Collation: A
  Cardinality: 10
    Sub_part: NULL
      Packed: NULL
      Null: YES
  Index_type: BTREE
*****
2. row ****
```

```

Table: mahasiswa
Non_unique: 1
Key_name: asal
Seq_in_index: 1
Column_name: asal
Collation: A
Cardinality: 10
Sub_part: NULL
Packed: NULL
Null: YES
Index_type: BTREE
***** 3. row *****
Table: mahasiswa
Non_unique: 1
Key_name: jurusan
Seq_in_index: 1
Column_name: jurusan
Collation: A
Cardinality: 10
Sub_part: NULL
Packed: NULL
Null: YES
Index_type: BTREE
3 rows in set (0.00 sec)

```

Terdapat 3 index di dalam database mahasiswa: nama, asal dan jurusan.

Dari tampilan diatas, ada yang sedikit aneh. Kenapa kolom **Cardinality** berisi angka 10 untuk ketiga index? Padahal kolom asal dan jurusan terdapat nilai yang berulang, dan seharusnya memiliki nilai cardinality yang lebih kecil sebagaimana yang kita lihat dari query COUNT (DISTINCT).

Jawabannya saya dapat dari MySQL manual - SHOW INDEX² :

*“Cardinality is an estimate of the number of unique values in the index. Cardinality is counted based on statistics stored as integers, so **the value is not necessarily exact even for small tables**. The higher the cardinality, the greater the chance that MySQL uses the index when doing joins. ”*

Artinya, nilai kolom cardinality dari query SHOW INDEX hanya *perkiraan* dari MySQL. Bisa jadi hasilnya tidak sesuai untuk tabel-tabel kecil. Alasan dibalik ini mungkin karena masalah performa supaya MySQL server tidak selalu menghitung cardinality setiap terjadi perubahan data.

Cara pembuatan index ketiga adalah menulisnya pada query CREATE TABLE, yakni pada saat pembuatan tabel.

Sebagai contoh, untuk membuat index nama pada pembuatan tabel mahasiswa, saya bisa menulis query CREATE TABLE sebagai berikut:

²<https://dev.mysql.com/doc/refman/5.6/en/show-index.html>

```
CREATE TABLE mahasiswa (
    nama VARCHAR(50),
    asal VARCHAR(50),
    kel ENUM('L','P'),
    tinggi TINYINT UNSIGNED,
    jurusan VARCHAR (20),
    nilai_uan DEC(5,2),
    INDEX tinggi (tinggi)
);
```

Jika ditulis seperti ini, format penulisan dari query CREATE TABLE...INDEX adalah sebagai berikut:

```
CREATE TABLE nama_tabel (
    nama_kolom tipe_data,
    ...
    INDEX nama_index (kolom_index)
);
```

15.3 Cara Menghapus Index

Untuk menghapus index, tersedia query DROP INDEX dengan format dasar sebagai berikut:

```
DROP INDEX nama_index ON nama_tabel
```

Sebagai contoh, saya ingin menghapus ketiga index yang sudah kita buat sebelum ini:

```
DROP INDEX nama ON mahasiswa;
DROP INDEX asal ON mahasiswa;
DROP INDEX jurusan ON mahasiswa;

SHOW INDEX FROM mahasiswa \G
Empty set (0.00 sec)
```

Sekarang, tabel mahasiswa tidak lagi memiliki index.

15.4 Jenis-jenis Index

Di dalam MySQL, terdapat beberapa jenis index:

- **Regular index:** index biasa (regular) yang sudah kita buat sebelum ini. Regular index membolehkan duplikasi data.

- **UNIQUE index:** index yang membatasi duplikasi data. Kolom yang diset sebagai UNIQUE index tidak bisa diisi dengan nilai yang sama (setiap data harus unik).
- **PRIMARY KEY index:** index khusus yang sekaligus berfungsi sebagai primary key dari sebuah tabel (kunci / kolom utama). Kolom yang di-set sebagai PRIMARY KEY tidak bisa diisi dengan data yang sama (sama seperti UNIQUE index).
- **FULLTEXT index:** index yang bisa digunakan untuk membuat pencarian berbasis full-text. FULLTEXT index akan dibahas dengan detail di akhir bab ini.
- **SPATIAL index:** index yang digunakan untuk spatial values, dan hanya bisa digunakan di MyISAM tabel. Index jenis ini tidak akan kita bahas.
- **HASH index:** index yang akan menggunakan algoritma HASH, bukan BTREE. Ini merupakan index default untuk tabel bertipe MEMORY, dan hanya bisa digunakan untuk tabel tipe MEMORY saja (tidak bisa untuk tabel MyISAM maupun InnoDB).

UNIQUE index sebenarnya sudah pernah kita praktekkan, yakni saat membahas tentang atribut tipe data. Jika sebuah kolom menggunakan atribut UNIQUE, itu maksudnya untuk membuat UNIQUE index.

Misalkan saya ingin menambah UNIQUE index ke dalam kolom nama, bisa menggunakan query berikut:

```
CREATE UNIQUE INDEX nama ON mahasiswa (nama);
```

Sekarang akan terjadi error jika terdapat data berulang yang diinput ke dalam kolom nama:

```
INSERT INTO mahasiswa VALUES  
( 'Riana Putria', 'Padang', 'P', 155, 'Kimia', 339.20);
```

```
ERROR 1062 (23000): Duplicate entry 'Riana Putria' for key 'nama'
```

Untuk mencoba membuat HASH index, kita harus menggunakan tabel bertipe MEMORY, berikut contohnya:

```
CREATE TABLE contoh_table_memory (  
    nama VARCHAR(50),  
    asal VARCHAR(50),  
    INDEX nama (nama)  
) ENGINE = MEMORY;  
Query OK, 0 rows affected (0.04 sec)
```

```
SHOW INDEX FROM contoh_table_memory \G  
*****  
Table: contoh_table_memory  
Non_unique: 1  
Key_name: nama  
Seq_in_index: 1
```

```
Column_name: nama
  Collation: NULL
Cardinality: 0
  Sub_part: NULL
    Packed: NULL
      Null: YES
Index_type: HASH
  Comment:
Index_comment:
```

Terlihat dari kolom Index_type: HASH, bahwa tabel bertipe MEMORY akan menggunakan algoritma HASH secara default.

Berikut contoh query untuk membuat berbagai jenis index menggunakan query CREATE INDEX dan ALTER TABLE:

```
CREATE INDEX nama_index ON nama_tabel (kolom_index);
CREATE UNIQUE INDEX nama_index ON nama_tabel (kolom_index);
CREATE FULLTEXT INDEX nama_index ON nama_tabel (kolom_index);
CREATE SPATIAL INDEX nama_index ON nama_tabel (kolom_index);

ALTER TABLE nama_tabel ADD INDEX nama_index (kolom_index);
ALTER TABLE nama_tabel ADD UNIQUE nama_index (kolom_index);
ALTER TABLE nama_tabel ADD PRIMARY KEY (kolom_index);
ALTER TABLE nama_tabel ADD FULLTEXT nama_index (kolom_index);
ALTER TABLE nama_tabel ADD SPATIAL nama_index (kolom_index);
```

15.5 Index Prefix

Untuk beberapa situasi, kita bisa membuat index hanya untuk sebagian karakter awal kolom saja (bukan untuk keseluruhan kolom). Fitur ini cocok dipakai untuk kolom yang panjang.

Sistem kerja index adalah membuat daftar dan menyusun data sedemikian rupa agar mudah diakses. Index hanya sekedar alat pencarian, data sebenarnya tetap disimpan di dalam tabel.

Untuk kolom yang panjang, misalnya VARCHAR(200), menjadi tidak efisien jika kita harus meng-index keseluruhan kolom. Apabila dengan mengakses beberapa karakter awal dari kolom sudah bisa membuat kumpulan data yang unik, itu sudah cukup untuk proses pencarian index.

Sebagai contoh, berikut hasil tampilan 3 karakter awal untuk kolom `nama` dari tabel `mahasiswa`:

```
SELECT SUBSTRING(nama, 1, 3) FROM mahasiswa;
+-----+
| SUBSTRING(nama, 1, 3) |
+-----+
| Bob                  |
| Chr                  |
| Hus                  |
| Ikh                  |
| Jam                  |
| Ria                  |
| Rin                  |
| Rud                  |
| San                  |
| Sar                  |
+-----+
```

Dinsi saya menggunakan fungsi `SUBSTRING()` untuk menampilkan 3 karakter awal dari kolom `nama`. Perhatikan bahwa cukup dengan 3 karakter saja, setiap nama sudah bisa dibedakan (tidak ada data yang sama), ini sudah cukup untuk membuat index dengan nilai cardinality tinggi.

Dengan membuat index berdasarkan 3 karakter awal saja (3 prefix), index menjadi lebih ringkas dan memerlukan ruang penyimpanan yang lebih kecil daripada membuat index untuk keseluruhan kolom `nama`.

Cara membuat index prefix adalah dengan menulis angka prefix di dalam kurung setelah nama kolom. Seperti contoh berikut:

```
DROP INDEX nama ON mahasiswa;

CREATE INDEX nama ON mahasiswa (nama(3));
```

Saya menggunakan query `DROP INDEX` untuk menghapus index `nama` yang sudah kita buat sebelum ini. Di baris kedua, saya membuat index prefix berdasarkan 3 karakter awal dari kolom `nama`.

Dibalik keunggulannya, index prefix belum tentu cocok untuk semua kasus. Jika 3 karakter awal ini menghasilkan banyak data berulang, penggunaannya juga tidak lagi efektif.

15.6 Multi-Column Index

MySQL juga mengizinkan kita untuk membuat index untuk 2 kolom sekaligus. Berikut contoh penulisannya:

```
CREATE INDEX asal_dan_jurusan ON mahasiswa (asal, jurusan);
```

Kegunaan dari multi-column index adalah untuk query yang memerlukan pencarian kedua kolom secara bersamaan.

15.7 Query EXPLAIN

MySQL menyediakan query EXPLAIN untuk melihat apakah sebuah query sudah memanfaatkan index atau tidak.

Sebagai contoh praktek, silahkan anda hapus seluruh index yang ada di tabel `mahasiswa` (jika ada), kemudian jalankan query berikut:

```
EXPLAIN SELECT * FROM mahasiswa WHERE nama = 'Sandri Fatmala' \G
*****
   id: 1
  select_type: SIMPLE
        table: mahasiswa
       type: ALL
possible_keys: NULL
         key: NULL
      key_len: NULL
        ref: NULL
       rows: 10
     Extra: Using where
1 row in set (0.05 sec)
```

Perhatikan nilai `NULL` dari beberapa kolom diatas. Artinya, MySQL tidak menemukan key atau index yang bisa dipakai.

Sekarang mari kita buat index untuk kolom `nama` dan menjalankan kembali query EXPLAIN:

```
CREATE INDEX nama ON mahasiswa (nama);
Query OK, 0 rows affected (0.69 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
EXPLAIN SELECT * FROM mahasiswa WHERE nama = 'Sandri Fatmala' \G
*****
   id: 1
  select_type: SIMPLE
        table: mahasiswa
       type: ref
possible_keys: nama
         key: nama
      key_len: 53
        ref: const
       rows: 1
     Extra: Using index condition
1 row in set (0.03 sec)
```

Di baris `possible_keys` terlihat nilai `nama`. Artinya query diatas bisa memanfaatkan index `nama` untuk mempercepat proses pencarian.

Untuk contoh yang sedikit rumit, bagaimana jika kita lihat efeknya di query JOIN? Dalam bab sebelum ini, saya membuat query JOIN untuk mencari nama mahasiswa berdasarkan nama dekan di tabel universitas.

Query yang dimaksud adalah sebagai berikut:

```
SELECT mahasiswa.nama, mahasiswa.jurusan, universitas.nama_dekan
FROM mahasiswa INNER JOIN universitas
ON mahasiswa.jurusan = universitas.jurusan
AND universitas.nama_dekan = 'Maya Fitrianti, M.M.';
```

nama	jurusan	nama_dekan
Rina Kumala Sari	Akuntansi	Maya Fitrianti, M.M.
Husli Khairan	Akuntansi	Maya Fitrianti, M.M.

Disini saya menampilkan gabungan tabel `mahasiswa` dengan tabel `universitas` untuk menampilkan data mahasiswa yang memilih jurusan dengan nama dekan Maya Fitrianti, M.M..

Mari lihat hasil dari query EXPLAIN:

```
EXPLAIN SELECT mahasiswa.nama, mahasiswa.jurusan, universitas.nama_dekan
FROM mahasiswa INNER JOIN universitas
ON mahasiswa.jurusan = universitas.jurusan
AND universitas.nama_dekan = 'Maya Fitrianti, M.M.' \G
```

```
***** 1. row *****
    id: 1
  select_type: SIMPLE
        table: universitas
        type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
        ref: NULL
      rows: 6
    Extra: Using where
***** 2. row *****
    id: 1
  select_type: SIMPLE
        table: mahasiswa
        type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
```

```

ref: NULL
rows: 10
Extra: Using where; Using join buffer (flat, BNL join)
2 rows in set (0.00 sec)

```

Terdapat 2 baris hasil query EXPLAIN. Ini karena di dalam query JOIN ada 2 proses pencarian, pertama untuk menyamakan nilai kolom jurusan, dan kedua untuk mencari nama dekan yang bernama Maya Fitrianti, M.M..

Tidak terlihat informasi seputar index, karena tentu saja kita belum membuat index untuk tabel universitas. Selain itu, di tabel mahasiswa juga baru ada index untuk kolom nama. Sayangnya, kolom nama tidak dipakai dalam query JOIN sehingga index tersebut tidak bisa dipakai.

Melihat query JOIN, terdapat 3 kandidat kolom yang layak dibuat index, yakni kolom jurusan di tabel mahasiswa, serta kolom jurusan dan kolom nama_dekan di tabel universitas. MySQL harus menelusuri ketiga kolom ini untuk bisa menampilkan data diatas.

Ingat, kandidat untuk index adalah kolom yang dipakai dalam proses pencarian, bukan data yang ditampilkan. Dalam query JOIN ini saya juga menampilkan kolom nama dari tabel mahasiswa, tapi kolom ini tidak dipakai untuk proses pencarian sehingga index untuk kolom ini juga tidak digunakan.

Baik, mari kita tambahkan index ke kolom jurusan di tabel mahasiswa:

```

ALTER TABLE mahasiswa ADD INDEX jurusan (jurusan);
Query OK, 0 rows affected (0.25 sec)
Records: 0  Duplicates: 0  Warnings: 0

EXPLAIN SELECT mahasiswa.nama, mahasiswa.jurusan, universitas.nama_dekan
FROM mahasiswa INNER JOIN universitas
ON mahasiswa.jurusan = universitas.jurusan
AND universitas.nama_dekan = 'Maya Fitrianti, M.M.' \G

```

```

***** 1. row *****
    id: 1
  select_type: SIMPLE
        table: universitas
       type: ALL
possible_keys: NULL
         key: NULL
      key_len: NULL
        ref: NULL
       rows: 6
  Extra: Using where
***** 2. row *****
    id: 1
  select_type: SIMPLE
        table: mahasiswa

```

```

    type: ALL
possible_keys: jurusan
      key: NULL
    key_len: NULL
      ref: NULL
     rows: 10
Extra: Using where; Using join buffer (flat, BNL join)
2 rows in set (0.00 sec)

```

Perhatikan nilai possible_keys di baris kedua, isinya berubah dari NULL menjadi jurusan. Artinya, index jurusan sudah dipakai untuk pencarian di kolom universitas.

Sekarang kita akan buat index untuk kolom jurusan dan nama_dekan di tabel universitas:

```

ALTER TABLE universitas ADD INDEX jurusan (jurusan);
Query OK, 0 rows affected (0.19 sec)
Records: 0  Duplicates: 0  Warnings: 0

ALTER TABLE universitas ADD INDEX nama_dekan (nama_dekan);
Query OK, 0 rows affected (0.34 sec)
Records: 0  Duplicates: 0  Warnings: 0

```

```

EXPLAIN SELECT mahasiswa.nama, mahasiswa.jurusan, universitas.nama_dekan
FROM mahasiswa INNER JOIN universitas
ON mahasiswa.jurusan = universitas.jurusan
AND universitas.nama_dekan = 'Maya Fitrianti, M.M.' \G

```

```

*****
1. row *****
    id: 1
  select_type: SIMPLE
        table: universitas
        type: ref
possible_keys: jurusan,nama_dekan
      key: nama_dekan
    key_len: 53
      ref: const
     rows: 1
Extra: Using index condition; Using where
*****
2. row *****
    id: 1
  select_type: SIMPLE
        table: mahasiswa
        type: ref
possible_keys: jurusan
      key: jurusan
    key_len: 23

```

```
ref: belajar.universitas.jurusan
rows: 1
Extra:
2 rows in set (0.13 sec)
```

Terlihat, tidak ada lagi nilai NULL. Artinya, query JOIN diatas sudah memanfaatkan index untuk proses pencarian data tabel.

Jika anda tertarik mempelajari optimasi query, bisa menelusuri lebih jauh apa maksud setiap baris diatas. Selain itu praktek tentang index akan lebih cocok jika melibatkan jumlah data yang besar, agar kita bisa menghitung jeda waktu nyata sebelum dan sesudah pembuatan index.

Materi tentang optimasi query juga bisa sangat kompleks dan butuh menjadi 1 buku sendiri. Selain itu, index hanya salah satu cara mempercepat proses menampilkan data.

Untuk aplikasi atau web yang sangat besar, kadang perlu pemisahan server untuk penulisan dan pembacaan data. Satu server khusus untuk proses pembacaan data saja (query SELECT), dan satu server khusus untuk penulisan (query INSERT, UPDATE, dan DELETE).

Nanti ada lagi trik lain seperti *cache*, *replikasi*, dan *load balancing* untuk mengefisienkan pemrosesan data di database. Materi-materi seperti ini termasuk kategori advanced dan baru dibutuhkan jika anda mulai merancang aplikasi skala besar.

Jika tertarik seperti apa perancangan server dan database untuk project skala besar seperti Tokopedia, bisa membaca artikelnya kesini: [Buka-bukaan Teknologi Tokopedia dari Masa ke Masa³](#).

15.8 FULLTEXT Index

FULLTEXT index adalah jenis index khusus yang menyediakan fitur pencarian. Index jenis ini hanya bisa dipakai untuk kolom bertipe CHAR, VARCHAR dan TEXT.

Pencarian berbasis FULLTEXT memberikan hasil yang lebih baik karena melibatkan faktor “relevansi” ke hasil pencarian. Jika menggunakan query LIKE atau REGEXP, hasil pencarian hanya mempertimbangkan ada atau tidaknya sebuah kata atau pola.

Misalkan kita ingin mencari teks ‘PHP’ dari sebuah tabel yang berisi kumpulan artikel, pencarian berbasis FULLTEXT akan menempatkan artikel yang memiliki banyak kata “PHP” di baris hasil paling atas. Sedangkan jika menggunakan query LIKE atau REGEXP, hasilnya akan ditampilkan sesuai dengan urutan kolom.

Sebagai sample, saya akan membuat tabel artikel dengan query berikut:

³<https://www.labana.id/view/buka-bukaan-teknologi-tokopedia-dari-masa-ke-masa/2016/04/29/>

```
CREATE TABLE artikel (
    id_article TINYINT PRIMARY KEY AUTO_INCREMENT,
    judul VARCHAR(200)
);

INSERT INTO artikel (judul) VALUES
('Pengertian dan Fungsi PHP dalam Pemrograman Web'),
('Sejarah PHP dan Perkembangan Versi PHP dari masa ke masa'),
('Cara Menginstall dan Mendownload PHP dengan XAMPP'),
('Cara Menjalankan Web Server Apache dengan XAMPP'),
('Cara Menjalankan File PHP dengan XAMPP'),
('Cara Kerja Web Server Apache'),
('Cara Memasukkan kode PHP ke dalam HTML'),
('Aturan Dasar Penulisan Kode PHP'),
('Mengenal Tipe Data Integer dalam Bahasa Pemrograman PHP'),
('Mengenal Tipe Data Float dalam Bahasa Pemrograman PHP');

SELECT * FROM artikel;
+-----+-----+
| id_article | judul
+-----+-----+
| 1 | Pengertian dan Fungsi PHP dalam Pemrograman Web |
| 2 | Sejarah PHP dan Perkembangan Versi PHP dari masa ke masa |
| 3 | Cara Menginstall dan Mendownload PHP dengan XAMPP |
| 4 | Cara Menjalankan Web Server Apache dengan XAMPP |
| 5 | Cara Menjalankan File PHP dengan XAMPP |
| 6 | Cara Kerja Web Server Apache |
| 7 | Cara Memasukkan kode PHP ke dalam HTML |
| 8 | Aturan Dasar Penulisan Kode PHP |
| 9 | Mengenal Tipe Data Integer dalam Bahasa Pemrograman PHP |
| 10 | Mengenal Tipe Data Float dalam Bahasa Pemrograman PHP |
+-----+-----+
```

Tabel artikel berisi 10 judul artikel tentang PHP. Mari kita coba mencari artikel yang didalamnya terdapat teks ‘PHP’ menggunakan query LIKE:

```
SELECT * FROM artikel WHERE judul LIKE '%PHP%';
+-----+
| id_article | judul
+-----+
| 1 | Pengertian dan Fungsi PHP dalam Pemrograman Web |
| 2 | Sejarah PHP dan Perkembangan Versi PHP dari masa ke masa |
| 3 | Cara Menginstall dan Mendownload PHP dengan XAMPP |
| 5 | Cara Menjalankan File PHP dengan XAMPP |
| 7 | Cara Memasukkan kode PHP ke dalam HTML |
| 8 | Aturan Dasar Penulisan Kode PHP |
| 9 | Mengenal Tipe Data Integer dalam Bahasa Pemrograman PHP |
| 10 | Mengenal Tipe Data Float dalam Bahasa Pemrograman PHP |
+-----+
```

Query diatas akan menampilkan seluruh judul artikel yang memiliki kata ‘PHP’ di dalamnya. Hasilnya sudah sesuai, dan tabel diurutkan berdasarkan kolom `id_article` yang juga di set sebagai PRIMARY KEY.

Pencarian berbasis FULLTEXT akan menampilkan hasil yang lebih baik dengan memperhitungkan faktor relevansi. Judul artikel yang memiliki banyak kata ‘PHP’ akan ditampilkan paling atas.

Untuk menambah FULLTEXT index ke dalam tabel yang sudah ada, kita bisa memakai query `CREATE INDEX` maupun `ALTER TABLE`. Serta juga bisa pada saat pembuatan tabel dengan query `CREATE TABLE`.

Saya akan menggunakan query `ALTER TABLE` untuk membuat FULLTEXT index ke dalam tabel `artikel`:

```
ALTER TABLE artikel ADD FULLTEXT judul (judul);
Query OK, 0 rows affected, 1 warning (1.03 sec)
Records: 0  Duplicates: 0  Warnings: 1
```

```
SHOW WARNINGS;
+-----+
| Level | Code | Message
+-----+
| Warning | 124 | InnoDB rebuilding table to add column FTS_DOC_ID |
+-----+
```

Terdapat 1 warnings yang isinya berupa pesan bahwa INNODB akan membuat ulang tabel untuk menambahkan FULLTEXT Index ke dalam kolom `judul`. Tidak ada masalah.

Proses pencarian FULLTEXT memakai perintah `SELECT...MATCH...AGAINST`, dengan format dasar sebagai berikut:

```
SELECT nama_kolom FROM nama_tabel WHERE MATCH(nama_kolom) AGAINST(pencarian)
```

Sebagai contoh, saya ingin mencari kata ‘PHP’ dari kolom judul di dalam tabel artikel:

```
SELECT * FROM artikel WHERE MATCH(judul) AGAINST ('PHP');
```

id_article	judul
2	Sejarah PHP dan Perkembangan Versi PHP dari masa ke masa
1	Pengertian dan Fungsi PHP dalam Pemrograman Web
3	Cara Menginstall dan Mendownload PHP dengan XAMPP
5	Cara Menjalankan File PHP dengan XAMPP
7	Cara Memasukkan kode PHP ke dalam HTML
8	Aturan Dasar Penulisan Kode PHP
9	Mengenal Tipe Data Integer dalam Bahasa Pemrograman PHP
10	Mengenal Tipe Data Float dalam Bahasa Pemrograman PHP

Perhatikan hasil pencarian yang di dapat. Di baris pertama yang tampil adalah judul nomor 2, karena di judul ini terdapat 2 buah kata ‘PHP’ sehingga akan diprioritaskan.

Contoh lain, saya ingin mencari artikel yang memiliki kata ‘PHP’ dan ‘XAMPP’:

```
SELECT * FROM artikel WHERE MATCH(judul) AGAINST ('PHP XAMPP');
```

id_article	judul
3	Cara Menginstall dan Mendownload PHP dengan XAMPP
5	Cara Menjalankan File PHP dengan XAMPP
4	Cara Menjalankan Web Server Apache dengan XAMPP
2	Sejarah PHP dan Perkembangan Versi PHP dari masa ke masa
1	Pengertian dan Fungsi PHP dalam Pemrograman Web
7	Cara Memasukkan kode PHP ke dalam HTML
8	Aturan Dasar Penulisan Kode PHP
9	Mengenal Tipe Data Integer dalam Bahasa Pemrograman PHP
10	Mengenal Tipe Data Float dalam Bahasa Pemrograman PHP

Untuk pencarian seperti ini, judul yang memiliki kedua kata akan ditempatkan di baris paling awal, kemudian baru diikuti dengan judul yang memiliki salah satu dari dua kata tersebut.

Berikutnya, saya ingin mencari judul artikel untuk ‘tipe data PHP’:

```
SELECT * FROM artikel WHERE MATCH(judul) AGAINST ('Tipe data PHP');
+-----+
| id_article | judul
+-----+
| 9 | Mengenal Tipe Data Integer dalam Bahasa Pemrograman PHP |
| 10 | Mengenal Tipe Data Float dalam Bahasa Pemrograman PHP |
| 2 | Sejarah PHP dan Perkembangan Versi PHP dari masa ke masa |
| 1 | Pengertian dan Fungsi PHP dalam Pemrograman Web |
| 3 | Cara Menginstall dan Mendownload PHP dengan XAMPP |
| 5 | Cara Menjalankan File PHP dengan XAMPP |
| 7 | Cara Memasukkan kode PHP ke dalam HTML |
| 8 | Aturan Dasar Penulisan Kode PHP |
+-----+
```

Kembali, judul yang memiliki ketiga kata ini akan ditampilkan terlebih dahulu, baru diikuti dengan judul yang hanya memiliki salah satu dari kata: ‘Tipe’, ‘data’, dan ‘PHP’.

Dengan percobaan ini, terlihat bahwa fitur FULLTEXT index akan memberikan hasil pencarian yang jauh lebih baik daripada query LIKE.

15.9 Jenis Pencarian FULLTEXT Index

Pencarian FULLTEXT index hadir dengan 3 pilihan mode pencarian:

- Natural Language Mode
- Boolean Mode
- Query Expansion Mode

Mari kita bahas satu per satu.

Natural Language Mode

Natural language mode merupakan mode pencarian default dari FULLTEXT index. Beberapa contoh pencarian yang sudah kita buat sebelum ini adalah hasil dari *Natural language mode*. *Natural language mode* menghitung relevansi dari setiap kata yang dicari, kemudian menyusunnya dengan hasil yang paling relevan ditempatkan di baris paling atas.

Meskipun mode ini sudah aktif secara default, namun bisa juga ditulis kembali di dalam perintah AGAINST, seperti contoh berikut:

```
SELECT * FROM artikel WHERE MATCH(judul)
AGAINST ('Tipe data PHP' IN NATURAL LANGUAGE MODE);
+-----+
| id_article | judul
+-----+
| 9 | Mengenal Tipe Data Integer dalam Bahasa Pemrograman PHP
| 10 | Mengenal Tipe Data Float dalam Bahasa Pemrograman PHP
| 2 | Sejarah PHP dan Perkembangan Versi PHP dari masa ke masa
| ... | ...
+-----+
```

Hasilnya akan sama jika ditulis tanpa tambahan perintah IN NATURAL LANGUAGE MODE.

Boolean Mode

Mode pencarian FULLTEXT Boolean mode menambahkan fitur *modifiers* untuk men-filter pencarian lebih lanjut.

Tanda minus (-) diawal kata bisa dipakai untuk *menegasikan* kata tersebut. Misalkan saya ingin mencari judul artikel yang mengandung kata ‘PHP’, tapi tidak boleh berisi kata ‘XAMPP’. Querynya adalah sebagai berikut:

```
SELECT * FROM artikel WHERE MATCH(judul)
AGAINST ('PHP -XAMPP' IN BOOLEAN MODE);
+-----+
| id_article | judul
+-----+
| 2 | Sejarah PHP dan Perkembangan Versi PHP dari masa ke masa
| 1 | Pengertian dan Fungsi PHP dalam Pemrograman Web
| 7 | Cara Memasukkan kode PHP ke dalam HTML
| 8 | Aturan Dasar Penulisan Kode PHP
| 9 | Mengenal Tipe Data Integer dalam Bahasa Pemrograman PHP
| 10 | Mengenal Tipe Data Float dalam Bahasa Pemrograman PHP
+-----+
```

Agar pencarian aktif di *boolean mode*, harus ditambahkan string IN BOOLEAN MODE di dalam perintah AGAINST yang ditulis setelah string pencarian.

Lawan dari karakter minus (-) adalah plus (+) yang juga bisa dipakai untuk menegaskan bahwa kata tersebut harus ada di hasil pencarian. Tanda plus ini boleh tidak ditulis (dianggap sebagai nilai default). Artinya, query diatas juga bisa ditulis seperti ini:

```
SELECT * FROM artikel WHERE MATCH(judul)
AGAINST ('+PHP -XAMPP' IN BOOLEAN MODE);
```

Tanda bintang (*) bisa dipakai sebagai *wildcard*, yang akan mencari kata dengan awalan karakter sebelum tanda bintang.

Sebagai contoh, saya ingin mencari judul yang di dalamnya terdapat kata dengan awalan ‘menja’, namun tanpa kata ‘PHP’:

```
SELECT * FROM artikel WHERE MATCH(judul)
AGAINST ('menja* -PHP' IN BOOLEAN MODE);
+-----+
| id_article | judul
+-----+
|        4 | Cara Menjalankan Web Server Apache dengan XAMPP |
+-----+
```

Karakter menja* akan cocok dengan kata Menjalankan.

Query Expansion Mode

Di dalam mode **Query Expansion**, string pencarian akan “dikembangkan” untuk selain kata yang dicari. Contoh penggunaannya sebagai berikut:

```
SELECT * FROM artikel WHERE MATCH(judul)
AGAINST ('Integer' WITH QUERY EXPANSION);
+-----+
| id_article | judul
+-----+
|        9 | Mengenal Tipe Data Integer dalam Bahasa Pemrograman PHP
|       10 | Mengenal Tipe Data Float dalam Bahasa Pemrograman PHP
|        1 | Pengertian dan Fungsi PHP dalam Pemrograman Web
|        7 | Cara Memasukkan kode PHP ke dalam HTML
|        2 | Sejarah PHP dan Perkembangan Versi PHP dari masa ke masa
|        3 | Cara Menginstall dan Mendownload PHP dengan XAMPP
|        5 | Cara Menjalankan File PHP dengan XAMPP
|        8 | Aturan Dasar Penulisan Kode PHP
+-----+
```

Dalam query diatas, saya ingin mencari judul dengan kata ‘Integer’, selain itu judul yang berdekatan juga ikut tampil.

Ketika hasil pencarian menemukan judul dengan kata ‘Integer’ di dalamnya, kata-kata lain dari hasil tersebut akan dipakai untuk mencari hasil yang berdekatan. Judul lain yang mengandung kata ‘Mengenal’, ‘Tipe’, ‘Data’, ‘dalam’, ‘Bahasa’, ‘Pemrograman’ dan ‘PHP’ juga ikut dicari.

15.10 Multi-Column FULLTEXT Index

Fitur FULLTEXT index juga bisa diterapkan ke lebih dari satu kolom. Untuk contoh ini saya akan membuat tabel artikel2:

```
CREATE TABLE artikel12 (
    no TINYINT PRIMARY KEY AUTO_INCREMENT,
    judul VARCHAR(200),
    isi TEXT,
    FULLTEXT KEY judul_dan_isi (judul,isi)
);

INSERT INTO artikel12 (judul, isi) VALUES
('Pengertian PHP', 'PHP adalah bahasa pemrograman server-side...'),
('Sejarah PHP', 'Sejarah PHP bermula pada tahun 1994 ketika...'),
('Menginstall XAMPP', 'Kode PHP harus dijalankan dari web server...'),
('Web Server Apache', 'Untuk menjalankan web server Apache, pertama...'),
('Menjalankan XAMPP', 'Agar dapat mengakses halaman PHP dari web...'),
('Menjalankan PHP', 'Untuk menjalankan kode PHP, XAMPP harus...');
```

```
SELECT * FROM artikel12;
```

no	judul	isi
1	Pengertian PHP	PHP adalah bahasa pemrograman server-side...
2	Sejarah PHP	Sejarah PHP bermula pada tahun 1994 ketika...
3	Menginstall XAMPP	Kode PHP harus dijalankan dari web server...
4	Web Server Apache	Untuk menjalankan web server Apache, pertama...
5	Menjalankan XAMPP	Agar dapat mengakses halaman PHP dari web...
6	Menjalankan PHP	Untuk menjalankan kode PHP, XAMPP harus...

Tabel artikel12 memiliki 3 kolom, yakni no, judul dan isi. Kolom judul dan isi langsung di set sebagai FULLTEXT Index dengan nama judul_dan_isi. Index ini merupakan Multi-Column FULLTEXT Index yang mencakup kolom judul dan isi.

Sebagai contoh, saya ingin mencari kata ‘XAMPP’ di kolom judul dan isi:

```
SELECT * FROM artikel12 WHERE MATCH(judul,isi) AGAINST ('XAMPP');
```

no	judul	isi
3	Menginstall XAMPP	Kode PHP harus dijalankan dari web server...
5	Menjalankan XAMPP	Agar dapat mengakses halaman PHP dari web...
6	Menjalankan PHP	Untuk menjalankan kode PHP, XAMPP harus...

Urutan kolom pada saat pembuatan index juga mempengaruhi hasil pencarian. Karena saya menempatkan kolom judul terlebih dahulu, maka hasil pencarian akan diprioritaskan di kolom judul, baru kemudian ke kolom isi.

Dari hasil diatas, artikel 3 dan 5 tampil di posisi teratas karena kata ‘XAMPP’ ditemukan di bagian kolom judul. Untuk artikel 6, kata ‘XAMPP’ berada di kolom isi.

Ketiga mode FULLTEXT index juga berlaku untuk multiple-colom index. Berikut hasil query yang sama, hanya saja kali ini menggunakan mode *query expansion*:

```
SELECT * FROM artikel2 WHERE MATCH(judul,isi)
AGAINST ('XAMPP' WITH QUERY EXPANSION);
+-----+-----+
| no | judul           | isi
+-----+-----+
| 5 | Menjalankan XAMPP | Agar dapat mengakses halaman PHP dari web...
| 3 | Menginstall XAMPP | Kode PHP harus dijalankan dari web server...
| 6 | Menjalankan PHP  | Untuk menjalankan kode PHP, XAMPP harus...
| 4 | Web Server Apache | Untuk menjalankan web server Apache, pertama...
| 1 | Pengertian PHP    | PHP adalah bahasa pemrograman server-side...
| 2 | Sejarah PHP       | Sejarah PHP bermula pada tahun 1994 ketika...
+-----+-----+
```

Fitur FULLTEXT index sangat cocok dipakai untuk membuat pencarian di kolom yang memiliki banyak teks seperti isi artikel. Kemampuan untuk menampilkan hasil berdasarkan relevansi sangat meningkatkan kualitas pencarian.

Dalam bab ini kita telah membahas tentang cara pembuatan index beserta fungsinya. Index merupakan fitur penting dalam optimasi query. Membuat query yang bisa memanfaatkan index juga merupakan faktor penting untuk membuat query yang bisa diproses dengan cepat.

Dalam bab tentang menampilkan data tabel (query SELECT), kita sudah melihat bahwa ada banyak query yang bisa dipakai untuk menampilkan data. Jika anda ingin mengoptimalkan query-query ini, bisa digabung dengan index lalu buat percobaan query mana yang akan diproses lebih cepat untuk menghasilkan tampilan yang sama.

16. Transaction dan Table Lock

Transaction dan **table lock** merupakan fitur lanjutan yang tersedia di MySQL dan MariaDB. Dengan *transaction* kita bisa membatalkan query dan dengan *table lock* kita bisa mengunci tabel agar tidak bisa diakses user lain.

16.1 Pengertian Transaction

Transaction adalah fitur untuk mengeksekusi beberapa perintah (query) sebagai satu kesatuan. Ketika terjadi masalah atau hal lain, seluruh query bisa dibatalkan atau di proses seluruhnya.

Fitur *transaction* memastikan kumpulan query tersebut dieksekusi bersamaan atau tidak sama sekali. Ini akan mencegah terjadinya tabel yang diupdate sebagian karena query tersebut memiliki hubungan dengan query lain.

Contoh yang sering dipakai untuk menggambarkan perlunya *transaction* adalah proses transfer uang. Misalkan Rudi ingin mengirim 100.000 kepada Alex. Proses ini melihatkan 2 query, pertama uang Rudi di dalam tabel akan kita kurangkan sebesar 100.000, dan kedua nilai uang Alex akan ditambah 100.000.

Kedua proses ini harus dieksekusi bersamaan. Tanpa *transaction*, bisa saja setelah perintah pertama diproses, terjadi error atau server bermasalah. Akibatnya, uang Rudi sudah berkurang 100.000, tapi tidak sampai ke Alex.

Dengan *transaction*, kedua operasi tersebut akan menjadi satu unit. Jika terjadi masalah, seluruh query akan dikembalikan ke kondisi semula.

Fungsi lain dari *transaction* adalah untuk memastikan data yang sedang diproses tidak bisa diubah oleh user lain. Disini *transaction* berfungsi untuk mencegah user lain melihat atau mengubah data yang belum selesai di proses.

Di balik keunggulannya, *transaction* juga punya kelemahan, yakni butuh pemrosesan CPU, memory dan ruang harddisk yang lebih banyak daripada query biasa. Oleh karena itu tidak semua query perlu fitur *transaction*.

Perintah query untuk mencatat jumlah pengunjung website tidak perlu menggunakan *transaction*, namun transaksi perbankan yang butuh presisi tinggi perlu sebuah mekanisme *transaction* untuk memastikan semua data terintegrasi dengan baik.

Di dalam MySQL dan MariaDB, tidak semua *storage engine* bisa menggunakan *transaction*. Storage engine **InnoDB** mendukung *transaction*, sedangkan **MyISAM** dan **MEMORY** tidak mendukung *transaction*.

16.2 Prinsip Kerja Transaction

Di dalam MySQL, ada pengaturan yang bernama **autocommit mode**. Mode ini mengatur apakah setiap perintah query bisa langsung diproses atau tidak. Secara default mode ini aktif, efeknya setiap perintah query yang kita ketik ke dalam MySQL akan langsung diproses saat itu juga.

Transaction akan menonaktifkan *autocommit mode* sementara waktu. Sehingga query yang diketik tidak langsung diproses, tapi menunggu sampai perintah tertentu mengaktifkan kembali *autocommit mode*.

Untuk memulai *transaction*, atau yang tidak lain adalah mematikan *autocommit mode*, kita bisa menggunakan salah satu dari perintah berikut:

- **BEGIN**
- **START TRANSACTION**
- **SET autocommit = 0**

Setelah salah satu perintah ini dijalankan, semua query yang ditulis setelahnya akan diproses sebagai satu unit *transaction* dan belum permanen.

Untuk mengakhiri *transaction*, kita memiliki 2 buah pilihan akhir: apakah akan mempermanakan hasil tersebut atau membatalkan semuanya.

Agar hasil *transaction* menjadi permanen, bisa menggunakan perintah:

- **COMMIT**
- **SET autocommit = 1**

Khusus untuk query `SET autocommit = 1`, digunakan untuk memproses *transaction* yang diawali dengan perintah `SET autocommit = 0`.

Jika ingin membatalkan seluruh *transaction*, jalankan perintah:

- **ROLLBACK**

Query yang berjalan juga hanya diproses dalam session yang aktif saja. User lain atau session lain tidak bisa melihat hasil *transaction* yang belum selesai.

16.3 Cara Menjalankan Transaction

Mari kita mulai praktik membuat query *transaction*. Untuk keperluan ini saya akan menggunakan tabel tabungan. Perintah pembuatan tabel tabungan adalah sebagai berikut:

```

CREATE TABLE tabungan (
    no TINYINT PRIMARY KEY AUTO_INCREMENT,
    nama VARCHAR(50),
    proses ENUM('D', 'K'),
    jumlah DEC(10,2)
) ENGINE = InnoDB;

INSERT INTO tabungan VALUES
    (NULL, 'Rudi Permana', 'K', 100000),
    (NULL, 'Sandri Fatmala', 'D', 50000);

SELECT * FROM tabungan;
+----+-----+-----+-----+
| no | nama      | proses | jumlah |
+----+-----+-----+-----+
| 1  | Rudi Permana | K     | 100000.00 |
| 2  | Sandri Fatmala | D     | 50000.00 |
+----+-----+-----+-----+

```

Tabel tabungan memiliki 4 kolom:

- Kolom `no` berisi nomor urut, sekaligus berfungsi sebagai PRIMARY KEY. Kolom ini di set dengan AUTO_INCREMENT sehingga angka di dalamnya otomatis menaik mulai dari 1, 2, dst.
- Kolom `nama` berisi nama penabung.
- Kolom `proses` berisi salah satu dari huruf ‘D’ atau ‘K’. Dalam sistem perbankan, ‘D’ artinya **Debet**, yakni si penabung mengambil uangnya dari bank. Sedangkan ‘K’ berarti **Kredit**, dimana penabung menambahkan uangnya ke bank.
- Kolom `jumlah` berisi nominal uang yang akan diproses.

Di akhir query `CREATE TABLE`, saya menambahkan perintah `ENGINE = InnoDB`, sebab *transaction* hanya bisa berjalan di tipe tabel InnoDB. Perintah ini sebenarnya tidak perlu ditulis karena secara default seluruh tabel di dalam MySQL sudah menggunakan InnoDB.

START TRANSACTION, ROLLBACK, dan COMMIT

Sebuah *transaction* bisa diawali dengan perintah `START TRANSACTION`, `BEGIN`, atau `SET autocommit = 0`. *Transaction* nantinya diakhiri dengan perintah `ROLLBACK` untuk membatalkan *transaction* serta `COMMIT` atau `SET autocommit = 1` untuk membuat permanen hasil *transaction*.

Baik, mari kita mulai prakteknya:

```

START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

```

Perintah `START TRANSACTION` diatas akan memulai sebuah sesi *transaction*. Query yang diketik setelah perintah ini akan aktif “sementara”, yakni sampai *transaction* dinyatakan selesai.

Saya akan coba jalankan query `INSERT` dan `UPDATE` ke dalam tabel `tabungan`:

```
INSERT INTO tabungan VALUES (NULL, 'Rudi Permana', 'D', 50000);
Query OK, 1 row affected (0.06 sec)

UPDATE tabungan SET jumlah = '125000' WHERE nama = 'Sandri Fatmala';
Query OK, 1 row affected (0.06 sec)
Rows matched: 1 Changed: 1 Warnings: 0

SELECT * FROM tabungan;
+-----+-----+-----+
| no | nama | proses | jumlah |
+-----+-----+-----+
| 1 | Rudi Permana | K | 100000.00 |
| 2 | Sandri Fatmala | D | 125000.00 |
| 3 | Rudi Permana | D | 50000.00 |
+-----+-----+-----+
```

Kedua query berjalan sebagai mana mestinya. Hasil dari perintah SELECT memperlihatkan bahwa data Rudi Permana sudah masuk ke tabel tabungan, serta jumlah untuk Sandri Fatmala telah berubah dari 50000 ke 125000.

Namun saya memutuskan bahwa query diatas ada yang salah dan ingin membatalkannya. Untuk membatalkan *transaction*, jalankan perintah ROLLBACK:

```
ROLLBACK;
Query OK, 0 rows affected (0.06 sec)
```

```
SELECT * FROM tabungan;
+-----+-----+-----+
| no | nama | proses | jumlah |
+-----+-----+-----+
| 1 | Rudi Permana | K | 100000.00 |
| 2 | Sandri Fatmala | D | 50000.00 |
+-----+-----+-----+
```

Terlihat bahwa perintah ROLLBACK akan membatalkan query INSERT dan UPDATE yang kita tulis di dalam *transaction*. Isi tabel tabungan kembali ke bentuk semua sebelum *transaction* berjalan.

Hal yang sama juga terjadi seandainya saya langsung menutup jendela cmd atau mematikan paksa komputer. Seluruh query yang telah di jalankan di dalam *transaction* akan dibatalkan.

Akan tetapi jika di akhir *transaction* dijalankan perintah COMMIT, seluruh query akan menjadi permanen.

Mari kita ulangi query yang sama, tapi kali ini diakhiri dengan perintah COMMIT:

```
START TRANSACTION;

INSERT INTO tabungan VALUES (NULL, 'Rudi Permana', 'D', 50000);

UPDATE tabungan SET jumlah = '125000' WHERE nama = 'Sandri Fatmala';

COMMIT;

SELECT * FROM tabungan;
```

no	nama	proses	jumlah
1	Rudi Permana	K	100000.00
2	Sandri Fatmala	D	125000.00
4	Rudi Permana	D	50000.00

Dengan perintah COMMIT, saya menginstruksikan kepada MySQL bahwa seluruh query di dalam *transaction* bisa diterapkan secara permanen.

Terlihat juga data Rudi Permana akan menggunakan no = 4. Hal ini terjadi karena no = 3 sudah dipakai pada *transaction* pertama (yang akhirnya dibatalkan dengan perintah ROLLBACK). Karena angka AUTO_INCREMENT untuk kolom no = 3 sudah terpakai, MySQL akan lanjut ke angka berikutnya, yakni 4.

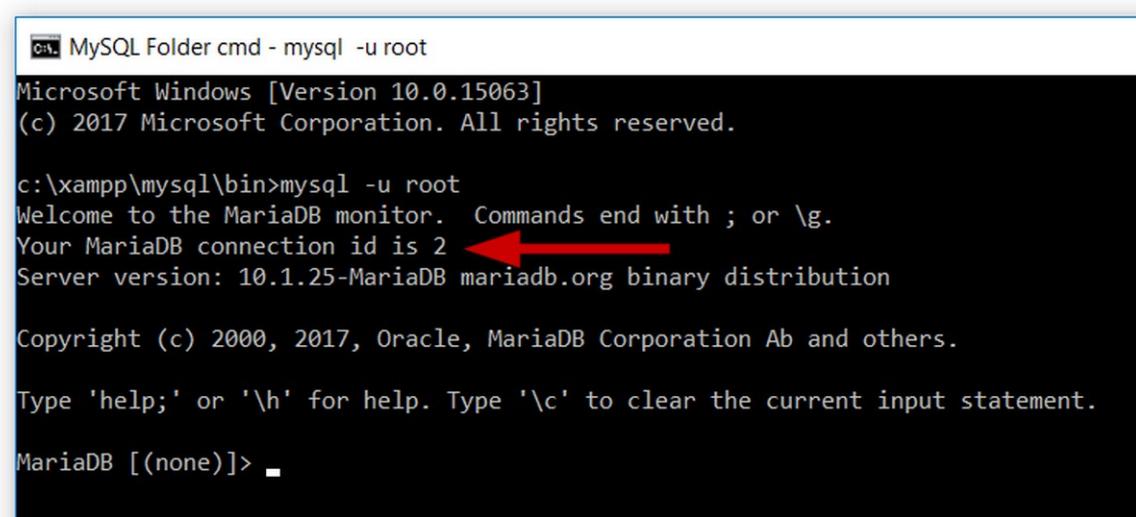
Efek Transaction pada Session Lain

Apakah anda masih ingat dengan pengertian *session* di dalam MySQL? Saya telah membahasnya sekilas di bab tentang **SQL Mode**.

Session atau sesi adalah sebutan untuk satu kali proses koneksi antara MySQL Client dengan MySQL Server.

Ketika kita membuka cmd Windows dan mengetik mysql -u root, itu akan membuka satu session. Di dalam teks “welcome” dari MySQL atau MariaDB bisa dilihat **connection id**, yakni identitas session yang sedang terhubung ke MySQL Server.

Sebagai contoh, dalam tampilan berikut session yang saya pakai memiliki *connection id* = 2:



The screenshot shows a Windows command prompt window titled "MySQL Folder cmd - mysql -u root". It displays the MySQL monitor welcome message, including the connection ID (2), and the server version (10.1.25-MariaDB mariadb.org binary distribution). A red arrow points to the connection ID value '2'.

Gambar: Membuat satu session dengan MySQL Server dengan connection id = 2

Nilai *connection id* di hasilkan secara otomatis oleh MySQL Server. Setiap MySQL client yang terhubung akan mendapatkan nilai *connection id* masing-masing.

Untuk melihat nilai *connection id*, bisa menjalankan query `SELECT CONNECTION_ID()`:

```
SELECT CONNECTION_ID();
+-----+
| CONNECTION_ID() |
+-----+
|          2 |
+-----+
```

Session akan berakhir ketika kita mengetik perintah `EXIT`, atau menutup paksa cmd Windows.

Sekarang, tanpa menutup jendela session pertama (*connection id* = 2), buka cmd baru dan login kembali sebagai user root:

```

MySQL Folder cmd - mysql -u root
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

c:\xampp\mysql\bin>mysql -u root
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 2
Server version: 10.1.25-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2017, Microsoft Corporation. All rights reserved.

MariaDB [(none)]> c:\xampp\mysql\bin>mysql -u root
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 3
Server version: 10.1.25-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>

```

Gambar: Dua buah session yang aktif bersamaan dengan *connection id* = 2 dan *connection id* = 3

Meskipun kedua jendela cmd menggunakan user yang sama (username = root). Tapi keduanya berada di session terpisah. Dalam tampilan diatas, satu jendela cmd menggunakan *connection id* = 2, dan satu lagi menggunakan *connection id* = 3.

Dalam praktek aslinya, setiap session diwakili oleh user yang berbeda-beda. Jika terdapat 10 MySQL client yang sedang mengakses database, setiap jendela akan mendapatkan *connection id* yang berlainan.

Ketika menjalankan *transaction*, session akan membatasi ruang lingkup dari query yang dieksekusi. Session lain tidak bisa melihat hasil *transaction* sebelum dinyatakan selesai. Entah itu dibatalkan dengan perintah ROLLBACK, atau diproses menggunakan perintah COMMIT.

Mari kita coba prakteknya.

Silahkan buka dua buah jendela cmd Windows dan masuk ke MySQL / MariaDB dengan user root. Artinya, terdapat 2 buah jendela cmd yang jalan bersamaan (seperti gambar diatas). Setiap jendela cmd menggunakan session MySQL yang berbeda.

Jendela cmd ini akan saya sebut sebagai **jendela pertama** dan **jendela kedua**.

Di dalam **jendela pertama**, jalankan perintah *transaction* berikut ini:

```

START TRANSACTION;

INSERT INTO tabungan VALUES (NULL, 'Rina Kumala Sari', 'K', 150000);

UPDATE tabungan SET jumlah = '275000' WHERE no = 1;

SELECT * FROM tabungan;
+-----+-----+-----+
| no | nama      | proses | jumlah   |
+-----+-----+-----+

```

```
+-----+-----+-----+
| 1 | Rudi Permana | K | 275000.00 |
| 2 | Sandri Fatmala | D | 125000.00 |
| 4 | Rudi Permana | D | 50000.00 |
| 5 | Rina Kumala Sari | K | 150000.00 |
+-----+-----+-----+
```

Setelah memulai *transaction* dengan perintah START TRANSACTION, saya menjalankan query INSERT dan UPDATE. Hasilnya, tabel tabungan akan bertambah menjadi 4 data, serta kolom jumlah pada baris pertama akan berubah menjadi 275000.

Di jendela kedua (session kedua), mari lihat isi tabel tabungan dengan query SELECT:

```
SELECT * FROM tabungan;
```

```
+-----+-----+-----+
| no | nama | proses | jumlah |
+-----+-----+-----+
| 1 | Rudi Permana | K | 100000.00 |
| 2 | Sandri Fatmala | D | 125000.00 |
| 4 | Rudi Permana | D | 50000.00 |
+-----+-----+-----+
```

The screenshot shows two separate MySQL command-line sessions. The left session (Session 1) is running under the 'belajar' database. It starts with a START TRANSACTION; statement, followed by an INSERT INTO tabungan VALUES (NULL, 'Rina Kumala Sari', 'K', 150000); statement which succeeds with 1 row affected. Then, an UPDATE tabungan SET jumlah = '275000' WHERE no = 1; statement is run, also succeeding with 1 row affected. A SELECT * FROM tabungan; statement is then run, showing the updated data with the first row having a jumlah value of 275000.00. The right session (Session 2) is also running under the 'belajar' database. It starts with a use belajar; statement, followed by a SELECT * FROM tabungan; statement which shows the original data with the first row having a jumlah value of 100000.00. This demonstrates that the changes made in Session 1 are not visible in Session 2 until a COMMIT or ROLLBACK is performed.

```
on Select MySQL Folder cmd - mysql -u root
MariaDB [belajar]> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

MariaDB [belajar]> INSERT INTO tabungan VALUES (NULL, 'Rina Kumala Sari', 'K', 150000);
Query OK, 1 row affected (0.03 sec)

MariaDB [belajar]> UPDATE tabungan SET jumlah = '275000' WHERE no = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [belajar]> SELECT * FROM tabungan; 1
+-----+-----+-----+
| no | nama | proses | jumlah |
+-----+-----+-----+
| 1 | Rudi Permana | K | 275000.00 |
| 2 | Sandri Fatmala | D | 125000.00 |
| 4 | Rudi Permana | D | 50000.00 |
| 5 | Rina Kumala Sari | K | 150000.00 |
+-----+-----+-----+
4 rows in set (0.00 sec)

MariaDB [belajar]>

on Select MySQL Folder cmd - mysql -u root
MariaDB [(none)]> use belajar;
Database changed
MariaDB [belajar]> SELECT * FROM tabungan; 2
+-----+-----+-----+
| no | nama | proses | jumlah |
+-----+-----+-----+
| 1 | Rudi Permana | K | 100000.00 |
| 2 | Sandri Fatmala | D | 125000.00 |
| 4 | Rudi Permana | D | 50000.00 |
+-----+-----+-----+
3 rows in set (0.00 sec)

MariaDB [belajar]>
```

Gambar: Hasil query **SELECT * FROM tabungan** di session yang berbeda

Disini tabel tabungan tetap terdiri dari 3 baris! Selain itu kolom jumlah untuk baris pertama juga masih 100000. Padahal di **jendela pertama** saya sudah menambah baris ke-4 dan mengubah baris pertama.

Inilah salah satu fitur dari *transaction*, yakni membatasi session lain agar tidak bisa mengubah query *transaction* yang belum selesai.

Baik, kembali ke **jendela pertama** dan jalankan COMMIT:

```
COMMIT;
Query OK, 0 rows affected (0.12 sec)
```

Jalankan kembali query SELECT di **jendela kedua**. Sekarang baru terlihat penambahan dan perubahan data di tabel tabungan hasil *transaction* dari **jendela pertama**.

Autocommit Mode

Cara lain untuk menjalankan *transaction* adalah dengan merubah settingan **autocommit mode**.

Jika kita menonaktifkan *autocommit mode* dengan perintah SET autocommit = 0, maka seluruh query yang di tulis dalam session tersebut akan berjalan sebagai *transaction*.

Perintah COMMIT dan ROLLBACK bisa dipakai untuk mempermanenkan hasil query, namun *transaction* kedua akan bersambung setelahnya (tidak perlu perintah START TRANSACTION atau BEGIN). Hal ini terus berlangsung hingga *autocommit mode* diaktifkan kembali dengan perintah SET autocommit = 1.

Silahkan pelajari sebentar perintah query berikut ini:

```
SELECT * FROM tabungan;
+-----+-----+-----+
| no | nama | proses | jumlah |
+-----+-----+-----+
| 1 | Rudi Permana | K | 275000.00 |
| 2 | Sandri Fatmala | D | 125000.00 |
| 4 | Rudi Permana | D | 50000.00 |
| 5 | Rina Kumala Sari | K | 150000.00 |
+-----+-----+-----+

SET autocommit = 0;

INSERT INTO tabungan VALUES (NULL, 'Ikhsan Prayoga', 'K', 750000);

DELETE FROM tabungan WHERE nama = 'Rudi Permana';

SELECT * FROM tabungan;
+-----+-----+-----+
| no | nama | proses | jumlah |
+-----+-----+-----+
| 2 | Sandri Fatmala | D | 125000.00 |
| 5 | Rina Kumala Sari | K | 150000.00 |
```

```
| 6 | Ikhsan Prayoga | K | 750000.00 |
+-----+-----+-----+
```

ROLLBACK;

SELECT * FROM tabungan;

```
+-----+-----+-----+
| no | nama | proses | jumlah |
+-----+-----+-----+
| 1 | Rudi Permana | K | 275000.00 |
| 2 | Sandri Fatmala | D | 125000.00 |
| 4 | Rudi Permana | D | 50000.00 |
| 5 | Rina Kumala Sari | K | 150000.00 |
+-----+-----+-----+
```

Setelah menampilkan isi tabel tabungan, saya mematikan *autocommit mode* dengan perintah `SET autocommit = 0`. Dengan demikian, kita sudah masuk ke sesi *transaction*.

Selanjutnya saya coba menambah data baru Ikhsan Prayoga ke dalam tabel, kemudian menghapus baris yang memiliki nama = Rudi Permana. Dari tampilan query `SELECT` terlihat bahwa nama Rudi Permana sudah terhapus serta nama Ikhsan Prayoga telah ditambahkan ke dalam tabel.

Saya kemudian berubah pikiran dan ingin membatalkan kedua query ini dengan perintah `ROLLBACK`. Terlihat nama Rudi Permana kembali tampil yang menandakan query *transaction* sukses di cancel.

Sampai disini, efek dari `SET autocommit = 0` masih berlaku. Akibatnya perintah query yang diketik setelah perintah `ROLLBACK` otomatis langsung berjalan sebagai sesi *transaction* selanjutnya:

UPDATE tabungan SET jumlah = '1550000' WHERE no = 1;

SELECT * FROM tabungan;

```
+-----+-----+-----+
| no | nama | proses | jumlah |
+-----+-----+-----+
| 1 | Rudi Permana | K | 1550000.00 |
| 2 | Sandri Fatmala | D | 125000.00 |
| 4 | Rudi Permana | D | 50000.00 |
| 5 | Rina Kumala Sari | K | 150000.00 |
+-----+-----+-----+
```

ROLLBACK;

SELECT * FROM tabungan;

```
+-----+-----+-----+
```

no	nama	proses	jumlah
1	Rudi Permana	K	275000.00
2	Sandri Fatmala	D	125000.00
4	Rudi Permana	D	50000.00
5	Rina Kumala Sari	K	150000.00

Pada awal kode diatas, saya menjalankan query UPDATE untuk mengubah kolom `jumlah` pada data no = 1 menjadi 1550000.

Sebagai pembuktian kita masih berada di sesi *transaction*, saya menjalankan kembali perintah ROLLBACK. Akibatnya, efek dari query UPDATE tersebut akan dibatalkan.

Kembali, setelah perintah ROLLBACK, sesi *transaction* berikutnya akan bersambung. Demikian seterusnya hingga *autocommit mode* kembali aktif.

Untuk mengaktifkan *autocommit mode* (kembali ke prilaku “normal”), bisa menjalankan query berikut:

```
SET autocommit = 1;
Query OK, 0 rows affected (0.00 sec)
```

Setelah *autocommit mode* kembali aktif, setiap perintah query akan langsung permanen sebagaimana biasanya.

SAVEPOINT Transaction

Transaction memiliki fitur SAVEPOINT yang bisa dipakai untuk menandakan sebuah posisi di dalam *transaction*. Penanda ini nantinya bisa dipakai oleh perintah ROLLBACK untuk membatalkan query hingga posisi SAVEPOINT tersebut (tidak harus membatalkan seluruh *transaction*)

Konsep ini ibarat kita main game. Tanpa SAVEPOINT, setiap kali karakter kalah atau mati, game harus diulang dari awal. Tapi kalau sudah sampai ke SAVEPOINT, game bisa bersambung dari tempat tersebut (tidak harus dari awal).

Berikut format dasar penulisan query SAVEPOINT:

```
SAVEPOINT nama_savepoint;
```

Untuk kembali ke posisi SAVEPOINT, bisa menggunakan perintah berikut:

```
ROLLBACK TO SAVEPOINT nama_savepoint;
```

Agar lebih mudah dipahami, langsung saja kita lihat contoh prakteknya. Silahkan jalankan query dibawah ini:

```
SELECT * FROM tabungan;
+-----+-----+-----+
| no | nama | proses | jumlah |
+-----+-----+-----+
| 1 | Rudi Permana | K | 275000.00 |
| 2 | Sandri Fatmala | D | 125000.00 |
| 4 | Rudi Permana | D | 50000.00 |
| 5 | Rina Kumala Sari | K | 150000.00 |
+-----+-----+-----+

BEGIN;

INSERT INTO tabungan VALUES (NULL, 'Husli Khairan', 'K', 180000);

SAVEPOINT pertama;

SELECT * FROM tabungan;
+-----+-----+-----+
| no | nama | proses | jumlah |
+-----+-----+-----+
| 1 | Rudi Permana | K | 275000.00 |
| 2 | Sandri Fatmala | D | 125000.00 |
| 4 | Rudi Permana | D | 50000.00 |
| 5 | Rina Kumala Sari | K | 150000.00 |
| 7 | Husli Khairan | K | 180000.00 |
+-----+-----+-----+

INSERT INTO tabungan VALUES (NULL, 'Ikhsan Prayoga', 'D', 52000);

SAVEPOINT kedua;

SELECT * FROM tabungan;
+-----+-----+-----+
| no | nama | proses | jumlah |
+-----+-----+-----+
| 1 | Rudi Permana | K | 275000.00 |
| 2 | Sandri Fatmala | D | 125000.00 |
| 4 | Rudi Permana | D | 50000.00 |
| 5 | Rina Kumala Sari | K | 150000.00 |
| 7 | Husli Khairan | K | 180000.00 |
| 8 | Ikhsan Prayoga | D | 52000.00 |
+-----+-----+-----+
```

Di dalam query diatas, saya membuat dua buah **SAVEPOINT**, yakni **SAVEPOINT** pertama dan **SAVEPOINT** kedua.

Dengan adanya **SAVEPOINT**, kita bisa memilih apakah ingin **ROLLBACK** ke posisi **SAVEPOINT**

pertama, atau SAVEPOINT kedua.

Jika kembali ke SAVEPOINT pertama, artinya nama Ikhsan Prayoga akan kembali dihapus, karena query INSERT untuk Ikhsan Prayoga baru dijalankan setelah SAVEPOINT pertama.

Mari kita coba:

```
ROLLBACK TO SAVEPOINT pertama;
```

```
SELECT * FROM tabungan;
```

no	nama	proses	jumlah
1	Rudi Permana	K	275000.00
2	Sandri Fatmala	D	125000.00
4	Rudi Permana	D	50000.00
5	Rina Kumala Sari	K	150000.00
7	Husli Khairan	K	180000.00

Sekarang kita kembali ke posisi ketika perintah SAVEPOINT pertama ditulis. Disini belum ada nama Ikhsan Prayoga.

Yang juga perlu diingat, kita masih berada di dalam *transaction*, oleh karena itu harus diakhiri dengan perintah COMMIT agar *transaction* bisa permanen:

```
COMMIT;
```

```
SELECT * FROM tabungan;
```

no	nama	proses	jumlah
1	Rudi Permana	K	275000.00
2	Sandri Fatmala	D	125000.00
4	Rudi Permana	D	50000.00
5	Rina Kumala Sari	K	150000.00
7	Husli Khairan	K	180000.00

Terdapat juga perintah RELEASE SAVEPOINT yang berfungsi untuk menghapus SAVEPOINT. Perintah ini baru diperlukan jika kita menjalankan sesi *transaction* yang cukup panjang.

Menghapus SAVEPOINT yang tidak perlu akan mengosongkan ruang memory yang dipakai oleh SAVEPOINT tersebut.

Berikut contoh penggunaannya:

```
BEGIN;

INSERT INTO tabungan VALUES (NULL, 'Bobby Permana', 'D', 180000);

SELECT * FROM tabungan;
+-----+-----+-----+
| no | nama | proses | jumlah |
+-----+-----+-----+
| 1 | Rudi Permana | K | 275000.00 |
| 2 | Sandri Fatmala | D | 125000.00 |
| 4 | Rudi Permana | D | 50000.00 |
| 5 | Rina Kumala Sari | K | 150000.00 |
| 7 | Husli Khairan | K | 180000.00 |
| 9 | Bobby Permana | D | 180000.00 |
+-----+-----+-----+

SAVEPOINT pertama;

UPDATE tabungan SET nama = 'Ikhsan Prayoga' WHERE no = 2;

SELECT * FROM tabungan;
+-----+-----+-----+
| no | nama | proses | jumlah |
+-----+-----+-----+
| 1 | Rudi Permana | K | 275000.00 |
| 2 | Ikhsan Prayoga | D | 125000.00 |
| 4 | Rudi Permana | D | 50000.00 |
| 5 | Rina Kumala Sari | K | 150000.00 |
| 7 | Husli Khairan | K | 180000.00 |
| 9 | Bobby Permana | D | 180000.00 |
+-----+-----+-----+

RELEASE SAVEPOINT pertama;
Query OK, 0 rows affected (0.00 sec)

RELEASE SAVEPOINT pertama;
ERROR 1305 (42000): SAVEPOINT pertama does not exist

ROLLBACK TO SAVEPOINT pertama;
ERROR 1305 (42000): SAVEPOINT pertama does not exist

COMMIT;

SELECT * FROM tabungan;
+-----+-----+-----+
| no | nama | proses | jumlah |

```

```
+-----+-----+-----+
| 1 | Rudi Permana | K | 275000.00 |
| 2 | Ikhsan Prayoga | D | 125000.00 |
| 4 | Rudi Permana | D | 50000.00 |
| 5 | Rina Kumala Sari | K | 150000.00 |
| 7 | Husli Khairan | K | 180000.00 |
| 9 | Bobby Permana | D | 180000.00 |
+-----+-----+-----+
```

Setelah berhasil menghapus SAVEPOINT pertama dengan perintah RELEASE SAVEPOINT pertama, saya coba jalankan kembali perintah tersebut. Hasilnya: *ERROR 1305 (42000): SAVEPOINT pertama does not exist*, karena tentu saja SAVEPOINT pertama sudah tidak ada lagi.

Error juga terjadi untuk perintah ROLLBACK TO SAVEPOINT pertama karena SAVEPOINT pertama sudah terhapus.

Implicit COMMIT

Selain menulis langsung perintah COMMIT untuk mem-permanenkan hasil *transaction*, terdapat beberapa query yang bisa membuat *transaction* ter-COMMIT secara otomatis.

Query tersebut adalah yang termasuk kelompok **Data definition language (DDL)**, yakni query yang berguna untuk membuat / mengupdate / menghapus tabel atau database, seperti CREATE TABLE, DROP TABLE, ALTER TABLE, dll. Daftar lengkapnya bisa kesini: [Statements That Cause an Implicit Commit¹](#):

Berikut contoh kasusnya:

```
SELECT * FROM tabungan;
+-----+-----+-----+
| no | nama | proses | jumlah |
+-----+-----+-----+
| 1 | Rudi Permana | K | 275000.00 |
| 2 | Ikhsan Prayoga | D | 125000.00 |
| 4 | Rudi Permana | D | 50000.00 |
| 5 | Rina Kumala Sari | K | 150000.00 |
| 7 | Husli Khairan | K | 180000.00 |
| 9 | Bobby Permana | D | 180000.00 |
+-----+-----+-----+
```

```
BEGIN;
```

```
DELETE FROM tabungan;
Query OK, 6 rows affected (0.03 sec)
```

¹<https://dev.mysql.com/doc/refman/5.7/en/implicit-commit.html>

```
SELECT * FROM tabungan;
Empty set (0.00 sec)

CREATE TABLE tabungan2 (
    a INT,
    b VARCHAR(5)
);
Query OK, 0 rows affected (0.26 sec)

ROLLBACK;
Query OK, 0 rows affected (0.00 sec)

SELECT * FROM tabungan;
Empty set (0.00 sec)
```

Di dalam *transaction*, saya menjalankan query `DELETE FROM tabungan`. Query `DELETE` yang tanpa kondisi `WHERE` akan menghapus seluruh isi tabel. Tapi karena kita masih di dalam *transaction*, seharusnya isi tabel tetap bisa dikembalikan dengan perintah `ROLLBACK`.

Akan tetapi, sebelum perintah `ROLLBACK`, saya menjalankan query `CREATE TABLE tabungan2`. Query ini akan menyebabkan terjadinya **implicit COMMIT**. Saat query `CREATE TABLE` diproses, *transaction* otomatis di-COMMIT, sehingga query `ROLLBACK` tidak lagi bisa mengembalikan isi tabel `tabungan`.

Agar lebih aman dan mencegah terjadinya **implicit COMMIT**, batasi perintah di dalam *transaction* hanya untuk query yang tergabung ke **Data Manipulation Language (DML)**, seperti query `SELECT`, `INSERT`, `UPDATE`, dan `DELETE`, yakni query yang berfungsi untuk memanipulasi data tabel, bukan memanipulasi struktur tabel.

Pembahasan mengenai **implicit COMMIT** ini menutup materi tentang *transaction*. Silahkan anda coba buat berbagai skenario lain, misalnya melibatkan beberapa tabel yang kemudian di `ROLLBACK`.

16.4 Table Lock

Aplikasi RDBMS seperti MySQL dan MariaDB memiliki sistem internal untuk mencegah terjadinya “bentrok data”. Contoh kasus adalah ketika 2 user mencoba mengubah data tabel pada saat yang bersamaan.

MySQL (dan juga MariaDB) mengatasi masalah ini dengan sistem **table lock**. Tabel akan dikunci sementara waktu bagi satu user hingga perintah tersebut selesai di eksekusi. Setelah itu baru perintah berikutnya dijalankan. Jika terdapat banyak proses yang datang, setiap query akan antri menunggu giliran.

Proses penguncian tabel ini dipengaruhi berbagai faktor, mulai dari query yang dieksekusi hingga tipe tabel (*storage engine*) yang dipakai.

Untuk jenis tabel **MyISAM**. Proses penguncian dilakukan di level tabel (*table lock*). Pada saat MySQL memproses query `UPDATE` dari seorang user, seluruh tabel akan dikunci untuk sementara

waktu. User lain tidak bisa melakukan query UPDATE hingga proses dari user pertama selesai dijalankan.

Untuk jenis tabel **InnoDB**, proses penguncian berlangsung di level baris (*row lock*). Berbeda dengan MyISAM, ketika seorang user memproses query UPDATE, yang terkunci hanya baris yang sedang diupdate saja, bukan keseluruhan tabel. User lain tetap bisa mengupdate baris lain di tabel yang sama.

Akibat perbedaan sistem ini, performa tabel InnoDB lebih bagus dalam lingkungan multi-user dibandingkan MyISAM, terutama untuk tabel yang banyak melakukan perubahan data. Di dalam InnoDB, user lain tetap bisa bekerja di tabel yang sama (dengan catatan harus di baris yang berbeda)

Perlu juga dijelaskan bahwa mekanisme lock ini terjadi hanya ketika query tersebut diproses. Untuk query UPDATE yang sederhana, mungkin hanya butuh waktu 0,05 detik. Proses penguncian akan berlangsung di waktu sepersekian detik ini.

Akan tetapi untuk aplikasi yang sibuk, dalam satu waktu bisa jadi terdapat ribuan user yang mengakses tabel pada saat yang bersamaan. Proses penguncian bisa membuat antrian query yang cukup panjang. Efeknya, user harus menunggu lama sampai querynya bisa dieksekusi.

Jenis query juga akan mempengaruhi apakah sebuah tabel boleh diakses atau tidak. Ketika sebuah tabel dikunci karena sedang berlangsung proses UPDATE, pembacaan data menggunakan query SELECT tetap bisa dilakukan.

Dalam prakteknya, proses penguncian tabel ini sudah berjalan secara otomatis. MySQL memiliki mekanisme internal sendiri untuk memutuskan kapan sebuah tabel harus dikunci (**lock**), dan kapan kunci tabel tersebut dibuka kembali (**unlock**).

Dalam kasus tertentu, ada kalanya kita butuh mengunci tabel secara manual. Misalkan pada saat proses maintenance server atau penambahan fitur. Proses update tabel dengan ratusan ribu baris butuh waktu, sehingga lebih baik mengunci keseluruhan tabel agar pengguna lain tidak bisa mengakses tabel yang belum selesai di update.

MySQL menyediakan query `LOCK TABLE` untuk mengunci tabel, dan query `UNLOCK TABLES` untuk membuka kunci tabel. Query `LOCK TABLE` juga terdiri dari 2 jenis, yakni `LOCK TABLE...READ` dan `LOCK TABLE...WRITE`.

LOCK TABLE ... READ

Query `LOCK TABLE...READ` dipakai untuk mengunci tabel karena ingin melakukan proses **pembacaan data** (read). Efek dari query ini, seluruh user hanya bisa menjalankan query `SELECT` untuk menampilkan data tabel, tapi tidak bisa menulis ke dalam tabel, termasuk kita sendiri sebagai user yang menjalankan query tersebut.

Proses penguncian akan terus berlangsung hingga tabel dibuka dengan perintah `UNLOCK TABLES`. Mari kita lihat prakteknya.

Saya kembali menggunakan tabel tabungan. Agar datanya seragam, saya akan input ulang isi tabel:

```
TRUNCATE TABLE tabungan;

INSERT INTO tabungan (nama, proses, jumlah) VALUES
('Rudi Permana', 'K', 100000),
('Sandri Fatmala', 'D', 50000),
('Rina Kumala Sari', 'K', 25000);

SELECT * FROM tabungan;
+-----+-----+-----+
| no | nama | proses | jumlah |
+-----+-----+-----+
| 1 | Rudi Permana | K | 100000.00 |
| 2 | Sandri Fatmala | D | 50000.00 |
| 3 | Rina Kumala Sari | K | 25000.00 |
+-----+-----+-----+
```

Sekarang, mari kita kunci tabel ini untuk proses pembacaan:

```
LOCK TABLE tabungan READ;
Query OK, 0 rows affected (0.00 sec)
```

```
SELECT * FROM tabungan;
+-----+-----+-----+
| no | nama | proses | jumlah |
+-----+-----+-----+
| 1 | Rudi Permana | K | 100000.00 |
| 2 | Sandri Fatmala | D | 50000.00 |
| 3 | Rina Kumala Sari | K | 25000.00 |
+-----+-----+-----+
```

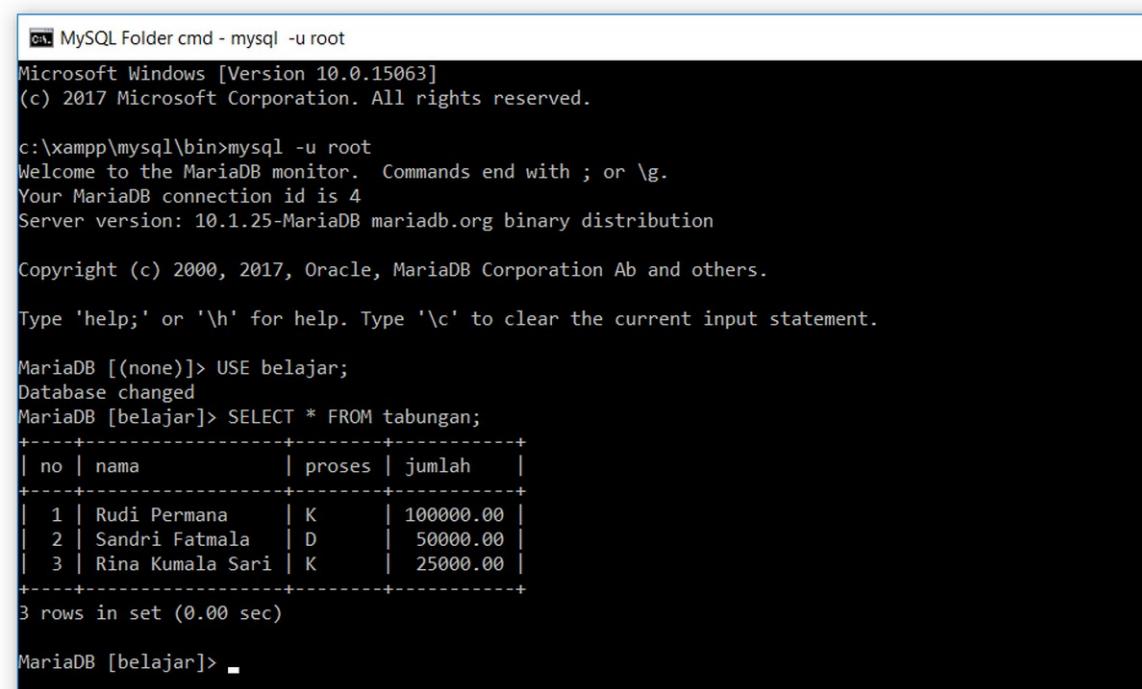
```
INSERT INTO tabungan VALUES (NULL, 'Sari Citra Lestari', 'D', 180000);
ERROR 1099 (HY000): Table 'tabungan' was locked with a READ lock
and cant be updated
```

```
UPDATE tabungan SET jumlah = '1550000' WHERE no = 2;
ERROR 1099 (HY000): Table 'tabungan' was locked with a READ lock
and cant be updated
```

Setelah proses penguncian dengan perintah **LOCK TABLE** tabungan **READ**, tabel tabungan akan terkunci dan hanya bisa diakses menggunakan query **SELECT** (untuk pembacaan data).

Query untuk mengubah data tabel akan ditolak, termasuk yang dibuat oleh user itu sendiri. Dalam contoh diatas saya mencoba menjalankan query **INSERT** dan **UPDATE**, hasilnya tampil pesan error karena tabel tabungan sudah di kunci dengan **READ lock**.

Hal yang sama juga terjadi ketika tabel ini diakses oleh user / session lain. Untuk pembuktian, silahkan buka jendela cmd lain, kemudian masuk sebagai user root dan akses tabel tabungan dengan query **SELECT**.



```

MySQL Folder cmd - mysql -u root
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

c:\xampp\mysql\bin>mysql -u root
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 4
Server version: 10.1.25-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> USE belajar;
Database changed
MariaDB [belajar]> SELECT * FROM tabungan;
+----+-----+-----+
| no | nama      | proses | jumlah |
+----+-----+-----+
| 1 | Rudi Permana | K     | 100000.00 |
| 2 | Sandri Fatmala | D     | 50000.00  |
| 3 | Rina Kumala Sari | K     | 25000.00  |
+----+-----+-----+
3 rows in set (0.00 sec)

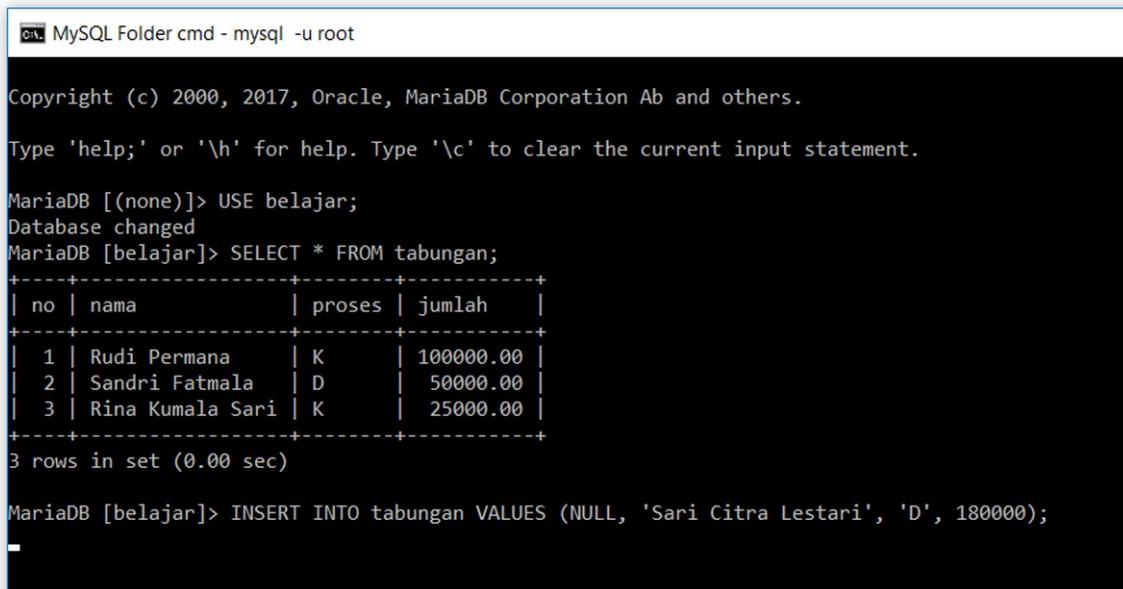
MariaDB [belajar]> -

```

Gambar: Mengakses tabel tabungan dari session lain

Hasilnya, tabel tabungan tetap bisa diakses.

Sekarang, mari kita coba jalankan query `INSERT` dari jendela cmd kedua ini:



```

MySQL Folder cmd - mysql -u root

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> USE belajar;
Database changed
MariaDB [belajar]> SELECT * FROM tabungan;
+----+-----+-----+
| no | nama      | proses | jumlah |
+----+-----+-----+
| 1 | Rudi Permana | K     | 100000.00 |
| 2 | Sandri Fatmala | D     | 50000.00  |
| 3 | Rina Kumala Sari | K     | 25000.00  |
+----+-----+-----+
3 rows in set (0.00 sec)

MariaDB [belajar]> INSERT INTO tabungan VALUES (NULL, 'Sari Citra Lestari', 'D', 180000);
-

```

Gambar: Query `INSERT` “tertahan”

Tidak ada pesan error yang keluar, namun cursor cmd seperti “tertahan”. Ini terjadi karena session MySQL sedang menunggu giliran untuk diproses. Akan tetapi proses tersebut tertahan di MySQL Server karena tabel tabungan sedang dikunci.

Dalam kondisi normal, proses menunggu ini biasanya hanya beberapa detik (bahkan kurang), yakni waktu yang dibutuhkan untuk memproses sebuah query. Tapi kali ini kita mengunci tabel

tabungan secara manual.

Sekarang, kembali ke jendela pertama, usahakan jendela cmd kedua tetap terlihat dan buka kunci tabel tabungan dengan query berikut:

```
UNLOCK TABLES;
Query OK, 0 rows affected (0.00 sec)
```

Sesaat setelah kunci tabel dibuka dari jendela cmd pertama (session pertama), query INSERT dari jendela kedua (session kedua) yang sebelumnya tertahan akan langsung di proses:

The screenshot shows two separate MySQL command-line sessions (cmd - mysql -u root).

Session 1 (Top Window):

```
MariaDB [belajar]> SELECT * FROM tabungan;
+----+-----+-----+
| no | nama | proses | jumlah |
+----+-----+-----+
| 1 | Rudi Permana | K | 100000.00 |
| 2 | Sandri Fatmala | D | 50000.00 |
| 3 | Rina Kumala Sari | K | 25000.00 |
+----+-----+-----+
3 rows in set (0.00 sec)

MariaDB [belajar]> INSERT INTO tabungan VALUES (NULL, 'Sari Citra Lestari', 'D', 180000);
ERROR 1099 (HY000): Table 'tabungan' was locked with a READ lock and can't be updated
MariaDB [belajar]> UPDATE tabungan SET jumlah = '1550000' WHERE no = 2;
ERROR 1099 (HY000): Table 'tabungan' was locked with a READ lock and can't be updated
MariaDB [belajar]> UNLOCK TABLES;
Query OK, 0 rows affected (0.00 sec)
```

A red circle labeled "1" is positioned over the error message in Session 1.

Session 2 (Bottom Window):

```
| no | nama | proses | jumlah |
+----+-----+-----+
| 1 | Rudi Permana | K | 100000.00 |
| 2 | Sandri Fatmala | D | 50000.00 |
| 3 | Rina Kumala Sari | K | 25000.00 |
+----+-----+-----+
3 rows in set (0.00 sec)

MariaDB [belajar]> INSERT INTO tabungan VALUES (NULL, 'Sari Citra Lestari', 'D', 180000);
Query OK, 1 row affected (5 min 40.54 sec) 2
MariaDB [belajar]>
```

A red circle labeled "2" is positioned over the successful query output in Session 2.

Gambar: Query INSERT di jendela cmd kedua bisa diproses

Perhatikan feedback dari query INSERT tersebut: *Query OK, 1 row affected (5 min 40.54 sec)*, artinya query INSERT tersebut butuh waktu selama 5 menit 40 detik agar bisa diproses. Waktu yang lama ini disebabkan karena saya baru membuka kunci tabel 5 menit kemudian. Inilah efek dari proses penguncian tabel dengan query `LOCK TABLE...READ`.

LOCK TABLE ... WRITE

Jenis kedua penguncian tabel adalah `LOCK TABLE...WRITE`. Perintah ini akan mengunci tabel untuk **proses penulisan** (write). User yang mengunci tabel mendapat akses exclusive untuk tabel tersebut. User lain tidak akan bisa mengakses, termasuk sekedar membaca tabel menggunakan query `SELECT`.

Mari kita lihat prakteknya. Saya akan mengunci tabel tabungan untuk proses penulisan:

```
LOCK TABLE tabungan WRITE;
Query OK, 0 rows affected (0.00 sec)
```

```
SELECT * FROM tabungan;
```

no	nama	proses	jumlah
1	Rudi Permana	K	100000.00
2	Sandri Fatmala	D	50000.00
3	Rina Kumala Sari	K	25000.00
4	Sari Citra Lestari	D	180000.00

```
INSERT INTO tabungan VALUES (NULL, 'James Situmorang', 'K', 325000);
Query OK, 1 row affected (0.07 sec)
```

```
SELECT * FROM tabungan;
```

no	nama	proses	jumlah
1	Rudi Permana	K	100000.00
2	Sandri Fatmala	D	50000.00
3	Rina Kumala Sari	K	25000.00
4	Sari Citra Lestari	D	180000.00
5	James Situmorang	K	325000.00

Terlihat, saya bisa menjalankan baik query SELECT maupun INSERT ke dalam tabel tabungan yang dikunci untuk proses penulisan (**WRITE lock**).

Mari kita lihat hasilnya dari jendela cmd kedua (session kedua):

```
MySQL Folder cmd - mysql -u root
MariaDB [belajar]> INSERT INTO tabungan VALUES (NULL, 'Sari Citra Lestari', 'D', 180000);
Query OK, 1 row affected (5 min 40.54 sec)

MariaDB [belajar]> SELECT * FROM tabungan;
```

Gambar: Query SELECT tertahan di jendela cmd kedua

Kali ini query SELECT tidak mendapat hak akses untuk sekedar melihat isi dari tabel tabungan dan cursor cmd kembali tertahan.

Akses kembali jendela cmd pertama (session pertama), dan buka kunci tabel:

```
UNLOCK TABLES;
Query OK, 0 rows affected (0.00 sec)
```

Pada saat yang bersamaan, barulah di jendela cmd kedua tampil hasil query SELECT yang sebelumnya tertahan:

The screenshot shows two separate MySQL command-line sessions. The left window (session 1) displays the results of a SELECT query on the 'tabungan' table, showing five rows of data. The right window (session 2) shows the same SELECT query being run again, but it is still processing, indicated by the message '5 rows in set (4 min 0.80 sec)'.

```
MariaDB [belajar]> SELECT * FROM tabungan;
+----+-----+-----+
| no | nama | proses | jumlah |
+----+-----+-----+
| 1 | Rudi Permana | K | 100000.00 |
| 2 | Sandri Fatmala | D | 50000.00 |
| 3 | Rina Kumala Sari | K | 25000.00 |
| 4 | Sari Citra Lestari | D | 180000.00 |
| 5 | James Situmorang | K | 325000.00 |
+----+-----+-----+
5 rows in set (0.00 sec)

MariaDB [belajar]> UNLOCK TABLES; ①
Query OK, 0 rows affected (0.00 sec)

MariaDB [belajar]>
```



```
MariaDB [belajar]> SELECT * FROM tabungan; ②
+----+-----+-----+
| no | nama | proses | jumlah |
+----+-----+-----+
| 1 | Rudi Permana | K | 100000.00 |
| 2 | Sandri Fatmala | D | 50000.00 |
| 3 | Rina Kumala Sari | K | 25000.00 |
| 4 | Sari Citra Lestari | D | 180000.00 |
| 5 | James Situmorang | K | 325000.00 |
+----+-----+-----+
5 rows in set (4 min 0.80 sec)

MariaDB [belajar]> ■
```

Gambar: Query SELECT dari jendela cmd kedua baru diproses setelah kunci dibuka

Inilah efek dari penguncian tabel menggunakan query `LOCK TABLE... WRITE`, dimana tabel akan dikunci agar tidak bisa diakses oleh pengguna lain hingga kunci tabel dibuka.

16.5 Multiple Table Lock

Ketika kita mengunci sebuah tabel, MySQL juga membatasi hak akses dari user yang bersangkutan untuk mengakses tabel lain yang tidak terkunci. Berikut percobaannya:

```
LOCK TABLE tabungan READ;
Query OK, 0 rows affected (0.00 sec)
```

```
SELECT * FROM mahasiswa;
ERROR 1100 (HY000): Table 'mahasiswa' was not locked with LOCK TABLES
```

```
SELECT * FROM universitas;
ERROR 1100 (HY000): Table 'universitas' was not locked with LOCK TABLES
```

```
UNLOCK TABLES;
Query OK, 0 rows affected (0.00 sec)
```

Setelah mengunci tabel tabungan, saya mencoba mengakses tabel mahasiswa dan tabel universitas, hasilnya terjadi error karena MySQL tidak mengizinkan kita mengakses tabel lain di luar tabel yang sudah terkunci.

Apabila kita juga ingin mengakses tabel tersebut, tabel mahasiswa dan universitas juga harus dikunci terlebih dahulu.

Berikut percobaannya:

```
LOCK TABLE tabungan READ;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
LOCK TABLE mahasiswa READ;
```

```
Query OK, 0 rows affected (0.04 sec)
```

```
LOCK TABLE universitas READ;
```

```
Query OK, 0 rows affected (0.03 sec)
```

```
SELECT * FROM tabungan;
```

```
ERROR 1100 (HY000): Table 'tabungan' was not locked with LOCK TABLES
```

```
SELECT * FROM mahasiswa;
```

```
ERROR 1100 (HY000): Table 'mahasiswa' was not locked with LOCK TABLES
```

```
SELECT * FROM universitas;
```

jurusan	tgl_berdiri	nama_dekan	jumlah_mhs	akr
Kimia	1987-07-12	Prof. Mulyono, M.Sc.	662	B
Ilmu Komputer	2003-02-23	Dr. Syahrial, M.Kom.	412	A
...

```
UNLOCK TABLES;
```

```
Query OK, 0 rows affected (0.00 sec)
```

Apa yang terjadi? Ternyata query LOCK TABLE juga tidak boleh ditulis terpisah. Jika ini yang ditulis, query LOCK TABLE otomatis akan membuka kunci tabel yang sebelumnya telah dikunci (**implicit unlock**).

Proses penguncian untuk banyak tabel harus ditulis dalam satu kali pemanggilan query LOCK TABLE, dengan format sebagai berikut:

```
LOCK TABLE nama_tabel1 level_akses1, nama_tabel2 level_akses2, ...;
```

Mari kita coba:

```
LOCK TABLE tabungan READ, mahasiswa READ, universitas WRITE;
Query OK, 0 rows affected (0.00 sec)
```

```
SELECT * FROM tabungan;
```

no	nama	proses	jumlah
1	Rudi Permana	K	100000.00
2	Sandri Fatmala	D	50000.00
.

```
SELECT * FROM mahasiswa;
```

nama	asal	kel	tinggi	jurusan	nilai_uan
Riana Putria	Padang	P	155	Kimia	339.20
Rudi Permana	Bandung	L	163	Ilmu Komputer	290.44
...

```
SELECT * FROM universitas;
```

jurusan	tgl_berdiri	nama_dekan	jum_mhs	akr
Kimia	1987-07-12	Prof. Mulyono, M.Sc.	662	B
Ilmu Komputer	2003-02-23	Dr. Syahrial, M.Kom.	412	A
...

```
UNLOCK TABLES;
```

```
Query OK, 0 rows affected (0.00 sec)
```

Sekarang, kita bisa mengakses tabel tabungan, mahasiswa, dan universitas karena ketiganya sudah terkunci.

Ini pula yang menjadi alasan perintah untuk membuka kunci tabel berbentuk jamak, yakni **UNLOCK TABLES**, bukan **UNLOCK TABLE**. Tambahan huruf ‘S’ diakhir kata **TABLES** menegaskan bahwa bisa jadi terdapat lebih dari satu tabel. Hal ini juga berarti kita tidak bisa membuka satu-satu kunci tabel, tapi harus langsung sekaligus.

Materi mengenai **transaction** dan **lock table** mungkin tidak akan sering kita gunakan. Akan tetapi dengan mengetahui bahwa fitur ini “ada” di dalam MySQL akan mempersiapkan kita jika suatu saat butuh mengembangkan aplikasi yang memerlukan **transaction** dan penguncian tabel.

17. Primary Key dan Referential Integrity

Dalam bab ini kita akan mempelajari apa peranan **primary key** dalam pembuatan tabel *relational*. Selain itu akan dibahas juga tentang **foreign key** serta konsep **referential integrity**.

Materi ini menjadi pondasi dasar dalam pembuatan database yang terdiri dari beberapa tabel. Query `SELECT...JOIN` yang sudah kita pelajari merupakan penerapan dari konsep **primary key** dan **foreign key**.

17.1 Pengertian Primary Key

Primary key adalah sebuah kolom (atau beberapa kolom) yang bisa mengidentifikasi setiap baris di dalam tabel. Karena pengertiannya seperti ini, kolom yang diset sebagai **primary key** harus memiliki nilai yang unik, yakni tidak boleh terdapat data berulang di kolom tersebut.

Dalam prakteknya, kita bisa saja membuat kolom apapun sebagai **primary key**, namun ada kolom yang cocok digunakan, ada yang tidak.

Kolom yang berisi nama mahasiswa misalnya, tidak cocok untuk menjadi **primary key**. Karena sangat mungkin ada mahasiswa yang bernama sama. Hasilnya, nama mahasiswa yang sama tidak bisa diinput lagi ke dalam tabel.

NIM (Nomor Induk Mahasiswa), bisa menjadi kandidat paling pas untuk **primary key**, karena tidak akan ada nomor induk mahasiswa yang berulang. Jika terjadi, berarti ada yang salah di sistem pembuatan angka NIM.

NIK (Nomor Induk Kependudukan) yang ada di KTP juga merupakan nilai yang cocok untuk **primary key**. Seharusnya tidak mungkin ada penduduk yang memiliki NIK sama.

Setiap tabel hanya bisa memiliki satu kolom sebagai **primary key**. Di dalam MySQL, kolom yang diset sebagai **primary key** juga otomatis akan di-index.

17.2 Membuat Primary Key

Terdapat beberapa cara untuk membuat **primary key**. Cara pertama adalah dengan menambahkan atribut `PRIMARY KEY` pada saat pembuatan tabel (query `CREATE TABLE`). Atribut ini ditulis setelah pendefinisian tipe data kolom. Berikut contoh penggunaannya:

```
DROP TABLE IF EXISTS mahasiswa;

CREATE TABLE mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20)
);

DESC mahasiswa;
+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| nim   | char(8) | NO | PRI | NULL |       |
| nama  | varchar(50) | YES |     | NULL |       |
| asal  | varchar(50) | YES |     | NULL |       |
| jurusan | varchar(20) | YES |     | NULL |       |
+-----+-----+-----+-----+-----+
```

Di baris paling awal, saya menggunakan query `DROP TABLE IF EXISTS mahasiswa` karena kita sudah pernah membuat tabel `mahasiswa` sebelumnya. Query ini akan menghapus tabel tersebut.

Pada query `CREATE TABLE`, saya membuat kolom `nim` sebagai *primary key* dengan menulis `nim CHAR(8) PRIMARY KEY`.

Dari hasil perintah `DESC mahasiswa`, terlihat di kolom `Key` berkode `PRI` untuk baris `nim`. Artinya kolom `nim` sudah berhasil di set sebagai *primary key*.

Ketika sebuah kolom diset sebagai *primary key*, kolom tersebut juga akan dibuatkan indexnya. Mari kita periksa:

```
SHOW INDEX FROM mahasiswa \G
*****
 1. row ****
      Table: mahasiswa
      Non_unique: 0
      Key_name: PRIMARY
      Seq_in_index: 1
      Column_name: nim
      Collation: A
      Cardinality: 0
      Sub_part: NULL
      Packed: NULL
      Null:
      Index_type: BTREE
      Comment:
Index_comment:
```

Hasilnya, tabel `mahasiswa` memiliki index dengan nama `PRIMARY` untuk kolom `nim`. Ini terlihat dari nilai `Table`, `Key_name`, dan `Column_name`.

Cara lain untuk membuat *primary key* adalah dengan menulisnya setelah pendefinisian semua kolom, seperti contoh berikut:

```
DROP TABLE IF EXISTS mahasiswa;

CREATE TABLE mahasiswa (
    nim CHAR(8),
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20),
    PRIMARY KEY (nim)
);

DESC mahasiswa;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| nim   | char(8)   | NO   | PRI | NULL    |       |
| nama  | varchar(50) | YES  |     | NULL    |       |
| asal  | varchar(50) | YES  |     | NULL    |       |
| jurusan | varchar(20) | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
```

Disini pendefinisian kolom `nim` ditulis di akhir query `CREATE TABLE`. Cara ini mirip seperti pembuatan index.

Pada saat pembuatan kolom *primary key*, MySQL tidak melarang kita membuat nama index untuk kolom tersebut, namun nama ini akan diabaikan dan diganti menjadi `PRIMARY`, seperti contoh berikut:

```
DROP TABLE IF EXISTS mahasiswa;

CREATE TABLE mahasiswa (
    nim CHAR(8),
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20),
    PRIMARY KEY nim_mhs (nim)
);

SHOW INDEX FROM mahasiswa \G
***** 1. row *****
  Table: mahasiswa
  Non_unique: 0
```

```
Key_name: PRIMARY
Seq_in_index: 1
Column_name: nim
Collation: A
Cardinality: 0
Sub_part: NULL
Packed: NULL
Null:
Index_type: BTREE
Comment:
Index_comment:
```

Saya menulis PRIMARY KEY nim_mhs (`nim`) dengan harapan nama index yang dipakai adalah `nim_mhs`. Namun dari hasil query SHOW INDEX terlihat bahwa nama index (`Key_name`) adalah PRIMARY.

Selain membuat *primary key* dari query CREATE TABLE, kita juga bisa menggunakan query ALTER TABLE. Ini dipakai jika tabel sudah terlanjur dibuat sebelumnya:

```
DROP TABLE IF EXISTS mahasiswa;

CREATE TABLE mahasiswa (
    nim CHAR(8),
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20)
);

ALTER TABLE mahasiswa ADD PRIMARY KEY (nim);
```

Setelah membuat tabel `mahasiswa`, saya menggunakan query ALTER TABLE untuk men-set kolom `nim` menjadi *primary key*.

Yang juga harus diingat, jika tabel `mahasiswa` ini sudah berisi data, nilai untuk kolom `nim` harus memenuhi syarat sebagai *primary key*, yakni tidak boleh ada data yang sama. Jika di kolom `nim` sudah terlanjur berisi data yang berulang, query ALTER akan menghasilkan error.

17.3 Menghapus Primary Key

Untuk menghapus *primary key*, kita bisa menggunakan query ALTER TABLE dengan format sebagai berikut:

```
ALTER TABLE nama_tabel DROP PRIMARY KEY;
```

Dalam query penghapusan diatas, kita tidak perlu menulis kolom apa yang bertindak sebagai *primary key*. MySQL bisa menemukan kolom tersebut karena di dalam sebuah tabel hanya boleh terdapat 1 kolom yang bertindak sebagai *primary key*.

Sebagai contoh, untuk menghapus *primary key* dari tabel `mahasiswa`, querynya adalah sebagai berikut:

```
ALTER TABLE mahasiswa DROP PRIMARY KEY;
Query OK, 0 rows affected (0.71 sec)
Records: 0 Duplicates: 0 Warnings: 0

DESC mahasiswa;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| nim   | char(8)   | NO   |     | NULL    |       |
| nama  | varchar(50) | YES  |     | NULL    |       |
| asal  | varchar(50) | YES  |     | NULL    |       |
| jurusan | varchar(20) | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
```

Terlihat atribut PRI sudah tidak ada lagi di baris `nim` untuk kolom Key.

Akan tetapi kolom `nim` tetap memiliki atribut NOT NULL yang sudah menjadi bawaan dari kolom *primary key*. Tentu saja kita juga bisa mengubahnya dengan query `ALTER TABLE` apabila dikehendaki.

17.4 Sifat Primary Key

Sifat utama dari kolom yang di set sebagai *primary key* adalah tidak bisa diinput data yang sama serta juga tidak bisa diisi dengan nilai NULL. Berikut percobaannya:

```
DROP TABLE IF EXISTS mahasiswa;

CREATE TABLE mahasiswa (
  nim CHAR(8) PRIMARY KEY,
  nama VARCHAR(50),
  asal VARCHAR(50),
  jurusan VARCHAR(20)
);

INSERT INTO mahasiswa VALUES
('17090113', 'Riana Putria', 'Padang', 'Kimia'),
('17140143', 'Rudi Permana', 'Bandung', 'Ilmu Komputer');
```

```
INSERT INTO mahasiswa VALUES
('17140143', 'Sari Citra Lestari', 'Jakarta', 'Manajemen');
-- ERROR 1062 (23000): Duplicate entry '17140143' for key 'PRIMARY'

INSERT INTO mahasiswa VALUES
(NULL, 'Sari Citra Lestari', 'Jakarta', 'Manajemen');
-- ERROR 1048 (23000): Column 'nim' cannot be null
```

Setelah membuat tabel `mahasiswa` dengan kolom `nim` sebagai *primary key*, saya mencoba menginput data lain yang kebetulan bernali sama. Hasilnya akan tampil pesan error. Hal yang sama juga terjadi saat diinput nilai `NULL` ke dalam kolom `nim`.

17.5 Composite Primary Key

Primary key juga bisa beranggotakan lebih dari satu kolom. Ini dikenal sebagai **composite primary key**.

Untuk membuat *composite primary key*, tulis daftar nama kolom yang akan menjadi *primary key* di dalam tanda kurung saat pendefinisian PRIMARY KEY. Berikut contohnya:

```
DROP TABLE IF EXISTS mahasiswa;

CREATE TABLE mahasiswa (
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20),
    PRIMARY KEY (nama, asal)
);

DESC mahasiswa;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key   | Default | Extra |
+-----+-----+-----+-----+-----+
| nama  | varchar(50) | NO   | PRI   | NULL    |       |
| asal  | varchar(50) | NO   | PRI   | NULL    |       |
| jurusan | varchar(20) | YES  |       | NULL    |       |
+-----+-----+-----+-----+-----+
```

Perhatikan cara pendefinisian PRIMARY KEY, saya menulis `PRIMARY KEY (nama, asal)`, artinya kolom `nama` dan `asal` secara bersama-sama akan menjadi *primary key*. Hasil dari query DESC `mahasiswa` juga memperlihatkan bahwa baris `nama` dan `asal` berisi nilai `PRI` di kolom Key.

Aturan bahwa nilai untuk kolom *primary key* harus unik (tidak boleh memiliki nilai yang sama) tetap berlaku. Hanya saja sekarang perhitungannya melibatkan kedua kolom. Mahasiswa yang bernama sama tetap bisa diinput, selama berasal dari kota yang berbeda.

Mari kita lihat:

```
INSERT INTO mahasiswa VALUES ('Riana Putria', 'Padang', 'Kimia');
Query OK, 1 row affected (0.07 sec)
```

```
INSERT INTO mahasiswa VALUES ('Riana Putria', 'Bandung', 'Kimia');
Query OK, 1 row affected (0.20 sec)
```

```
INSERT INTO mahasiswa VALUES ('Rudi Permana', 'Bandung', 'Akuntansi');
Query OK, 1 row affected (0.07 sec)
```

```
INSERT INTO mahasiswa VALUES ('Rudi Permana', 'Bandung', 'Ilmu Komputer');
ERROR 1062 (23000): Duplicate entry 'Rudi Permana-Bandung' for key 'PRIMARY'
```

Composite primary key biasa di pakai untuk tabel yang tidak memiliki kolom dengan nilai unik. Dengan menggunakan beberapa kolom sebagai *primary key*, diharapkan bisa mengeliminasi kebutuhan kolom tambahan.

Sebagai contoh, dalam tabel `mahasiswa` diatas, kecil kemungkinan terdapat nama siswa yang sama dan berasal dari kota yang sama. Namun ini kembali lagi ke desain database yang kita rancang, apakah perlu menambah satu kolom baru untuk menampung nilai NIM, atau cukup menjadikan beberapa kolom sebagai *primary key*.

17.6 Primary Key dengan Index Prefix

Dalam bab tentang index, kita pernah membahas tentang **index prefix**, yakni menjadikan beberapa karakter awal kolom sebagai index. Hal yang sama juga bisa diterapkan untuk *primary key*.

Berikut contoh penggunaannya:

```
DROP TABLE IF EXISTS mahasiswa;

CREATE TABLE mahasiswa (
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20),
    PRIMARY KEY (nama(3))
);
```

Baris `PRIMARY KEY (nama(3))` artinya saya ingin menjadikan kolom `nama` sebagai *primary key*, namun hanya 3 karakter awal saja yang di index.

Efek dari *index prefix* seperti ini, perhitungan sama tidaknya sebuah nilai hanya dilakukan berdasarkan 3 karakter awal dari kolom `nama`. Mari kita coba:

```
INSERT INTO mahasiswa VALUES ('Riana Putria', 'Padang', 'Kimia');
Query OK, 1 row affected (0.06 sec)

INSERT INTO mahasiswa VALUES ('Rianti Kusuma', 'Bogor', 'Akuntansi');
-- ERROR 1062 (23000): Duplicate entry 'Ria' for key 'PRIMARY'
```

Disini, meskipun nama Riana Putria dan Rianti Kusuma berbeda, akan tetapi yang diperhitungkan hanya tiga karakter awal saja, yakni Ria. Nama Rianti Kusuma akan tertolak karena bentrok dengan karakter Ria dari nama Riana Putria.

Apabila tanpa *index prefix*, kedua nilai ini akan dianggap berbeda.

17.7 Primary Key dengan Auto Increment

Kombinasi kolom *auto increment* dan *primary key* sangat sering dipakai. Dengan menggabungkan keduanya, kita tidak perlu khawatir terdapat data berulang karena nilai kolom akan di generate secara otomatis oleh MySQL.

Berikut contoh penggunaannya:

```
DROP TABLE IF EXISTS mahasiswa;

CREATE TABLE mahasiswa (
    no INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20)
);
```

Sekarang, kita bisa menginput nilai 0, NULL atau bahkan tidak menginput nilai apapun ke kolom *primary key*. MySQL akan membuat nomor urut secara otomatis:

```
INSERT INTO mahasiswa VALUES (0, 'Riana Putria', 'Padang', 'Kimia');
Query OK, 1 row affected (0.18 sec)

INSERT INTO mahasiswa VALUES
(NULL, 'Rudi Permana', 'Bandung', 'Ilmu Komputer');
Query OK, 1 row affected (0.07 sec)

INSERT INTO mahasiswa (nama, asal, jurusan) VALUES
('Sari Citra Lestari', 'Jakarta', 'Manajemen');
Query OK, 1 row affected (0.04 sec)

SELECT * FROM mahasiswa;
+-----+-----+-----+
```

no	nama	asal	jurusan
1	Riana Putria	Padang	Kimia
2	Rudi Permana	Bandung	Ilmu Komputer
3	Sari Citra Lestari	Jakarta	Manajemen

Teknik menggabungkan *primary key* dengan AUTO_INCREMENT seperti ini sering dipakai untuk tabel yang tidak memiliki kolom dengan nilai unik (syarat sebagai *primary key*). Daripada menggunakan *composite primary key*, menambah kolom baru sebagai AUTO_INCREMENT lebih mudah dan praktis.

17.8 Pengertian Referential Integrity dan Foreign Key

Referential Integrity adalah sebuah konsep hubungan antar tabel yang tujuannya untuk memastikan data di dalam kedua tabel tetap valid dan konsisten.

Agar bisa memahami pengertian ini, mari kita bahas menggunakan tabel mahasiswa dan universitas. Perhatikan isi kedua tabel berikut:

```
SELECT * FROM universitas;
```

jurusan	tgl_berdiri	nama_dekan	jumlah_mhs	akr
Kimia	1987-07-12	Prof. Mulyono, M.Sc.	662	B
Ilmu Komputer	2003-02-23	Dr. Syahrial, M.Kom.	412	A
Akuntansi	1985-03-19	Maya Fitrianti, M.M.	895	B
Farmasi	1997-05-30	Prof. Silvia Nst, M.Farm.	312	C
Fisika	1989-12-10	Dr. Umar Agustinus, M.Sc.	275	A
Hukum	1983-08-08	Prof. Gunarto, M.H.	754	B

```
SELECT * FROM mahasiswa;
```

nim	nama	asal	jurusan	nilai_uan
17080225	Husli Khairan	Jakarta	Akuntansi	288.55
17080305	Rina Kumala Sari	Jakarta	Akuntansi	337.99
17090113	Riana Putria	Padang	Kimia	339.20
17140119	Sandri Fatmala	Bandung	Ilmu Komputer	322.91
17140120	Bobby Permana	Medan	Ilmu Komputer	280.82
17140133	Ikhsan Prayoga	Jakarta	Ilmu Komputer	300.16
17140143	Rudi Permana	Bandung	Ilmu Komputer	290.44

Tabel universitas berisi daftar jurusan yang ada di dalam suatu universitas. Sedangkan tabel mahasiswa berisi data mahasiswa di dalam sebuah universitas. Kolom apakah yang bisa dipakai untuk menghubungkan kedua tabel ini?

Betul, kolom jurusan. Kita sudah membahas cara menampilkan hasil gabungan kedua tabel ini menggunakan query SELECT...JOIN, dan disana kolom jurusan berfungsi sebagai penghubung.

Seharusnya, ketika data mahasiswa ditambahkan ke dalam tabel mahasiswa, si mahasiswa tersebut hanya bisa memilih jurusan yang ada di tabel universitas. Logikanya adalah seorang mahasiswa tidak bisa memilih jurusan yang belum ada. Hubungan inilah yang dinamakan sebagai **referential integrity**.

Agar data di kedua tabel valid dan konsisten, mahasiswa hanya bisa memilih jurusan yang sudah tersedia di dalam tabel universitas. Isi tabel mahasiswa harus berpatokan kepada tabel universitas.

Jika si mahasiswa ingin memilih jurusan yang tidak ada di tabel universitas, kita harus menambah jurusan tersebut di tabel universitas terlebih dahulu.

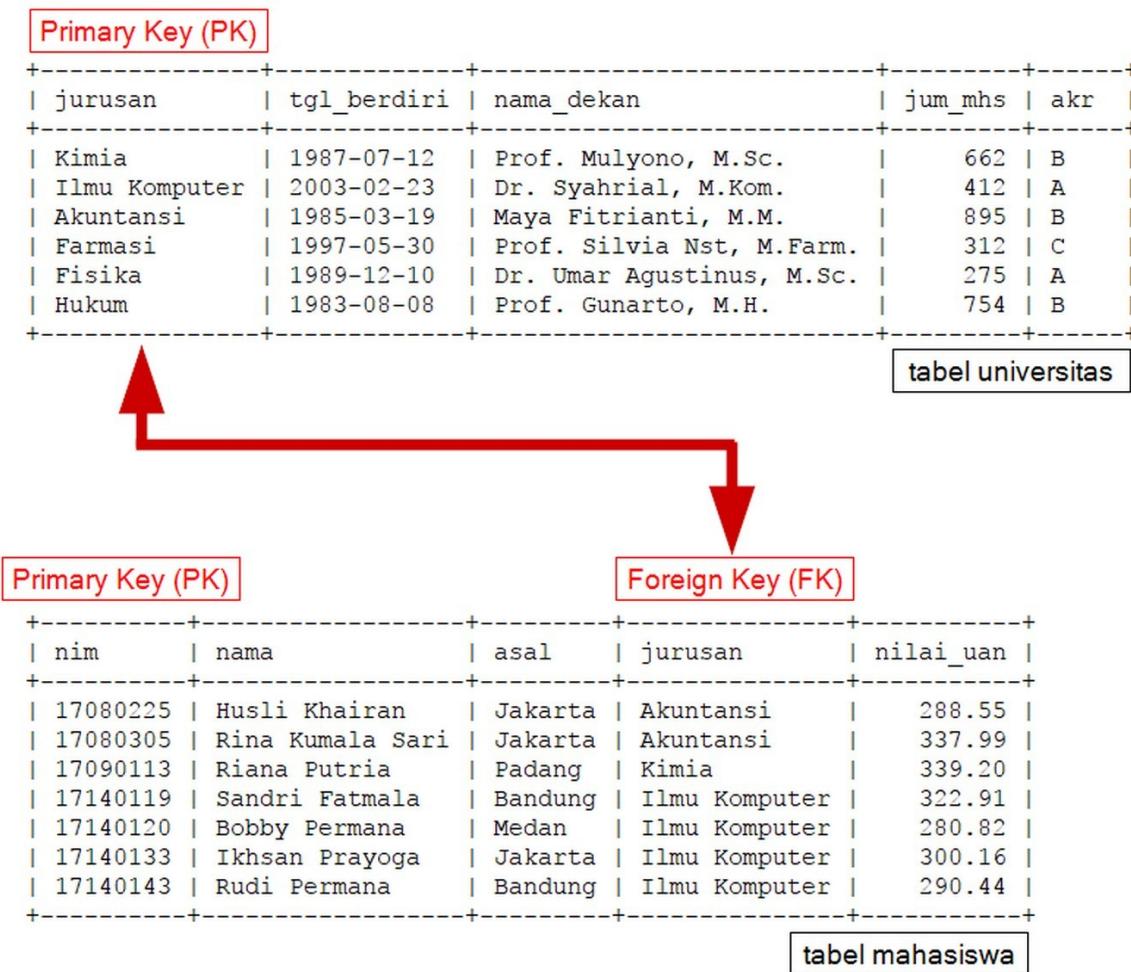
Dalam hubungan antara tabel universitas dengan tabel mahasiswa, tabel universitas disebut sebagai **parent table** (tabel induk), sedangkan tabel mahasiswa disebut sebagai **child table** (tabel anak). Ini karena tabel universitas akan menjadi patokan ketika mengisi data di tabel mahasiswa.

Mengikuti standar tabel yang baik, kolom jurusan di tabel universitas bisa di set sebagai *primary key*, begitu juga dengan kolom nim di tabel mahasiswa. Kedua kolom ini akan mengidentifikasi secara unik setiap baris tabel.

Kolom jurusan di dalam tabel mahasiswa merupakan “perpanjangan tangan” dari kolom jurusan di tabel universitas. Kolom jurusan di tabel universitas ini dikenal juga sebagai *foreign key*.

Foreign Key atau kunci tamu adalah sebutan untuk kolom yang ada di suatu tabel, dimana nilainya merujuk ke *primary key* dari tabel lain. Kolom yang bertindak sebagai *foreign key* akan dipakai untuk proses penggabungan tabel, misalnya dengan query JOIN.

Diagram berikut memperlihatkan hubungan antara tabel universitas dengan tabel mahasiswa:

Gambar: Hubungan antara *primary key* dengan *foreign key*

Dalam diagram diatas, dapat dilihat hubungan antara *primary key* (sering disingkat sebagai PK) dengan *foreign key* (sering disingkat sebagai FK). Untuk setiap tabel di dalam database, seharusnya akan saling terhubung dengan konsep *primary key - foreign key* seperti ini.

Dalam teori database, hubungan antar tabel terbagi menjadi 3 kelompok:

- Satu dengan satu (*one to one*)
- Satu dengan banyak (*one to many*)
- Banyak dengan banyak (*many to many*).

Sebagai contoh, setiap jurusan yang ada di tabel universitas boleh dipilih oleh lebih dari satu mahasiswa. Namun satu mahasiswa hanya bisa memilih satu jurusan. Hubungan ini termasuk ke dalam kelompok **one to many**.

Contoh **one to one** adalah antara NIM mahasiswa dengan NIK mahasiswa. Setiap mahasiswa hanya bisa memiliki satu nomor NIK di KTP, dan satu nomor NIK hanya bisa dimiliki oleh satu mahasiswa.

Contoh **many to many** seperti hubungan mata kuliah dengan dosen pengajar. Setiap dosen bisa mengajar lebih dari 1 mata kuliah, dan satu mata kuliah bisa diajar oleh lebih dari satu dosen.



Jika anda tertarik mempelajari lebih jauh tentang teori seperti ini, bisa mencari buku khusus mengenai **Perancangan dan Design Database**. Materi ini sangat luas dan menjadi mata kuliah khusus bagi mahasiswa jurusan komputer. Nanti akan dipelajari tentang pemodelan database, proses normalisasi, hingga diagram ERD.

Hal ini mirip seperti hubungan antara design web dengan kode CSS. Kode CSS dipakai untuk membuat tampilan web. Tapi untuk bisa membuat design web yang baik, harus belajar seni design itu sendiri (psikologi warna, golden ratio, typography, dll).

Dalam buku **MySQL Uncover**, kita hanya fokus ke praktek cara penggunaan aplikasi database MySQL Server dan MariaDB Server. Jika anda ingin menguasai cara membuat design database yang baik, sangat disarankan belajar juga teori seputar “Perancangan dan Design Database”, kemudian digabung dengan buku praktek seperti **MySQL Uncover** ini.

17.9 Cara Membuat Referential Integrity

Dalam penjelasan sebelumnya, kita telah membahas apa itu *referential integrity*, *foreign key*, *parent table*, dan *child table*. Kalimat berikut ini merangkum konsep tentang *referential integrity*:

“*Referential integrity* membuat pembatasan bahwa untuk setiap *foreign key* yang ada di *child table*, nilainya harus merujuk ke *primary key* di *parent table*”.

Dengan menggunakan contoh tabel `universitas` dan `mahasiswa`, ini artinya:

Untuk setiap jurusan yang ada di tabel `mahasiswa`, pilihannya harus merujuk ke kolom jurusan yang ada di tabel `universitas`. Seorang mahasiswa tidak bisa memilih jurusan yang daftarnya belum ada di tabel `universitas`.

Pembatasan seperti ini sebenarnya juga bisa dirancang dari sisi programming, misalnya menggunakan bahasa PHP. Kita bisa membuat form `<select>` yang pilihannya diambil dari kolom jurusan di tabel `universitas`. Sehingga di dalam form input data, jurusan yang bisa dipilih hanya yang ada di dalam tag `<select>` saja.

Namun kita juga bisa membuatnya langsung dari MySQL. Keuntungannya, jika ada yang mencoba input langsung ke database (tidak melalui form PHP), integritas data bisa tetap terjaga karena MySQL akan menolak jurusan yang tidak terdaftar (akan tampil pesan error).

Untuk membuat batasan *referential integrity* seperti ini, terdapat 2 cara: sewaktu pembuatan tabel dengan query `CREATE TABLE`, atau setelah tabel dibuat menggunakan query `ALTER TABLE`.

Jika ditulis dari query `CREATE TABLE`, perintah untuk pembuatan *referential integrity* harus ditulis di *child table*, dengan format dasar sebagai berikut:

```
CREATE TABLE nama tabel
(
    ... pendefenisian kolom...,
    ... pendefenisian kolom...,
    [CONSTRAINT]
    FOREIGN KEY (nama_kolom_child) REFERENCES parent_table (nama_kolom_parent)
        [ON DELETE reference_option]
        [ON UPDATE reference_option]
) ENGINE = INNODB;
```

Perintah yang dipakai untuk membuat *referential integrity* adalah FOREIGN KEY...REFERENCES. Query ini ditulis setelah pendefinisian kolom tabel.

Perintah opsional ON DELETE reference_option dan ON UPDATE reference_option dipakai untuk menentukan apa yang terjadi saat nilai di *parent table* dihapus. Kita akan bahas bagian ini dengan lebih detail nantinya. Karena bersifat opsional, bagian ini boleh tidak ditulis.

Sama seperti *transaction*, *referential integrity* juga hanya didukung oleh tipe tabel tertentu saja, dimana salah satunya InnoDB. Tipe tabel MyISAM dan MEMORY belum mendukung *referential integrity*.

Baik, mari kita buat praktek sebenarnya dari *referential integrity* ini. Saya akan membuat ulang tabel universitas dan tabel mahasiswa:

```
CREATE TABLE universitas (
    jurusan VARCHAR(20) PRIMARY KEY,
    tgl_berdiri DATE,
    nama_dekan VARCHAR(50)
) ENGINE = InnoDB;

INSERT INTO universitas VALUES
('Kimia', '1987-07-12', 'Prof. Mulyono, M.Sc.'),
('Ilmu Komputer', '2003-02-23', 'Dr. Syahrial, M.Kom.'),
('Akuntansi', '1985-03-19', 'Maya Fitrianti, M.M.'),
('Farmasi', '1997-05-30', 'Prof. Silvia Nst, M.Farm.');

DROP TABLE IF EXISTS mahasiswa;

CREATE TABLE mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20),
    FOREIGN KEY (jurusan) REFERENCES universitas (jurusan)
) ENGINE = InnoDB;
```

Ketika membuat tabel universitas, saya mendefinisikan kolom jurusan sebagai *primary key*, kemudian menginput 4 data jurusan: Kimia, Ilmu Komputer, Akuntansi dan Farmasi.

Di dalam query pembuatan tabel mahasiswa, saya menerapkan *referential integrity* dengan query FOREIGN KEY (jurusan) REFERENCES universitas (jurusan). Artinya, saya ingin agar kolom jurusan di tabel mahasiswa berfungsi sebagai *foreign key* yang ber-referensi ke kolom jurusan yang ada di tabel universitas.

Alternatif penulisan query pembuatan *referential integrity* ini adalah dengan tambahan perintah CONSTRAINT, seperti berikut ini:

```
CREATE TABLE mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20),
    CONSTRAINT FOREIGN KEY (jurusan) REFERENCES universitas (jurusan)
) ENGINE = InnoDB;
```

Kedua tabel universitas maupun tabel mahasiswa sama-sama di set menggunakan database engine **InnoDB**.

Selanjutnya, kita akan uji apakah batasan *referential integrity* ini sudah aktif atau tidak. Pertama, input data mahasiswa yang jurusannya ada di tabel universitas:

```
INSERT INTO mahasiswa VALUES ('17090113', 'Riana Putria', 'Padang', 'Kimia');
Query OK, 1 row affected (0.05 sec)
```

```
SELECT * FROM mahasiswa;
+-----+-----+-----+-----+
| nim | nama | asal | jurusan |
+-----+-----+-----+-----+
| 17090113 | Riana Putria | Padang | Kimia |
+-----+-----+-----+-----+
```

Tidak ada masalah, karena jurusan Kimia memang sudah tersedia di tabel universitas. Mari kita coba input mahasiswa yang memilih jurusan yang belum ada di tabel universitas:

```
INSERT INTO mahasiswa VALUES
('17140143', 'Rudi Permana', 'Bandung', 'Fisika');

-- ERROR 1452 (23000): Cannot add or update a child row: a foreign key
-- constraint fails (`belajar`.`mahasiswa`, CONSTRAINT `mahasiswa_ibfk_1` 
-- FOREIGN KEY (`jurusan`) REFERENCES `universitas` (`jurusan`))
```

Inilah batasan yang diterapkan dari *referential integrity*. MySQL akan menolak dan menampilkan pesan error yang kira-kira berbunyi: *tidak bisa menambah atau mengupdate sebuah baris di dalam child table: batasan foreign key gagal (cannot add or update a child row: a foreign key constraint fails)*.

Agar data Rudi Permana yang memilih jurusan Kimia bisa diinput, kita harus tambahkan dulu jurusan Kimia ini ke dalam tabel universitas terlebih dahulu, lalu baru input kembali:

```
INSERT INTO universitas VALUES
('Fisika', '1989-12-10', 'Dr. Umar Agustinus');
```

```
SELECT * FROM universitas;
```

jurusan	tgl_berdiri	nama_dekan
Akuntansi	1985-03-19	Maya Fitrianti, M.M.
Farmasi	1997-05-30	Prof. Silvia Nst, M.Farm.
Fisika	1989-12-10	Dr. Umar Agustinus
Ilmu Komputer	2003-02-23	Dr. Syahrial, M.Kom.
Kimia	1987-07-12	Prof. Mulyono, M.Sc.

```
INSERT INTO mahasiswa VALUES
('17140143', 'Rudi Permana', 'Bandung', 'Fisika');
```

```
SELECT * FROM mahasiswa;
```

nim	nama	asal	jurusan
17090113	Riana Putria	Padang	Kimia
17140143	Rudi Permana	Bandung	Fisika

Sekarang jurusan Kimia sudah bisa dipilih oleh mahasiswa.

Error yang sama juga akan terjadi jika kita mengupdate data di tabel `mahasiswa` dengan mengubah pilihan jurusan yang belum ada di tabel `universitas`:

```
UPDATE mahasiswa SET jurusan = 'Hukum' WHERE nim = '17140143';

--ERROR 1452 (23000): Cannot add or update a child row: a foreign key
--constraint fails (`belajar`.`mahasiswa`, CONSTRAINT `mahasiswa_ibfk_1`
--FOREIGN KEY (`jurusan`) REFERENCES `universitas` (`jurusan`))
```

Dengan batasan seperti ini, *referential integrity* memastikan bahwa data yang diinput ke tabel `mahasiswa` hanya data yang valid saja, dimana jurusan yang dipilih oleh mahasiswa harus sudah tersedia.

Percobaan berikutnya, mari kita hapus jurusan Kimia dari tabel `universitas`:

```
DELETE FROM universitas WHERE jurusan = 'Kimia';

--ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key
--constraint fails (`belajar`.`mahasiswa`, CONSTRAINT `mahasiswa_ibfk_1`
--FOREIGN KEY (`jurusan`) REFERENCES `universitas` (`jurusan`))
```

Apa yang terjadi? Padahal kali ini kita bukan mengubah nilai di tabel `mahasiswa`, tapi di tabel `universitas` (*parent table*).

Secara default, batasan *referential integrity* di dalam MySQL berlaku dua arah. Kita tidak bisa menginput data mahasiswa yang jurusannya belum tersedia. Sebaliknya, kita juga tidak bisa menghapus jurusan yang sudah dipilih mahasiswa.

Berikut isi tabel `mahasiswa` saat ini:

```
SELECT * FROM mahasiswa;
+-----+-----+-----+-----+
| nim      | nama           | asal      | jurusan |
+-----+-----+-----+-----+
| 17090113 | Riana Putria | Padang    | Kimia    |
| 17140143 | Rudi Permana | Bandung   | Fisika   |
+-----+-----+-----+-----+
```

Seandainya saya menghapus jurusan Kimia dari tabel `universitas`, bagaimana nasib si Riana Putria? Padahal syarat dari *referential integrity* adalah data di kolom *foreign key* harus tersedia di tabel induk. Karena alasan inilah MySQL melarang kita untuk menghapus jurusan Kimia di tabel `universitas`.

Agar jurusan Kimia bisa dihapus, tidak boleh ada mahasiswa yang masih memilih jurusan tersebut. Caranya, kita harus hapus data Riana Putria di tabel `mahasiswa` terlebih dahulu, baru kemudian menghapus jurusan Kimia di tabel `universitas`:

```
DELETE FROM mahasiswa WHERE nama = 'Riana Putria';
Query OK, 1 row affected (0.07 sec)
```

```
DELETE FROM universitas WHERE jurusan = 'Kimia';
Query OK, 1 row affected (0.06 sec)
```

Alasan yang sama juga melarang kita mengupdate nama jurusan selama terdapat tabel lain yang merujuk ke jurusan tersebut.

Sebagai contoh, saya tidak bisa mengubah nama jurusan `Fisika` karena di tabel `mahasiswa` masih ada mahasiswa yang memilih jurusan `Fisika`:

```
UPDATE universitas SET jurusan = 'Teknik Fisika'
WHERE nama_dekan = 'Dr. Umar Agustinus';

--ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key
--constraint fails (`belajar`.`mahasiswa`, CONSTRAINT `mahasiswa_ibfk_1`
--FOREIGN KEY (`jurusan`) REFERENCES `universitas` (`jurusan`))
```

Agar bisa mengupdate jurusan Fisika, saya harus menghapus mahasiswa yang memilih jurusan tersebut terlebih dahulu. Atau bisa juga mengupdate data mahasiswa ke jurusan lain selain Fisika.

17.10 Penggunaan aturan ON UPDATE dan ON DELETE

Pembatasan mengenai bisa tidaknya menghapus dan mengupdate *parent table*, diatur dari tambahan perintah ON UPDATE dan ON DELETE. Kedua perintah ini ditulis setelah query FOREIGN KEY...REFERENCES seperti format berikut:

```
CREATE TABLE nama tabel
(
    ... pendefenisian kolom...,
    ... pendefenisian kolom...,
    [CONSTRAINT]
    FOREIGN KEY (nama_kolom_child) REFERENCES parent_table (nama_kolom_parent)
        [ON DELETE reference_option]
        [ON UPDATE reference_option]
) ENGINE = INNODB;
```

Pengaturan ON UPDATE dan ON DELETE akan menentukan bagaimana “nasib” data di *child table* seandainya isi dari *parent table* di update atau di hapus.

Dalam contoh sebelum ini kita tidak bisa mengupdate maupun menghapus data jurusan di tabel universitas selama terdapat mahasiswa yang memilih jurusan tersebut di tabel mahasiswa. Ini merupakan aturan default bawaan MySQL.

Terdapat beberapa aturan yang bisa kita buat menggunakan perintah ON UPDATE dan ON DELETE. Kita akan lihat bagaimana pengaruhnya dalam hubungan *referential integrity*.

NO ACTION dan RESTRICT

Kedua nilai ini bermakna sama dan merupakan nilai default bawaan MySQL.

Jika ditulis sebagai ON UPDATE NO ACTION atau ON UPDATE RESTRICT, artinya MySQL akan membatasi dan milarang kita mengupdate *parent table* jika terdapat data lain di *child table* yang masih merujuk ke nilai tersebut.

Begitu juga halnya dengan ON DELETE NO ACTION atau ON DELETE RESTRICT yang akan menampilkan pesan error ketika kita menghapus data di *parent table* yang nilainya masih digunakan oleh *child table*.

Karena ini merupakan nilai default, tidak akan terlihat perbedaan ada atau tidaknya kedua nilai ini. Tiga query pembuatan tabel mahasiswa berikut akan menghasilkan aturan yang sama:

```
CREATE TABLE mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20),
    FOREIGN KEY (jurusan) REFERENCES universitas(jurusan)
) ENGINE = InnoDB;
```

```
CREATE TABLE mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20),
    FOREIGN KEY (jurusan) REFERENCES universitas(jurusan)
    ON UPDATE NO ACTION
    ON DELETE NO ACTION
) ENGINE = InnoDB;
```

```
CREATE TABLE mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20),
    FOREIGN KEY (jurusan) REFERENCES universitas(jurusan)
    ON UPDATE RESTRICT
    ON DELETE RESTRICT
) ENGINE = InnoDB;
```

Ketiga query pembuatan tabel mahasiswa ini akan menampilkan pesan error jika kita mencoba mengupdate atau menghapus kolom jurusan di tabel universitas selama nilainya masih dipakai oleh tabel mahasiswa. Agar bisa mengupdate atau menghapus jurusan tersebut, data mahasiswa harus dihapus terlebih dahulu.

CASCADE

Jika perintah ON UPDATE dan ON DELETE diberikan nilai CASCADE, akan terjadi efek turunan (*cascading*).

Maksudnya, ketika kita mengupdate kolom *primary key* di *parent table*, nilai kolom *foreign key* di *child table* juga akan ikut berubah. Atau jika kolom tersebut dihapus, data di *child table* juga akan ikut dihapus.

Dalam contoh tabel `universitas` dan `mahasiswa`, ini artinya ketika sebuah jurusan di tabel `universitas` diubah dari Kimia menjadi Fisika, seluruh data di tabel `mahasiswa` yang menggunakan jurusan Kimia juga akan berubah menjadi Fisika.

Atau jika data jurusan Kimia di hapus di tabel `universitas`, seluruh data `mahasiswa` yang memiliki jurusan Kimia juga akan terhapus secara otomatis.

Mari masuk ke contoh praktik. Saya akan membuat ulang tabel `mahasiswa` dengan tambahan perintah `ON UPDATE CASCADE` dan `ON DELETE CASCADE`:

```
DROP TABLE IF EXISTS mahasiswa;

CREATE TABLE mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20),
    FOREIGN KEY (jurusan) REFERENCES universitas(jurusan)
    ON UPDATE CASCADE
    ON DELETE CASCADE
) ENGINE = InnoDB;
```

Sebelumnya, saya akan periksa apa saja jurusan yang ada di tabel `universitas` saat ini:

```
SELECT * FROM universitas;
+-----+-----+-----+
| jurusan | tgl_berdiri | nama_dekan |
+-----+-----+-----+
| Akuntansi | 1985-03-19 | Maya Fitrianti, M.M. |
| Farmasi | 1997-05-30 | Prof. Silvia Nst, M.Farm. |
| Fisika | 1989-12-10 | Dr. Umar Agustinus |
| Ilmu Komputer | 2003-02-23 | Dr. Syahrial, M.Kom. |
+-----+-----+-----+
```

Saya akan input beberapa data awal ke dalam tabel `mahasiswa`:

```
INSERT INTO mahasiswa VALUES
('17140143', 'Rudi Permana', 'Bandung', 'Ilmu Komputer'),
('17080305', 'Rina Kumala Sari', 'Jakarta', 'Akuntansi'),
('17140119', 'Sandri Fatmala', 'Bandung', 'Ilmu Komputer'),
('17080225', 'Husli Khairan', 'Jakarta', 'Akuntansi'),
('17140133', 'Ikhsan Prayoga', 'Jakarta', 'Fisika');
```

```
SELECT * FROM mahasiswa;
```

nim	nama	asal	jurusan
17080225	Husli Khairan	Jakarta	Akuntansi
17080305	Rina Kumala Sari	Jakarta	Akuntansi
17140119	Sandri Fatmala	Bandung	Ilmu Komputer
17140133	Ikhsan Prayoga	Jakarta	Fisika
17140143	Rudi Permana	Bandung	Ilmu Komputer

Aturan pembatasan dari *referential integrity* tetap berlaku disini. Saya tidak bisa menginput mahasiswa yang memilih jurusan selain yang ada di tabel universitas saat ini.

Sekarang, saya akan coba ubah nama jurusan Ilmu Komputer menjadi Teknik Informatika di tabel universitas. Mari kita lihat hasilnya:

```
UPDATE universitas SET jurusan = 'Teknik Informatika'
WHERE nama_dekan = 'Dr. Syahrial, M.Kom.';
```

```
SELECT * FROM universitas;
```

jurusan	tgl_berdiri	nama_dekan
Akuntansi	1985-03-19	Maya Fitrianti, M.M.
Farmasi	1997-05-30	Prof. Silvia Nst, M.Farm.
Fisika	1989-12-10	Dr. Umar Agustinus
Teknik Informatika	2003-02-23	Dr. Syahrial, M.Kom.

```
SELECT * FROM mahasiswa;
```

nim	nama	asal	jurusan
17080225	Husli Khairan	Jakarta	Akuntansi
17080305	Rina Kumala Sari	Jakarta	Akuntansi
17140119	Sandri Fatmala	Bandung	Teknik Informatika
17140133	Ikhsan Prayoga	Jakarta	Fisika
17140143	Rudi Permana	Bandung	Teknik Informatika

Terlihat, mahasiswa yang sebelumnya memilih jurusan Ilmu Komputer, juga akan ikut diupdate menjadi Teknik Informatika.

Percobaan berikutnya, saya ingin menghapus jurusan Akuntansi dari tabel universitas:

```
DELETE FROM universitas WHERE jurusan = 'Akuntansi';
```

```
SELECT * FROM universitas;
```

jurusan	tgl_berdiri	nama_dekan
Farmasi	1997-05-30	Prof. Silvia Nst, M.Farm.
Fisika	1989-12-10	Dr. Umar Agustinus
Teknik Informatika	2003-02-23	Dr. Syahrial, M.Kom.

```
SELECT * FROM mahasiswa;
```

nim	nama	asal	jurusan
17140119	Sandri Fatmala	Bandung	Teknik Informatika
17140133	Ikhsan Prayoga	Jakarta	Fisika
17140143	Rudi Permana	Bandung	Teknik Informatika

Ternyata seluruh mahasiswa yang memilih jurusan Akuntansi juga ikut terhapus. Inilah efek dari pengaturan ON UPDATE CASCADE dan ON DELETE CASCADE.

SET NULL

Nilai lain yang bisa kita pakai adalah ON UPDATE SET NULL dan ON DELETE SET NULL. Efek dari kedua perintah ini adalah, ketika kolom *primary key* yang berada di *parent table* di update atau dihapus, nilai di kolom *foreign key* pada *child table* akan di set menjadi NULL.

Mari kita coba. Pertama, saya akan siapkan tabel mahasiswa yang telah di set dengan batasan *referential integrity* ON UPDATE SET NULL dan ON DELETE SET NULL:

```
DROP TABLE IF EXISTS mahasiswa;
```

```
CREATE TABLE mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20),
    FOREIGN KEY (jurusan) REFERENCES universitas(jurusan)
    ON UPDATE SET NULL
```

```

ON DELETE SET NULL
) ENGINE = InnoDB;

INSERT INTO mahasiswa VALUES
('17140143', 'Rudi Permana', 'Bandung', 'Teknik Informatika'),
('17140119', 'Sandri Fatmala', 'Bandung', 'Teknik Informatika'),
('17140133', 'Ikhsan Prayoga', 'Jakarta', 'Fisika');

SELECT * FROM mahasiswa;
+-----+-----+-----+-----+
| nim      | nama           | asal       | jurusan        |
+-----+-----+-----+-----+
| 17140119 | Sandri Fatmala | Bandung   | Teknik Informatika |
| 17140133 | Ikhsan Prayoga | Jakarta   | Fisika          |
| 17140143 | Rudi Permana   | Bandung   | Teknik Informatika |
+-----+-----+-----+-----+

```

Terdapat 3 data mahasiswa di tabel mahasiswa. Selanjutnya mari lihat isi dari tabel universitas:

```

SELECT * FROM universitas;
+-----+-----+-----+
| jurusan        | tgl_berdiri | nama_dekan        |
+-----+-----+-----+
| Farmasi        | 1997-05-30  | Prof. Silvia Nst, M.Farm. |
| Fisika         | 1989-12-10  | Dr. Umar Agustinus    |
| Teknik Informatika | 2003-02-23 | Dr. Syahrial, M.Kom.    |
+-----+-----+-----+

```

Saya akan coba ubah nama jurusan Teknik Informatika di tabel universitas menjadi Sistem Informasi dan melihat efeknya ke dalam tabel mahasiswa:

```

UPDATE universitas SET jurusan = 'Sistem Informasi'
WHERE nama_dekan = 'Dr. Syahrial, M.Kom.';

```

```

SELECT * FROM universitas;
+-----+-----+-----+
| jurusan        | tgl_berdiri | nama_dekan        |
+-----+-----+-----+
| Farmasi        | 1997-05-30  | Prof. Silvia Nst, M.Farm. |
| Fisika         | 1989-12-10  | Dr. Umar Agustinus    |
| Sistem Informasi | 2003-02-23 | Dr. Syahrial, M.Kom.    |
+-----+-----+-----+

```

```

SELECT * FROM mahasiswa;
+-----+-----+-----+-----+

```

nim	nama	asal	jurusan
17140119	Sandri Fatmala	Bandung	NULL
17140133	Ikhsan Prayoga	Jakarta	Fisika
17140143	Rudi Permana	Bandung	NULL

Terlihat bahwa nilai mahasiswa yang sebelumnya memilih Teknik Informatika di kolom jurusan akan berganti menjadi NULL.

Hal yang sama juga terjadi ketika jurusan tersebut dihapus dari tabel universitas:

```
DELETE FROM universitas WHERE jurusan = 'Fisika';
```

```
SELECT * FROM universitas;
```

jurusan	tgl_berdiri	nama_dekan
Farmasi	1997-05-30	Prof. Silvia Nst, M.Farm.
Sistem Informasi	2003-02-23	Dr. Syahrial, M.Kom.

```
SELECT * FROM mahasiswa;
```

nim	nama	asal	jurusan
17140119	Sandri Fatmala	Bandung	NULL
17140133	Ikhsan Prayoga	Jakarta	NULL
17140143	Rudi Permana	Bandung	NULL

Ketika saya menghapus jurusan Fisika di tabel universitas, kolom jurusan di tabel mahasiswa yang memilih Fisika juga akan berganti menjadi NULL. Inilah efek dari perintah ON UPDATE SET NULL dan ON DELETE SET NULL.

Pilihan antara NO ACTION, RESTRICT, CASCADE maupun SET NULL akan bergantung ke design database. Meskipun begitu, pengaturan CASCADE dirasa lebih baik karena mengizinkan kita mengubah nilai kolom *parent table* namun tetap menjaga konsistensi data dengan *child table*.

Dalam prakteknya nanti, bisa saja terdapat hingga 10 *child table* yang menggunakan satu *parent table*. Efek CASCADE dalam pengaturan *referential integrity* akan memudahkan kita mengubah seluruh *child table* secara otomatis.

Pengaturan ON UPDATE dan ON DELETE juga tidak harus memiliki nilai yang sama. Kita bisa menggunakan pengaturan yang berbeda seperti contoh berikut:

```
DROP TABLE IF EXISTS mahasiswa;

CREATE TABLE mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20),
    FOREIGN KEY (jurusan) REFERENCES universitas(jurusan)
    ON UPDATE CASCADE
    ON DELETE RESTRICT
) ENGINE = InnoDB;
```

Kali ini, saat jurusan pada tabel universitas diupdate, kolom jurusan di tabel mahasiswa juga akan ter update (efek CASCADE). Namun jika kolom jurusan pada tabel universitas dihapus, tampilkan pesan error jika jurusan tersebut masih digunakan di tabel mahasiswa (efek RESTRICT).

17.11 Membuat Referential Integrity dengan ALTER TABLE

Pada contoh-contoh sebelum ini, saya mendefinisikan *referential integrity* pada saat pembuatan tabel, yakni dari query CREATE TABLE.

Sebagai alternatif, kita juga bisa membuat *referential integrity* untuk tabel yang sudah ada menggunakan query ALTER TABLE. Format dasar penulisannya adalah sebagai berikut:

```
ALTER TABLE nama_child_table
    ADD FOREIGN KEY (nama_kolom_child)
        REFERENCES nama_parent_table (nama_kolom_parent)
        [ON DELETE reference_option]
        [ON UPDATE reference_option]
```

Berikut contoh penggunaannya:

```
DROP TABLE IF EXISTS mahasiswa;

CREATE TABLE mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20)
) ENGINE = InnoDB;

ALTER TABLE mahasiswa
    ADD FOREIGN KEY (jurusan) REFERENCES universitas(jurusan)
    ON UPDATE CASCADE
    ON DELETE SET NULL;
```

Disini saya membuat terlebih dahulu tabel mahasiswa. *referential integrity* baru ditambah sesudahnya menggunakan query ALTER TABLE.

Yang juga perlu diperhatikan, apabila tabel mahasiswa ternyata sudah berisi data dan baru kita buat batasan *referential integrity* setelah itu, nilai yang ada di dalam tabel mahasiswa harus memenuhi syarat hubungan *foreign key* dan *primary key*. Jika ternyata ada data yang tidak valid, MySQL akan mengeluarkan pesan error.

Contoh kasusnya sebagai berikut:

```
DROP TABLE IF EXISTS mahasiswa;

CREATE TABLE mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20)
) ENGINE = InnoDB;

INSERT INTO mahasiswa VALUES
    ('17090113', 'Riana Putria', 'Padang', 'Kimia'),
    ('17140143', 'Rudi Permana', 'Bandung', 'Ilmu Komputer'),
    ('17090222', 'Sari Citra Lestari', 'Jakarta', 'Manajemen'),
    ('17080305', 'Rina Kumala Sari', 'Jakarta', 'Akuntansi');

SELECT * FROM mahasiswa;
+-----+-----+-----+-----+
| nim      | nama           | asal       | jurusan     |
+-----+-----+-----+-----+
| 17080305 | Rina Kumala Sari | Jakarta   | Akuntansi  |
| 17090113 | Riana Putria   | Padang    | Kimia       |
| 17090222 | Sari Citra Lestari | Jakarta   | Manajemen  |
| 17140143 | Rudi Permana   | Bandung   | Ilmu Komputer |
+-----+-----+-----+-----+

DROP TABLE IF EXISTS universitas;

CREATE TABLE universitas (
    jurusan VARCHAR(20) PRIMARY KEY,
    tgl_berdiri DATE,
    nama_dekan VARCHAR(50)
) ENGINE = InnoDB;

INSERT INTO universitas VALUES
    ('Kimia', '1987-07-12', 'Prof. Mulyono, M.Sc.'),
    ('Ilmu Komputer', '2003-02-23', 'Dr. Syahrial, M.Kom.'),
    ('Akuntansi', '1985-03-19', 'Maya Fitrianti, M.M.'),
```

```
('Farmasi', '1997-05-30', 'Prof. Silvia Nst, M.Farm.');
```

```
SELECT * FROM universitas;
+-----+-----+
| jurusan | tgl_berdiri | nama_dekan |
+-----+-----+
| Akuntansi | 1985-03-19 | Maya Fitrianti, M.M. |
| Farmasi | 1997-05-30 | Prof. Silvia Nst, M.Farm. |
| Ilmu Komputer | 2003-02-23 | Dr. Syahrial, M.Kom. |
| Kimia | 1987-07-12 | Prof. Mulyono, M.Sc. |
+-----+-----+
```

Dalam query diatas saya membuat ulang tabel mahasiswa dan tabel universitas. Kedua tabel tidak memiliki hubungan apa-apa karena *referential integrity* belum di definisikan.

Baik, mari kita coba buat menggunakan query ALTER TABLE:

```
ALTER TABLE mahasiswa
ADD FOREIGN KEY (jurusan) REFERENCES universitas(jurusan)
ON UPDATE CASCADE
ON DELETE SET NULL;

-- ERROR 1452 (23000): Cannot add or update a child row: a foreign key
-- constraint fails (`belajar`.`#sql-1c4c_3`, CONSTRAINT `#sql-1c4c_3_ibfk_1`
-- FOREIGN KEY (`jurusan`) REFERENCES `universitas` (`jurusan`)
-- ON DELETE SET NULL ON UPDATE CASCADE)
```

Proses pembuatan *referential integrity* gagal. Apa yang terjadi?

Sumber masalah ada di isi data tabel mahasiswa. Terdapat mahasiswa yang memilih jurusan manajemen. Jurusan ini tidak ada di tabel universitas, sehingga MySQL menolak membuat hubungan *referential integrity* antara kedua tabel berikut.

Solusinya, kita bisa menambahkan jurusan manajemen ke tabel universitas, atau menghapus data mahasiswa yang memilih jurusan manajemen di tabel mahasiswa. Mari kita coba pilihan yang kedua ini:

```
DELETE FROM mahasiswa WHERE jurusan = 'Manajemen';
Query OK, 1 row affected (0.08 sec)
```

```
ALTER TABLE mahasiswa
ADD FOREIGN KEY (jurusan) REFERENCES universitas(jurusan)
ON UPDATE CASCADE
ON DELETE SET NULL;
```

```
Query OK, 3 rows affected (0.94 sec)
Records: 3 Duplicates: 0 Warnings: 0
```

Sekarang, *referential integrity* sukses ditambahkan.

17.12 Menghapus Referential Integrity

Untuk menghapus *referential integrity* juga menggunakan query ALTER TABLE, dengan format dasar sebagai berikut:

```
ALTER TABLE nama_child_table DROP FOREIGN KEY nama_key;
```

Dalam contoh kita, `nama_child_table` ini adalah tabel `mahasiswa`, yakni tabel yang berperan sebagai *child table*. Yang menjadi masalah adalah `nama_key`. Dari mana nilai ini didapat?

Pada saat pembuatan *referential integrity*, kita juga bisa membuat nama key (nama dari index yang menghubungkan *foreign key* dengan *primary key*), namun nama ini bersifat opsional.

Untuk melihat nama key, bisa menggunakan query SHOW CREATE TABLE. Berikut hasil yang saya dapat:

```
SHOW CREATE TABLE mahasiswa \G
*****
1. row *****
Table: mahasiswa
Create Table: CREATE TABLE `mahasiswa` (
  `nim` char(8) NOT NULL,
  `nama` varchar(50) DEFAULT NULL,
  `asal` varchar(50) DEFAULT NULL,
  `jurusan` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`nim`),
  KEY `jurusan` (`jurusan`),
  CONSTRAINT `mahasiswa_ibfk_1` FOREIGN KEY (`jurusan`)
    REFERENCES `universitas` (`jurusan`) ON DELETE SET NULL ON UPDATE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1
1 row in set (0.07 sec)
```

Perhatikan isi di baris berikut:

```
CONSTRAINT `mahasiswa_ibfk_1` FOREIGN KEY (`jurusan`)
```

Nilai `mahasiswa_ibfk_1` adalah nama key yang digenerate MySQL sebagai nama dari *referential integrity*. Inilah nama key yang harus ditulis ke dalam query ALTER TABLE untuk menghapus *referential integrity*.

Mari kita coba:

```
ALTER TABLE mahasiswa DROP FOREIGN KEY mahasiswa_ibfk_1;
```

```
Query OK, 0 rows affected (0.17 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Sekarang batasan *referential integrity* sudah terhapus. Untuk memeriksanya, mari jalankan kembali query SHOW CREATE TABLE:

```
SHOW CREATE TABLE mahasiswa \G
*****
Table: mahasiswa
Create Table: CREATE TABLE `mahasiswa` (
  `nim` char(8) NOT NULL,
  `nama` varchar(50) DEFAULT NULL,
  `asal` varchar(50) DEFAULT NULL,
  `jurusan` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`nim`),
  KEY `jurusan` (`jurusan`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```

Sekarang tidak terlihat lagi query CONSTRAINT. Artinya di dalam table mahasiswa sudah tidak memiliki *referential integrity*.

17.13 Membuat nama Referential Integrity

Membuat nama *referential integrity* sebenarnya opsional. Seperti yang sudah kita lihat sebelumnya, MySQL akan memilih nama key secara otomatis apabila tidak ditulis.

Namun kita juga bisa membuat name key secara manual. Berikut contohnya:

```
CREATE TABLE mahasiswa (
  nim CHAR(8) PRIMARY KEY,
  nama VARCHAR(50),
  asal VARCHAR(50),
  jurusan VARCHAR(20),
  FOREIGN KEY jurusan_fk (jurusan) REFERENCES universitas(jurusan)
  ON UPDATE RESTRICT
  ON DELETE RESTRICT
) ENGINE = InnoDB;
```

Ketika membuat batasan *referential integrity*, saya menulis FOREIGN KEY jurusan_fk (jurusan) REFERENCES universitas(jurusan). Disini jurusan_fk akan menjadi nama dari key.

Hal ini bisa dilihat dari query SHOW CREATE TABLE:

```
SHOW CREATE TABLE mahasiswa \G
*****
1. row ****
Table: mahasiswa
Create Table: CREATE TABLE `mahasiswa` (
  `nim` char(8) NOT NULL,
  `nama` varchar(50) DEFAULT NULL,
  `asal` varchar(50) DEFAULT NULL,
  `jurusan` varchar(20) DEFAULT NULL,
  PRIMARY KEY (`nim`),
  KEY `jurusan_fk` (`jurusan`),
  CONSTRAINT `jurusan_fk` FOREIGN KEY (`jurusan`)
    REFERENCES `universitas` (`jurusan`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

Karena kita sudah membuat langsung nama key, untuk proses penghapusan *referential integrity*, nama inilah yang nantinya dipakai:

```
ALTER TABLE mahasiswa DROP FOREIGN KEY jurusan_fk;
```

```
Query OK, 0 rows affected (0.08 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Untuk membuat nama key menggunakan query ALTER TABLE, caranya juga sama. Yakni dengan menulis nama key setelah perintah ADD FOREIGN KEY:

```
ALTER TABLE mahasiswa
ADD FOREIGN KEY jurusan_fk (jurusan) REFERENCES universitas(jurusan)
ON UPDATE CASCADE
ON DELETE CASCADE;
```

17.14 Index di Parent Table

Sepanjang contoh yang sudah kita coba, saya selalu membuat kolom jurusan di tabel universitas sebagai *primary key*. Sebenarnya ini bukanlah syarat mutlak.

Agar batasan *referential integrity* bisa dibuat, setidaknya kolom utama di dalam *parent table* harus ter-index. Jika tidak, akan tampil pesan error ketika pada saat penambahan aturan *referential integrity*.

Mari kita lihat contohnya:

```

DROP TABLE IF EXISTS mahasiswa;

DROP TABLE IF EXISTS universitas;

CREATE TABLE universitas (
    jurusan VARCHAR(20),
    tgl_berdiri DATE,
    nama_dekan VARCHAR(50)
) ENGINE = InnoDB;

INSERT INTO universitas VALUES
('Kimia', '1987-07-12', 'Prof. Mulyono, M.Sc.'),
('Ilmu Komputer', '2003-02-23', 'Dr. Syahrial, M.Kom.'),
('Akuntansi', '1985-03-19', 'Maya Fitrianti, M.M.'),
('Farmasi', '1997-05-30', 'Prof. Silvia Nst, M.Farm.');

CREATE TABLE mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20),
    FOREIGN KEY (jurusan) REFERENCES universitas(jurusan)
) ENGINE = InnoDB;

-- ERROR 1005 (HY000): Can't create table `belajar`.`mahasiswa`
-- (errno: 150 "Foreign key constraint is incorrectly formed")

```

Di awal query saya menghapus tabel `mahasiswa` terlebih dahulu baru kemudian menghapus tabel `universitas`.

Ini karena aturan *referential integrity* akan membatasi proses delete data di tabel `universitas` jika data tersebut masih digunakan di tabel `mahasiswa` (mudah-mudahan anda masih ingat tentang konsep ON UPDATE dan ON DELETE).

Aturan ini juga berlaku saat menghapus tabel `universitas`. Karena dalam contoh sebelumnya kedua tabel masih terhubung, akan tampil pesan error ketika saya coba untuk menghapus tabel `universitas`. Karena, di tabel `mahasiswa` masih terdapat data yang menggunakan nilai jurusan dari tabel `universitas`. Oleh karena itu, tabel `mahasiswa` harus dihapus terlebih dahulu sebelum menghapus tabel `universitas`.

Selanjutnya saya membuat ulang tabel `universitas` dan diikuti tabel `mahasiswa` dan query FOREIGN KEY untuk membuat batasan *referential integrity*. Namun query pembuatan tabel `mahasiswa` gagal diproses dengan pesan error: *Foreign key constraint is incorrectly formed*.

Error ini terjadi karena kolom `jurus` di tabel `universitas` tidak memiliki index atau key. Jika kolom tersebut dijadikan *primary key*, otomatis akan ter-index. Namun kali ini mari kita coba menggunakan index biasa (bukan *primary key*):

```
ALTER TABLE universitas ADD INDEX (jurusan);

Query OK, 0 rows affected (0.30 sec)
Records: 0 Duplicates: 0 Warnings: 0

CREATE TABLE mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR(20),
    FOREIGN KEY (jurusan) REFERENCES universitas(jurusan)
) ENGINE = InnoDB;
Query OK, 0 rows affected (0.19 sec)
```

Saya menggunakan query **ALTER TABLE** untuk menambahkan index ke kolom jurusan di tabel universitas. Setelah itu saya menjalankan kembali query pembuatan tabel `mahasiswa` yang di dalamnya terdapat query **FOREIGN KEY...REFERENCES** untuk pembuatan *referential integrity*.

Kali ini tidak ada masalah. Mari kita test input data ke tabel `mahasiswa`:

```
INSERT INTO mahasiswa VALUES ('17090113', 'Riana Putria', 'Padang', 'Kimia');
Query OK, 1 row affected (0.18 sec)

INSERT INTO mahasiswa
VALUES ('17140143', 'Rudi Permana', 'Bandung', 'Fisika');

-- ERROR 1452 (23000): Cannot add or update a child row: a foreign key
-- constraint fails (`belajar`.`mahasiswa`, CONSTRAINT `mahasiswa_ibfk_1` 
-- FOREIGN KEY (`jurusan`) REFERENCES `universitas` (`jurusan`))
```

Yup, pembatasan *referential integrity* sudah berlaku.

Dari contoh ini terlihat bahwa MySQL membolehkan kita membuat hubungan *referential integrity* tanpa perlu menjadikan kolom utama di *parent table* sebagai *primary key*.

Meskipun begitu, ini bukanlah design database yang baik. Nilai di tabel `universitas` bisa menampung data yang berulang dan akan menjadi masalah seandainya ada yang menginput 2 buah jurusan Kimia di tabel `universitas`. Akibatnya, data di tabel tidak lagi konsisten. Oleh karena itu sebaiknya kita tetap membuat kolom utama di *parent table* sebagai *primary key*.

Sepanjang bab ini, kita telah membahas cara pembuatan **primary key**, **foreign key** serta konsep **referential integrity**.

Contoh tabel yang saya gunakan memang masih sangat sederhana, tapi ini menjadi media belajar yang mudah untuk memahami konsep *referential integrity*.

Dalam praktek di aplikasi “real world”, bisa jadi ada 10 tabel yang saling terhubung satu sama lain. Dengan menerapkan pembatasan *referential integrity*, data di dalam setiap tabel bisa konsisten satu sama lain.

18. Import dan Export Data

Sepanjang operasional database, ada kalanya kita butuh input data dalam jumlah banyak dari sebuah file external, atau kita perlu membuat backup database sebagai cadangan jika terjadi sesuatu. Semua ini butuh mekanisme **import** dan **export** data dari dan ke database MySQL.

Dalam bab ini kita akan belajar cara menginput data dari file external ke dalam tabel MySQL, serta cara menulis hasil database ke file khusus. Akan dibahas juga trik singkat untuk import data dari file CSV ke dalam MySQL.

- i** Karena kita akan banyak membuat file teks, dalam bab ini saya menggunakan aplikasi **Notepad++**. Text editor Notepad++ bisa di download gratis dari alamat berikut: <http://notepad-plus-plus.org/download/>. Tentu saja anda juga boleh menggunakan teks editor lain, seperti **Sublime Text** atau **Komodo Edit**.

18.1 Import Data dengan LOAD DATA INFILE

Query `LOAD DATA INFILE` berfungsi untuk menginput file teks external ke dalam tabel MySQL. Berikut format dasar penulisan query `LOAD DATA INFILE`:

```
LOAD DATA [LOW_PRIORITY | CONCURRENT] [LOCAL] INFILE 'file_name'  
[REPLACE | IGNORE]  
INTO TABLE tbl_name  
[PARTITION (partition_name [, partition_name] ...)]  
[CHARACTER SET charset_name]  
[{FIELDS | COLUMNS}  
    [TERMINATED BY 'string']  
    [[OPTIONALLY] ENCLOSED BY 'char']  
    [ESCAPED BY 'char']  
]  
[LINES  
    [STARTING BY 'string']  
    [TERMINATED BY 'string']  
]  
[IGNORE number {LINES | ROWS}]  
[(col_name_or_user_var  
    [, col_name_or_user_var] ...)]  
[SET col_name={expr | DEFAULT},  
    [, col_name={expr | DEFAULT}] ...]
```

Cukup banyak pengaturan untuk query `LOAD DATA INFILE` ini, dan akan kita bahas secara bertahap.

Sebagai tabel praktek, saya akan membuat tabel `provinsi` dengan query sebagai berikut:

```
DROP TABLE IF EXISTS provinsi;

CREATE TABLE provinsi (
    nama_prov VARCHAR(50),
    kode_iso CHAR(2),
    ibukota VARCHAR (50),
    populasi INT,
    luas DEC(8,2),
    apbd DEC(8,2)
);
```

Tabel provinsi ini akan berisi data provinsi yang ada di Indonesia. Data provinsi nantinya saya ambil dari Wikipedia: [Daftar provinsi di Indonesia](#)¹. Berikut penjelasan dari setiap kolom:

- nama_prov VARCHAR(50): nama provinsi.
- kode_iso CHAR(2): kode provinsi dalam bentuk 2 digit huruf, contoh: JK untuk Jakarta.
- ibukota VARCHAR (50): nama ibukota provinsi.
- populasi INT: jumlah populasi penduduk di sebuah provinsi.
- luas DEC(8,2): luas wilayah dalam satuan km².
- apbd DEC(8,2): jumlah APBD provinsi dalam satuan miliar rupiah.

Untuk menginput data ke dalam tabel provinsi, kita tidak lagi menggunakan query INSERT, tapi meng-importnya dari file external menggunakan query LOAD DATA INFILE.

Bentuk paling sederhana dari query LOAD DATA INFILE adalah sebagai berikut:

```
LOAD DATA INFILE 'nama_file' INTO TABLE nama_tabel
```

Sebelum menjalankan query tersebut, kita harus siapkan sebuah file teks yang berisi data untuk tabel provinsi. Silahkan buka aplikasi teks editor **NotePad++** (atau teks editor lain), buat file baru dengan isi teks sebagai berikut:

nama_provinsi_a.txt

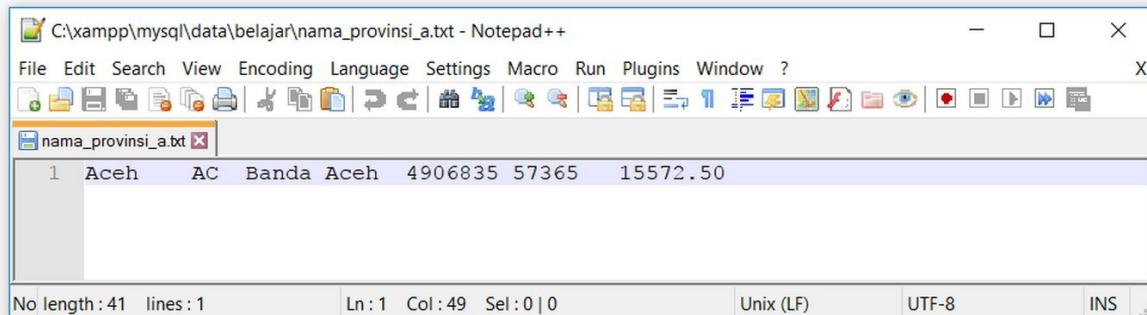
Aceh	AC	Banda Aceh	4906835	57365	15572.50
------	----	------------	---------	-------	----------

Diantara setiap data **dipisah dengan tab**, bukan spasi. Artinya teks diatas harus diketik seperti ini:

Aceh{tab}AC{tab}Banda Aceh{tab}4906835{tab}57365{tab}15572.50

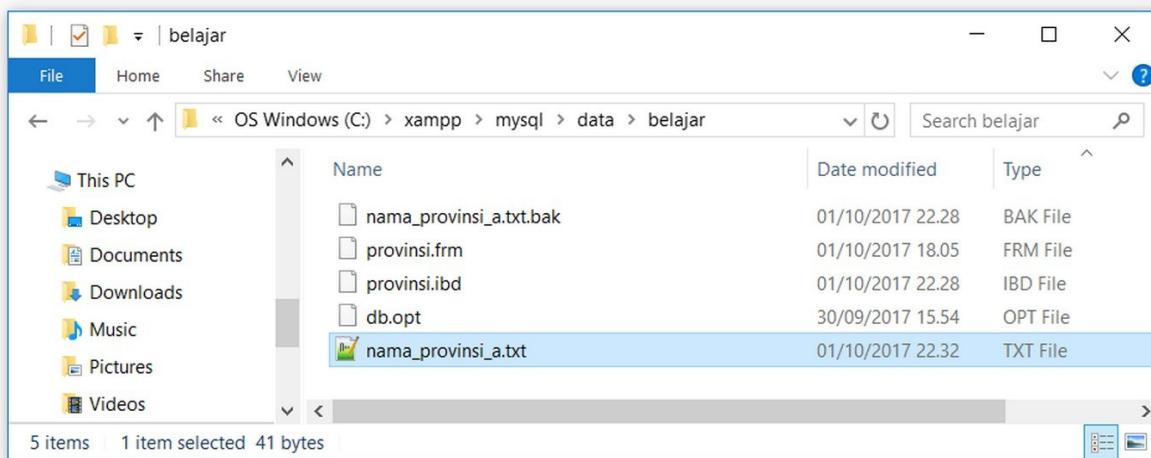
¹https://id.wikipedia.org/wiki/Daftar_provinsi_di_Indonesia

Pemisah tab harus ditulis persis seperti diatas. Karena jika tidak, query LOAD DATA INFILE tidak akan bisa membacanya.



Gambar: Pembuatan file `nama_provinsi_a.txt` menggunakan Notepad++

Save file tersebut sebagai `nama_provinsi_a.txt` dan simpan ke folder `C:\xampp\mysql\data\belajar`.



Gambar: Simpan file `nama_provinsi_a.txt` ke `C:\xampp\mysql\data\belajar`

Buka jendela cmd MySQL, lalu jalankan query berikut:

```
LOAD DATA INFILE 'nama_provinsi_a.txt' INTO TABLE provinsi;
```

```
Query OK, 1 row affected (0.13 sec)
Records: 1 Deleted: 0 Skipped: 0 Warnings: 0
```

```
SELECT * FROM provinsi;
+-----+-----+-----+-----+-----+
| nama_prov | kode_iso | ibukota | populasi | luas | apbd |
+-----+-----+-----+-----+-----+
| Aceh | AC | Banda Aceh | 4906835 | 57365.00 | 15572.50 |
+-----+-----+-----+-----+-----+
```

Hasilnya, teks yang kita ketik di `nama_provinsi_a.txt` akan di-input ke dalam tabel `provinsi`.



Jika anda kesulitan membuat file `nama_provinsi_a.txt` atau tidak mau repot mengetik manual, silahkan download file `belajar_mysql.zip` dari folder sharing Google Drive. Extract file zip tersebut dan buka folder `bab_18_import_dan_export_data`. Dalam folder ini sudah tersedia seluruh file yang nantinya kita pakai sepanjang bab ini.

Query `LOAD DATA INFILE 'nama_provinsi_a.txt' INTO TABLE provinsi` artinya saya ingin menginput seluruh teks dari `nama_provinsi_a.txt` ke tabel `provinsi`.

Secara default, karakter **tab** dipakai sebagai pemisah antar kolom. Karena tabel `provinsi` terdiri dari 6 kolom, maka di dalam file teks juga harus tersedia 6 kolom data yang dipisah dengan karakter tab (nantinya kita bisa menggunakan karakter lain).

Masih sebagai settingan default, query `LOAD DATA INFILE` akan mencari file teks di folder tempat database tersimpan. Karena saya menggunakan database `belajar`, maka file teks tersebut harus ditempatkan di folder `C:\xampp\mysql\data\belajar\`.

Folder `belajar` ini akan jadi titik patokan ketika kita menggunakan *alamat relatif*.

Jika anda sudah membaca buku **HTML Uncover**, tentu masih ingat perbedaan **alamat relatif** dan **alamat absolut**. Di dalam HTML, alamat relatif adalah alamat yang berpatokan dari file saat ini, seperti `../belajar_html/index.html`. Sedangkan alamat absolut adalah alamat yang ditulis lengkap seperti `www.duniaIlkom.com/belajar_html/index.html`.

Dalam contoh query yang kita jalankan sebelum ini, itu menggunakan *alamat relatif* dengan patokan dihitung dari folder `belajar`, yakni folder dimana database saat ini berada.

Jika saya menjalankan query yang sama tapi menggunakan database **sekolah** (`USE sekolah`), maka patokannya ada di folder `sekolah`. Folder `sekolah` otomatis akan dibuat oleh MySQL pada saat menjalankan query `CREATE DATABASE sekolah`. Di dalam folder inilah seluruh file tabel untuk database `sekolah` disimpan.

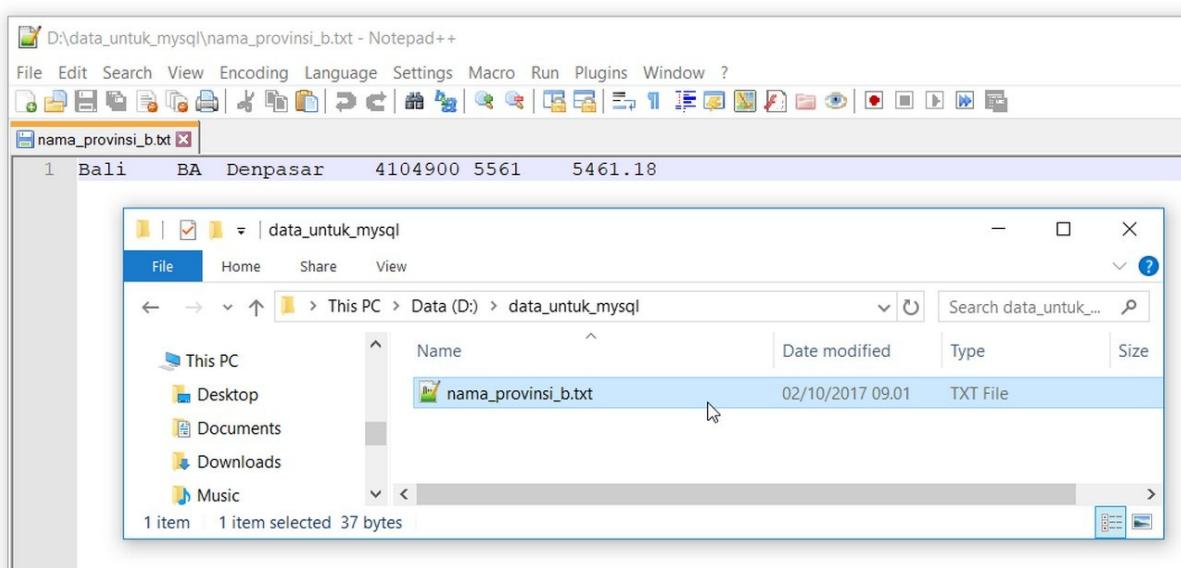
Selain *alamat relatif*, kita juga menulis *alamat absolut*, yakni dengan menulis lengkap alamat file seperti: `D:\data_untuk_mysql\nama_provinsi_b.txt`. Mari kita coba.

Kali ini silahkan buat folder `data_untuk_mysql` di drive D, kemudian di dalamnya buat file teks `nama_provinsi_b.txt` dengan isi sebagai berikut:

`nama_provinsi_b.txt`

Bali	BA	Denpasar	4104900	5561	5461.18
------	----	----------	---------	------	---------

Kembali, gunakan tab sebagai pemisah teks. Alamat lengkap dari file diatas menjadi: `D:\data_untuk_mysql\nama_provinsi_b.txt`.



Gambar: Simpan file `nama_provinsi_b.txt` di folder `D:\data_untuk_mysql`

Setelah file tersedia, mari kita tes dengan query berikut:

```
LOAD DATA INFILE 'D:\\data_untuk_mysql\\\\nama_provinsi_b.txt'
INTO TABLE provinsi;
```

```
Query OK, 1 row affected (0.10 sec)
Records: 1 Deleted: 0 Skipped: 0 Warnings: 0
```

```
SELECT * FROM provinsi;
+-----+-----+-----+-----+-----+
| nama_prov | kode_iso | ibukota | populasi | luas | apbd |
+-----+-----+-----+-----+-----+
| Aceh | AC | Banda Aceh | 4906835 | 57365.00 | 15572.50 |
| Bali | BA | Denpasar | 4104900 | 5561.00 | 5461.18 |
+-----+-----+-----+-----+-----+
```

File `nama_provinsi_b.txt` sukses diinput ke dalam tabel `provinsi`.

Perhatikan cara penulisan alamat file, dimana alamat `D:\data_untuk_mysql\nama_provinsi_b.txt` ditulis menjadi `D:\\data_untuk_mysql\\\\nama_provinsi_b.txt`.

Di sistem operasi Windows, tanda *forward slash* (`\`) dipakai sebagai penanda folder. Masalahnya, tanda `\` memiliki fungsi lain di dalam string MySQL, yakni untuk men-escape karakter. Oleh karena itu tanda *forward slash* (`\`) harus ditulis dua kali menjadi (`\\`).

Aternatifnya kita juga bisa menggunakan *backslash* (`/`). Query `LOAD DATA INFILE` sebelumnya juga bisa ditulis sebagai berikut:

```
LOAD DATA INFILE 'D:/data_untuk_mysql/nama_provinsi_b.txt'  
INTO TABLE provinsi;
```

Dengan menulis alamat absolut, kita bisa menempatkan file teks dimanapun di dalam komputer (tidak harus di dalam folder instalasi XAMPP atau MySQL).



Jika anda menjalankan query LOAD DATA INFILE di sistem operasi Linux atau Mac OS, MySQL harus diberikan hak akses agar bisa membaca file dan folder tersebut.

LOAD DATA LOCAL INFILE

Sebagaimana yang pernah kita bahas di bab 5 tentang “MySQL Server dan MySQL Client”, sistem kerja MySQL menggunakan metode *client-server*. Satu komputer bertindak sebagai **server** (*mysqld.exe*), dan komputer lain berperan sebagai **client** (*mysql.exe*).

Dalam praktek dunia nyata, MySQL server biasanya berjalan di satu komputer khusus (komputer server), kemudian diakses bersama-sama oleh MySQL client dari tempat lain. Satu MySQL server bisa diakses oleh puluhan, ratusan, hingga jutaan MySQL client.

Hanya saja sepanjang praktek dalam buku ini, kita menjalankan MySQL server dan MySQL client di satu komputer yang sama.

Query LOAD DATA INFILE yang dijalankan sebelum ini **akan mencari file di sistem Server**, yakni di komputer dimana file *mysqld.exe* dijalankan. MySQL menyediakan query opsional LOAD DATA LOCAL INFILE untuk **mencari file di komputer client**. File ini nantinya dikirim ke server baru kemudian diinput ke dalam tabel.

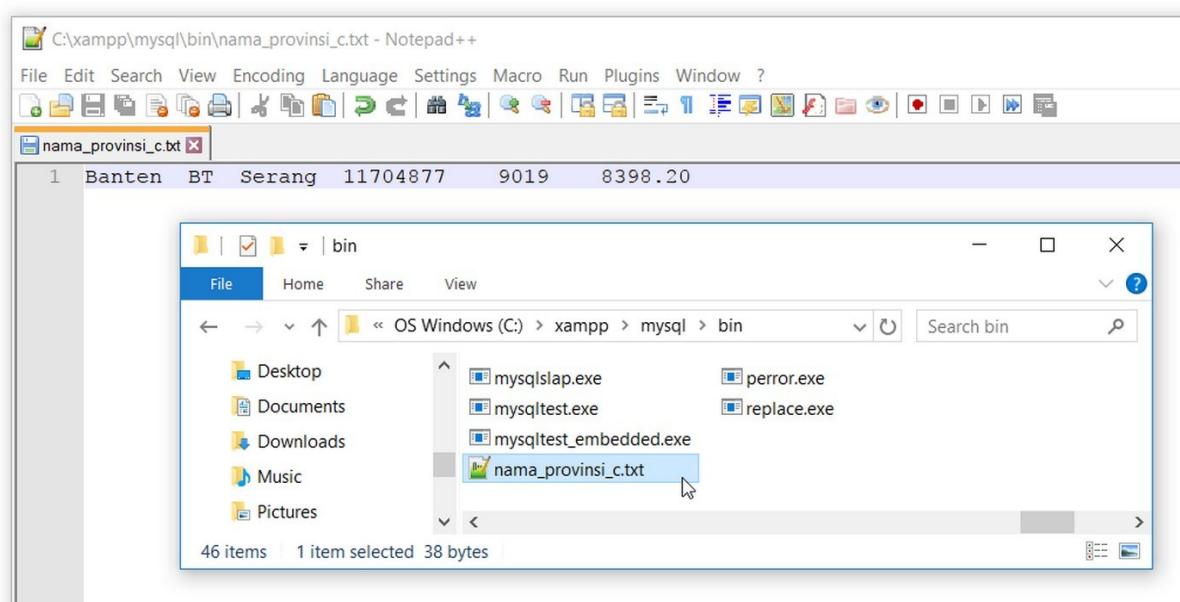
Karena kita menjalankan MySQL server dan MySQL client di komputer yang sama, perbedaan antara perintah LOAD DATA INFILE dengan LOAD DATA LOCAL INFILE tidak begitu berpengaruh. Bedanya baru terlihat jika kita menggunakan alamat **relatif** pada saat pembacaan file.

Query LOAD DATA INFILE berpatokan dari tempat **folder database berada**. Sedangkan query LOAD DATA LOCAL INFILE berpatokan dari folder dimana **mysql.exe berada**. Karena saya menggunakan **MariaDB** bawaan XAMPP, lokasi dari *mysql.exe* ada di: C:\xampp\mysql\bin.

Sebagai praktek, saya akan buat file teks bernama *nama_provinsi_c.txt* yang disimpan ke dalam folder C:\xampp\mysql\bin. Isi file teks ini adalah sebagai berikut:

nama_provinsi_c.txt

Banten	BT	Serang	11704877	9019	8398.20
--------	----	--------	----------	------	---------



Gambar: Simpan file `nama_provinsi_c.txt` di folder `C:\xampp\mysql\bin\`

Alamat file diatas menjadi: `C:\xampp\mysql\bin\nama_provinsi_c.txt`, dan tetap gunakan karakter `tab` sebagai pemisah kolom.

Berikut query yang dipakai untuk mengakses file `nama_provinsi_c.txt`:

```
LOAD DATA LOCAL INFILE 'nama_provinsi_c.txt' INTO TABLE provinsi;
```

```
Query OK, 1 row affected (0.06 sec)
Records: 1 Deleted: 0 Skipped: 0 Warnings: 0
```

```
SELECT * FROM provinsi;
+-----+-----+-----+-----+-----+
| nama_prov | kode_iso | ibukota | populasi | luas | apbd |
+-----+-----+-----+-----+-----+
| Aceh | AC | Banda Aceh | 4906835 | 57365.00 | 15572.50 |
| Bali | BA | Denpasar | 4104900 | 5561.00 | 5461.18 |
| Banten | BT | Serang | 11704877 | 9019.00 | 8398.20 |
+-----+-----+-----+-----+-----+
```

Sekarang sudah terdapat 3 baris data di tabel `provinsi`. Ketiganya berasal dari 3 file teks yang sudah kita jalankan.

Untuk menguji pemahaman, bisakah anda membedakan maksud dari kedua perintah dibawah ini?

```
LOAD DATA INFILE 'nama_provinsi_c.txt' INTO TABLE provinsi;

LOAD DATA LOCAL INFILE 'nama_provinsi_c.txt' INTO TABLE provinsi;
```

Query pertama (tanpa tambahan perintah LOCAL), artinya file teks akan **dicari di komputer server**. Dalam praktik kita yang menggunakan XAMPP, file `nama_provinsi_c.txt` harus berada di folder `C:\xampp\mysql\data\belajar\`.

Query kedua (dengan perintah LOCAL), artinya file teks akan **dicari di komputer client**. Dalam praktik kita yang menggunakan XAMPP, file `nama_provinsi_c.txt` harus berada di folder `C:\xampp\mysql\bin\`.

Perbedaan antara `LOAD DATA INFILE` dan `LOAD DATA LOCAL INFILE` hanya untuk penulisan alamat relatif. Jika kita menulis alamat absolut, tidak ada perbedaan sama sekali. Dua query berikut akan bermakna sama:

```
LOAD DATA INFILE 'D:/data_untuk_mysql/nama_provinsi_b.txt'
INTO TABLE provinsi;
```

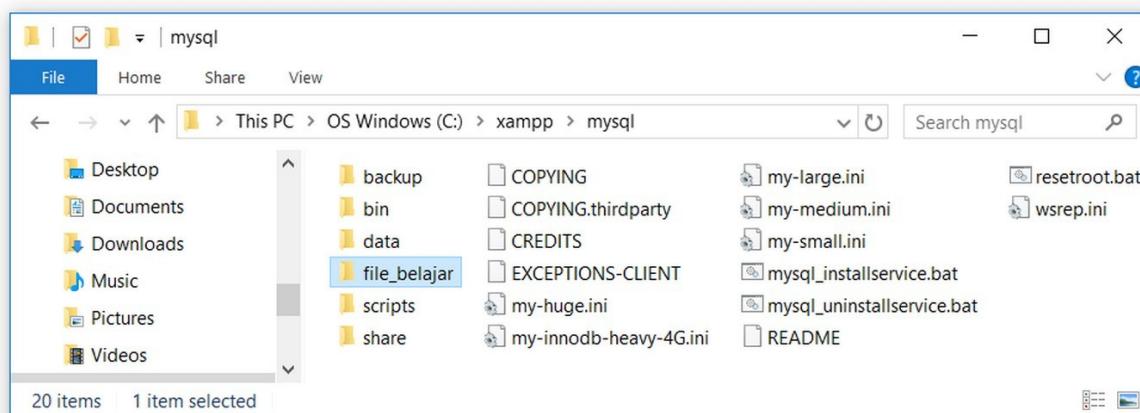
```
LOAD DATA LOCAL INFILE 'D:/data_untuk_mysql/nama_provinsi_b.txt'
INTO TABLE provinsi;
```

Kedua perintah ini akan mencari file `nama_provinsi_b.txt` di folder `D:\data_untuk_mysql`, karena kita menggunakan komputer yang sama untuk menjalankan `mysqld.exe` dan `mysql.exe`.

LOAD DATA INFILE ... FIELDS TERMINATED BY

Secara bawaan, query `LOAD DATA INFILE` membaca karakter `tab` sebagai pembatas data antar kolom. Namun kita juga bisa mendefinisikan sendiri karakter apa yang akan dipakai dengan perintah tambahan `FIELDS TERMINATED BY`.

Sebagai praktik, saya akan membuat folder `file_belajar` di `C:\xampp\mysql\`. Folder ini akan menjadi lokasi penyimpanan seluruh file teks dalam bab ini. Alamat folder tersebut berada di `C:\xampp\mysql\file_belajar\`.



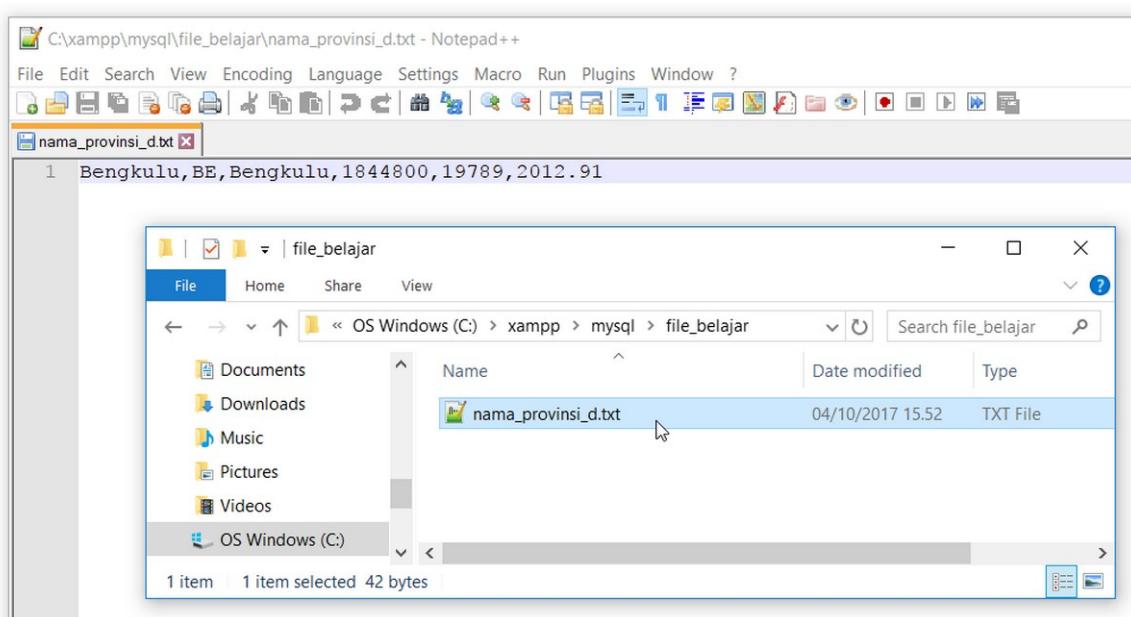
Gambar: Lokasi folder file belajar

Setelah folder **file_belajar** tersedia, buat file teks bernama **nama_provinsi_d.txt** dengan isi sebagai berikut:

nama_provinsi_d.txt

Bengkulu,BE,Bengkulu,1844800,19789,2012.91

Kali ini pemisah antar kolom bukan tab lagi, tapi karakter koma (,).



Gambar: Simpan file **nama_provinsi_d.txt** di folder **C:\xampp\mysql\file_text**

Karena disini tanda pemisah kolom tidak lagi menggunakan karakter tab, kita harus memberi-tahu MySQL agar membaca karakter koma sebagai pemisah kolom. Querynya menjadi sebagai berikut:

```
LOAD DATA INFILE 'C:\\xampp\\mysql\\file_belajar\\nama_provinsi_d.txt'
INTO TABLE provinsi
FIELDS TERMINATED BY ',';
```

```
Query OK, 1 row affected (0.09 sec)
Records: 1 Deleted: 0 Skipped: 0 Warnings: 0
```

```
SELECT * FROM provinsi;
+-----+-----+-----+-----+-----+
| nama_prov | kode_iso | ibukota | populasi | luas | apbd |
+-----+-----+-----+-----+-----+
| Aceh | AC | Banda Aceh | 4906835 | 57365.00 | 15572.50 |
| Bali | BA | Denpasar | 4104900 | 5561.00 | 5461.18 |
| Banten | BT | Serang | 11704877 | 9019.00 | 8398.20 |
| Bengkulu | BE | Bengkulu | 1844800 | 19789.00 | 2012.91 |
+-----+-----+-----+-----+-----+
```

Query FIELDS TERMINATED BY ',' adalah pengaturan tambahan agar karakter koma (,) dibaca sebagai pemisah kolom. Kita bisa menggantinya dengan karakter lain tergantung file teks yang tersedia.

Sebagai latihan, saya memiliki file `nama_provinsi_e.txt` dengan isi teks sebagai berikut:

```
nama_provinsi_e.txt
```

```
Gorontalo|GO|Gorontalo|1115633|11968|1406.32
```

Bisakah anda merancang query untuk bisa menginput file teks tersebut ke dalam tabel provinsi? Kita tinggal mengganti baris FIELDS TERMINATED BY ',' dengan FIELDS TERMINATED BY '|':

```
LOAD DATA INFILE 'C:\\xampp\\mysql\\file_belajar\\nama_provinsi_e.txt'
INTO TABLE provinsi
FIELDS TERMINATED BY '|';
```

```
Query OK, 1 row affected (0.12 sec)
Records: 1 Deleted: 0 Skipped: 0 Warnings: 0
```

```
SELECT * FROM provinsi;
+-----+-----+-----+-----+-----+
| nama_prov | kode_iso | ibukota      | populasi | luas       | apbd      |
+-----+-----+-----+-----+-----+
| Aceh      | AC       | Banda Aceh | 4906835  | 57365.00  | 15572.50  |
| Bali      | BA       | Denpasar    | 4104900  | 5561.00   | 5461.18   |
| Banten     | BT       | Serang      | 11704877 | 9019.00   | 8398.20   |
| Bengkulu   | BE       | Bengkulu    | 1844800  | 19789.00  | 2012.91   |
| Gorontalo  | GO       | Gorontalo   | 1115633  | 11968.00  | 1406.32   |
+-----+-----+-----+-----+-----+
```

Dalam query diatas, karakter pipe (|) akan dipakai sebagai pemisah kolom.

Kita bisa menggunakan karakter apa saja sebagai pemisah kolom, termasuk karakter yang "tidak terlihat" seperti tab. Tapi bagaimana cara menulis query LOAD DATA INFILE dengan karakter tab sebagai pemisah kolom? Kita tidak bisa menggunakan FIELDS TERMINATED BY 'tab' karena disini artinya adalah string tab, bukan karakter tab.

Untuk karakter khusus seperti ini, penulisannya harus di-escape menggunakan backslash (\), dengan daftar karakter sebagai berikut:

Karakter	Keterangan karakter
\0	ASCII NUL (X'00')
\b	backspace
\n	newline (linefeed)
\r	carriage return
\t	tab
\z	ASCII 26 (Control+Z)
\N	NULL

Melihat dari tabel diatas, karakter tab dikodekan sebagai \t, sehingga jika kita ingin menjadikan tab sebagai karakter pemisah kolom, perintahnya adalah: FIELDS TERMINATED BY '\t'. Mari kita coba.

Saya akan membuat file `nama_provinsi_f.txt` dengan isi teks sebagai berikut:

`nama_provinsi_f.txt`

Jakarta	JK	Jakarta	10012271	740	79285.13
---------	----	---------	----------	-----	----------

Disini setiap kolom dipisah dengan karakter tab. Sehingga untuk menginputnya ke dalam tabel provinsi, querynya adalah sebagai berikut:

```
LOAD DATA INFILE 'C:\\xampp\\mysql\\file_belajar\\nama_provinsi_f.txt'
INTO TABLE provinsi
FIELDS TERMINATED BY '\\t';
```

```
Query OK, 1 row affected (0.06 sec)
Records: 1 Deleted: 0 Skipped: 0 Warnings: 0
```

```
SELECT * FROM provinsi;
```

nama_prov	kode_iso	ibukota	populasi	luas	apbd
Aceh	AC	Banda Aceh	4906835	57365.00	15572.50
Bali	BA	Denpasar	4104900	5561.00	5461.18
Banten	BT	Serang	11704877	9019.00	8398.20
Bengkulu	BE	Bengkulu	1844800	19789.00	2012.91
Gorontalo	GO	Gorontalo	1115633	11968.00	1406.32
Jakarta	JK	Jakarta	10012271	740.00	79285.13

Perintah FIELDS TERMINATED BY '\t' sebenarnya tidak perlu kita tulis karena secara default, query LOAD DATA INFILE akan membaca tab sebagai pemisah antar kolom, sebagaimana yang sudah kita coba di awal bab ini.

LOAD DATA INFILE ... LINES TERMINATED BY

Latihan yang telah kita coba sebelum ini baru menginput 1 baris data ke tabel provinsi. Tentu saja kita juga bisa menginput banyak baris sekaligus menggunakan query LOAD DATA INFILE.

Perintah tambahan LINES TERMINATED BY digunakan untuk memberitahu MySQL karakter apa yang akan dijadikan sebagai pemisah antar baris. Penggunaannya mirip seperti query FIELDS TERMINATED BY.

Sebagai contoh praktek, saya akan membuat file `nama_provinsi_g.txt` dengan isi teks sebagai berikut:

nama_provinsi_g.txt

Jakarta-JK-Jakarta-10012271-740-79285.13|Jambi-JA-Jambi-3344400-53509-3548.66

Dari teks diatas, tentunya anda bisa menebak bahwa karakter *min* (-) digunakan sebagai pemisah kolom, dan karakter *pipe* (|) dipakai sebagai pemisah baris.

Agar teks ini bisa dibaca oleh query LOAD DATA INFILE, querinya adalah sebagai berikut:

```
TRUNCATE provinsi;
```

```
LOAD DATA INFILE 'C:\\xampp\\mysql\\file_belajar\\nama_provinsi_g.txt'
INTO TABLE provinsi
FIELDS TERMINATED BY '-'
LINES TERMINATED BY '|';
```

```
Query OK, 2 rows affected (0.09 sec)
Records: 2 Deleted: 0 Skipped: 0 Warnings: 0
```

```
SELECT * FROM provinsi;
```

nama_prov	kode_iso	ibukota	populasi	luas	apbd
Jakarta	JK	Jakarta	10012271	740.00	79285.13
Jambi	JA	Jambi	3344400	53509.00	3548.66

Saya menggunakan query TRUNCATE provinsi untuk menghapus isi tabel provinsi. Setelah itu, query LOAD DATA INFILE akan membaca dan menginput isi teks nama_provinsi_g.txt ke dalam tabel provinsi.

Teks di dalam nama_provinsi_g.txt ditulis dalam 1 baris panjang. Ini memang tidak umum dipakai karena biasanya kita memisahkannya ke baris baru. Sebagai contoh, saya akan membuat file nama_provinsi_h.txt dengan isi teks sebagai berikut:

nama_provinsi_h.txt

Jawa,Jawa Barat,JB,Bandung,46029668,35245,2306720
 Jawa,Jawa Tengah,JT,Semarang,33522663,33987,1433687
 Jawa,Jawa Timur,JI,Surabaya,38610202,47921,1878651

Setiap kolom dipisah dengan tanda koma. Artinya kita akan menggunakan perintah FIELDS TERMINATED BY ',',. Masalahnya, karakter apa yang digunakan sebagai pemisah baris?

Untuk bisa menjawab pertanyaan ini, kita harus membahas sedikit tentang konsep “Enter” atau **baris baru** di dalam programming (dan komputer secara keseluruhan). Pemahaman ini juga akan terus terpakai ketika belajar bahasa pemrograman lain, tidak hanya di MySQL saja.

Pengertian New Line, Line Feed dan Carriage Return.

Di materi tentang *charset*, kita sudah pelajari bahwa setiap karakter yang ada di komputer dikodekan ke dalam daftar tabel ASCII. Di dalamnya termasuk karakter untuk menandakan baris baru.

Mari kita umpamakan ada 2 orang yang saling berkomunikasi dengan sandi morse, yakni A dan B. Agar komunikasi lebih lancar, A akan membaca teks yang sedang dipegangnya, mengkonversi setiap huruf menjadi sandi morse, lalu dikirim ke B menggunakan media transmisi.

Saat menerima sandi morse, B akan menerjemahkan kode tersebut ke dalam karakter latin dan menulisnya kembali ke sebuah kertas agar bisa dibaca oleh orang yang dituju.

Selama komunikasi berlangsung, A dan B perlu sebuah kode untuk “pindah baris”, yakni kode agar B menulis paragraf berikutnya di baris baru. Di dalam komputer modern, kode untuk pindah baris ini dikenal sebagai **Newline character**².

Akan tetapi, ada perbedaan pemahaman karakter *Newline* di sistem operasi modern:

- Sistem operasi lama **Mac OS** serta beberapa OS lain menggunakan karakter **Carriage Return** sebagai penanda baris baru. Karakter *carriage return* berada di nomor urut 13 dari daftar karakter ASCII. Dalam programming, *carriage return* ditulis sebagai \r.
- Sistem operasi **Mac OS, Linux, UNIX** serta beberapa OS lain, menggunakan karakter **Line Feed** sebagai penanda baris baru. Karakter *line feed* berada di nomor urut 10 dari daftar karakter ASCII. Dalam programming, *line feed* ditulis sebagai \n.
- Sistem operasi **Windows** serta beberapa OS lain menggunakan gabungan karakter **Carriage Return** dan **Line Feed** sebagai penanda baris baru. Dalam programming, ini ditulis sebagai \r\n.

Sejarah perbedaan ini berasal dari cara mengetik di mesin tik manual. Untuk pindah baris, kita harus melakukan dua hal: memutar tuas silinder, lalu mendorong silinder ke kiri. Sistem operasi Windows menerjemahkannya sebagai 2 karakter, yakni *carriage return* dan *line feed* : \r\n.

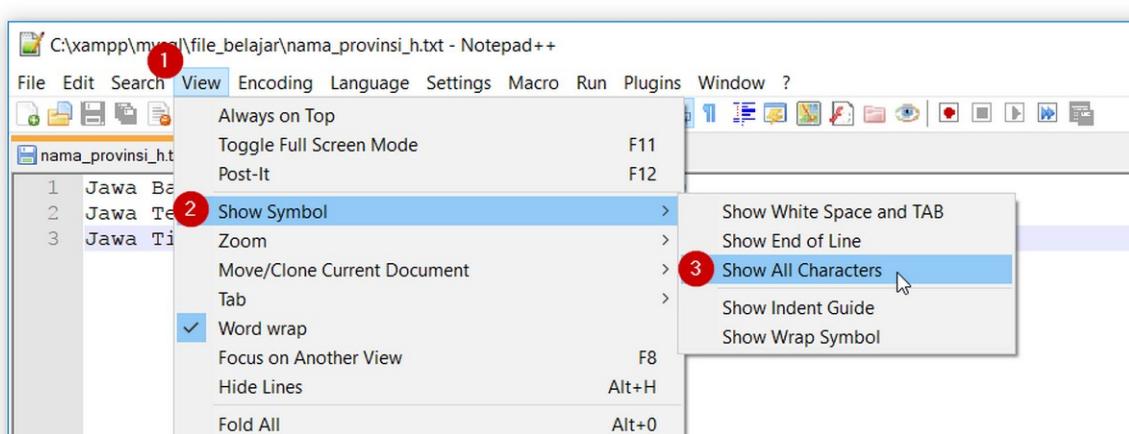
Kembali ke query LOAD DATA INFILE, kita harus periksa dulu apa karakter yang dipakai untuk membuat baris baru. Karena ada 3 kemungkinan, apakah file tersebut menggunakan \r, \n atau \r\n.

Saya yakin mayoritas dari pembaca buku ini menggunakan sistem operasi Windows, sehingga karakter penanda baris baru tersebut adalah \r\n.

Tapi agar lebih pasti, teks editor **Notepad++** bisa dipakai untuk melihat karakter New Line ini. Bahkan kita bisa membuat teks dengan karakter \r dan \n sebagai penanda baris baru dengan Notepad++ (meskipun kita berada di sistem operasi Windows).

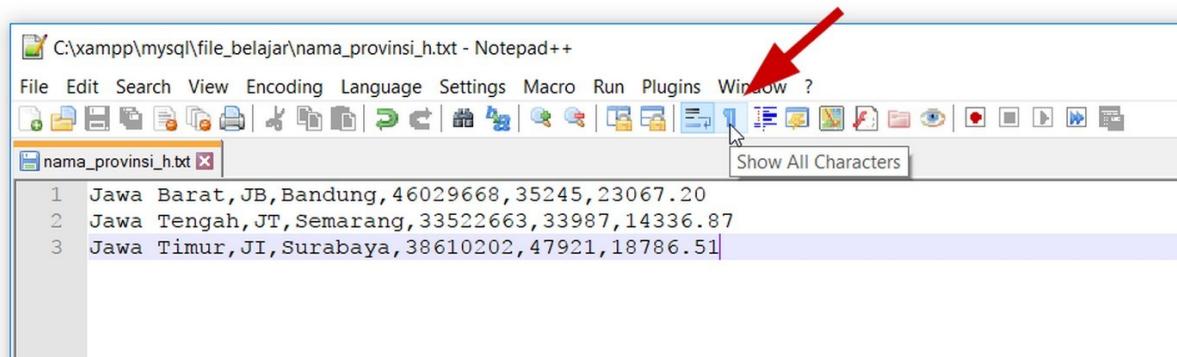
Silahkan buka teks editor **Notepad++**, lalu pilih menu **View -> Show Symbol -> Show All Character**.

²<https://en.wikipedia.org/wiki/Newline>



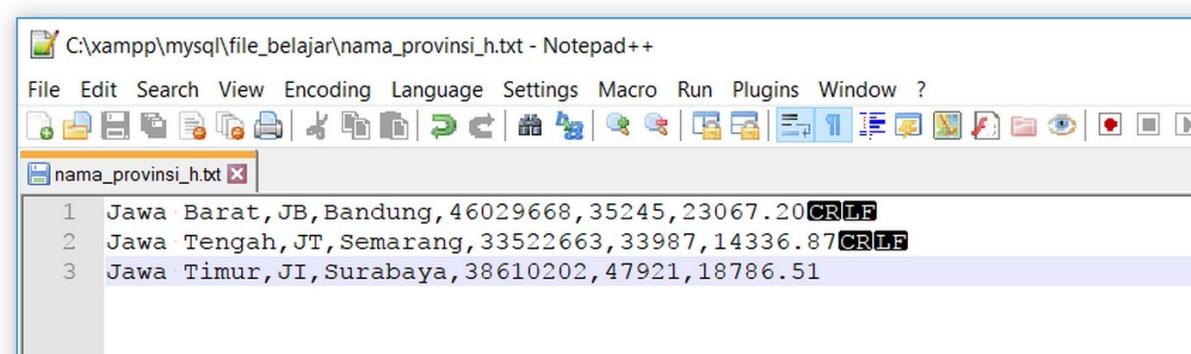
Gambar: Menu “Show All Character” di Notepad++

Atau cara lain, klik icon Show All Character seperti gambar berikut ini:



Gambar: Icon “Show All Character” di Notepad++

Di dalam teks, akan terlihat huruf CR dan LR. CR berarti *Carriage Return* dan LF berarti *Line Feed*. Hampir setiap dokumen yang dibuat di sistem operasi Windows menggunakan kedua karakter ini sebagai penanda baris baru. Secara default karakter ini memang tidak terlihat, sama halnya seperti karakter tab.



Gambar: Tampilan CR (Carriage Return) dan LR (Line Feed) di Notepad++

Sekarang, setiap kita menekan ‘Enter’ untuk membuat baris baru di Notepad++, akan terlihat tambahan CR dan LR di akhir baris. Kedua karakter ini merupakan tanda di Windows agar pindah ke baris berikutnya.

- i** Anda bisa menonaktifkan tampilan “Show All Character” dengan menghapus tanda centang di View -> Show Symbol -> Show All Character, atau klik kembali icon “Show All Character”.

Kembali ke query LOAD DATA INFILE, kita sudah mendapat informasi apa yang mesti ditulis untuk perintah LINES TERMINATED BY, yakni karakter \r\n. Dengan demikian, query yang dibutuhkan untuk menginput file nama_provinsi_h.txt ke dalam tabel provinsi adalah sebagai berikut:

```
TRUNCATE provinsi;

LOAD DATA INFILE 'C:\\xampp\\mysql\\file_belajar\\nama_provinsi_h.txt'
INTO TABLE provinsi
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\\r\\n';

Query OK, 3 rows affected (0.08 sec)
Records: 3 Deleted: 0 Skipped: 0 Warnings: 0
```

```
SELECT * FROM provinsi;
+-----+-----+-----+-----+-----+
| nama_prov | kode_iso | ibukota | populasi | luas | apbd |
+-----+-----+-----+-----+-----+
| Jawa Barat | JB | Bandung | 46029668 | 35245.00 | 23067.20 |
| Jawa Tengah | JT | Semarang | 33522663 | 33987.00 | 14336.87 |
| Jawa Timur | JI | Surabaya | 38610202 | 47921.00 | 18786.51 |
+-----+-----+-----+-----+-----+
```

Jika anda menggunakan OS Linux atau Mac OS, baris baru hanya menggunakan 1 karakter saja, yakni **Line Feed** atau \n. Query LOAD DATA INFILE menjadi seperti berikut:

```
LOAD DATA INFILE 'lokasi_file_teks.txt'
INTO TABLE provinsi
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\\n';
```

Perintah LINES TERMINATED BY '\n' merupakan nilai default jika perintah ini tidak ditulis. Karena di OS Windows baris baru ditandai dengan \r\n, maka kita harus terus menulis perintah LINES TERMINATED BY '\r\n' setiap kali menginput file teks yang terdiri dari beberapa baris.

- i** Dalam mayoritas bahasa pemrograman, tanda \n banyak dipakai sebagai penanda baris baru, misalnya dalam bahasa pemrograman C ditulis printf("Hello, World!\\n");.

LOAD DATA INFILE ... FIELDS ENCLOSED BY

Pengaturan lain yang bisa kita pakai untuk query LOAD DATA INFILE adalah FIELDS ENCLOSED BY. Perintah ini digunakan jika nilai di dalam file teks diapit oleh karakter khusus.

Sebagai contoh, saya akan membuat file `nama_provinsi_i.txt` dengan isi teks sebagai berikut:

`nama_provinsi_i.txt`

```
"Kalimantan Barat", "KB", "Pontianak", "4716093", "115114", "3929.94"
"Kalimantan Selatan", "KS", "Banjarmasin", "3922790", "36805", "6010.91"
"Kalimantan Tengah", "KT", "Palangkaraya", "2439858", "153564", "3510.91"
"Kalimantan Timur", "KI", "Samarinda", "3351432", "194849", "15731.10"
"Kalimantan Utara", "KU", "Tanjungselor", "618384", "71177", "2299.60"
```

Terlihat setiap data diapit dengan tanda kutip dua. Agar tanda ini tidak menjadi bagian dari data, kita bisa menambahkan perintah FIELDS ENCLOSED BY. Berikut cara penulisannya:

```
TRUNCATE provinsi;
```

```
LOAD DATA INFILE 'C:\\xampp\\mysql\\file_belajar\\nama_provinsi_i.txt'
INTO TABLE provinsi
FIELDS TERMINATED BY ',' ENCLOSED BY ''
LINES TERMINATED BY '\\r\\n';
```

```
Query OK, 5 rows affected (0.07 sec)
Records: 5 Deleted: 0 Skipped: 0 Warnings: 0
```

```
SELECT * FROM provinsi;
```

nama_prov	kode_iso	ibukota	populasi	luas	apbd
Kalimantan Barat	KB	Pontianak	4716093	115114.00	...
Kalimantan Selatan	KS	Banjarmasin	3922790	36805.00	...
Kalimantan Tengah	KT	Palangkaraya	2439858	153564.00	...
Kalimantan Timur	KI	Samarinda	3351432	194849.00	...
Kalimantan Utara	KU	Tanjungselor	618384	71177.00	...

Cara penulisannya digabung menjadi FIELDS TERMINATED BY ',' ENCLOSED BY ''.

LOAD DATA INFILE ... IGNORE LINES

Perintah tambahan IGNORE LINES dipakai untuk mengabaikan beberapa baris awal dari file teks. Ini sering digunakan karena biasanya di baris pertama teks terdapat nama kolom tabel. Nama kolom ini bukan bagian dari data sehingga harus dilompotki.

Sebagai contoh, saya membuat file `nama_provinsi_j.txt` dengan isi teks sebagai berikut:

nama_provinsi_j.txt

```
Provinsi,Singkatan ISO,Ibukota,Populasi,Luas Total,APBD2014
Maluku,MA,Ambon,1657409,49350,2013.83
Maluku Utara,MU,Sofifi,1138667,42960,1599.66
Nusa Tenggara Barat,NB,Mataram,4773795,19950,2884.89
Nusa Tenggara Timur,NT,Kupang,5036897,47676,2916.88
```

Perhatikan di baris pertama, terdapat nama kolom yang bukan bagian dari data. Ini bisa dilompot dengan query IGNORE LINES. Berikut cara penggunaannya:

```
TRUNCATE provinsi;
```

```
LOAD DATA INFILE 'C:\\xampp\\mysql\\file_belajar\\nama_provinsi_j.txt'
INTO TABLE provinsi
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\\r\\n'
IGNORE 1 LINES;
```

```
Query OK, 4 rows affected (0.06 sec)
Records: 4 Deleted: 0 Skipped: 0 Warnings: 0
```

```
SELECT * FROM provinsi;
```

nama_prov	kode_iso	ibukota	populasi	luas	apbd
Maluku	MA	Ambon	1657409	49350.00	2013.83
Maluku Utara	MU	Sofifi	1138667	42960.00	1599.66
Nusa Tenggara Barat	NB	Mataram	4773795	19950.00	2884.89
Nusa Tenggara Timur	NT	Kupang	5036897	47676.00	2916.88

Tambahan perintah IGNORE 1 LINES akan membuat query LOAD DATA INFILE mengabaikan 1 baris awal dari file teks. Kita bisa menggantinya dengan angka lain misalnya IGNORE 5 LINES untuk melompati 5 baris awal dari file teks.

LOAD DATA INFILE ... LINES STARTING BY

Jika perintah IGNORE LINES dipakai untuk mengabaikan beberapa baris awal dalam file teks, maka perintah LINES STARTING BY dipakai untuk mengabaikan beberapa karakter awal dari setiap baris.

Saya akan membuat file nama_provinsi_k.txt dengan isi teks sebagai berikut:

nama_provinsi_k.txt

```
##Provinsi,Singkatan ISO,Ibukota,Populasi,Luas Total,APBD2014
##
##Maluku,MA,Ambon,1657409,49350,2013.83
##Maluku Utara,MU,Sofifi,1138667,42960,1599.66
##Nusa Tenggara Barat,NB,Mataram,4773795,19950,2884.89
##Nusa Tenggara Timur,NT,Kupang,5036897,47676,2916.88
```

Setiap baris di dalam file teks ini diawali dengan karakter ##, selain itu informasi di 2 baris pertama file nama_provinsi_k.txt harus kita lompati karena bukan bagian dari data.

Berikut query yang bisa dipakai untuk membaca file nama_provinsi_k.txt diatas:

```
TRUNCATE provinsi;
```

```
LOAD DATA INFILE 'C:\xampp\mysql\file_belajar\nama_provinsi_k.txt'
INTO TABLE provinsi
FIELDS TERMINATED BY ','
LINES STARTING BY '##' TERMINATED BY '\r\n'
IGNORE 2 LINES;
```

```
Query OK, 4 rows affected (0.05 sec)
Records: 4 Deleted: 0 Skipped: 0 Warnings: 0
```

```
SELECT * FROM provinsi;
```

nama_prov	kode_iso	ibukota	populasi	luas	apbd
Maluku	MA	Ambon	1657409	49350.00	2013.83
Maluku Utara	MU	Sofifi	1138667	42960.00	1599.66
Nusa Tenggara Barat	NB	Mataram	4773795	19950.00	2884.89
Nusa Tenggara Timur	NT	Kupang	5036897	47676.00	2916.88

Perintah LINES STARTING BY '##' TERMINATED BY '\r\n' Artinya setiap baris akan diawali dengan tanda ##, serta diakhiri dengan \r\n. Saya juga menambahkan perintah IGNORE 2 LINES agar dua baris pertama tidak masuk ke dalam tabel.

LOAD DATA INFILE ... REPLACE | IGNORE

Perintah yang kita bahas kali ini tidak lagi berhubungan dengan struktur file teks, tapi mengatur apa yang terjadi jika terdapat data yang berulang di dalam tabel. Kasus seperti ini baru akan terjadi jika di dalam tabel terdapat kolom yang di set sebagai PRIMARY KEY atau menggunakan UNIQUE key.

Jika ditambahkan perintah REPLACE ke dalam query LOAD DATA INFILE, maka MySQL akan menimpa baris yang berulang tersebut dengan nilai baru. Jika ditulis IGNORE, data yang berulang akan dilompati. Jika tidak ditulis, efek default adalah MySQL akan menghasilkan pesan error saat menemukan data yang berulang.

Sebagai bahan praktik, saya akan menulis ulang tabel provinsi dengan menambahkan kolom kode_iso sebagai **primary key**:

```
DROP TABLE IF EXISTS provinsi;

CREATE TABLE provinsi (
    nama_prov VARCHAR(50),
    kode_iso CHAR(2) PRIMARY KEY,
    ibukota VARCHAR (50),
    populasi INT,
    luas DEC(8,2),
    apbd DEC(8,2)
);

DESC provinsi;
+-----+-----+-----+-----+-----+
| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| nama_prov | varchar(50) | YES |   | NULL    |       |
| kode_iso | char(2)    | NO  | PRI | NULL    |       |
| ibukota | varchar(50) | YES |   | NULL    |       |
| populasi | int(11)   | YES |   | NULL    |       |
| luas     | decimal(8,2) | YES |   | NULL    |       |
| apbd    | decimal(8,2) | YES |   | NULL    |       |
+-----+-----+-----+-----+-----+
```

Kali ini kolom kode_iso saya set sebagai *primary key* sehingga tidak boleh ada nilai berulang yang diinput.

Untuk file teks, saya akan menggunakan `nama_provinsi_1.txt` dengan isi sebagai berikut:

`nama_provinsi_1.txt`

```
Sulawesi Barat,SR,Mamuju,1258090,16787,1388.27
Sulawesi Selatan,SN,Makassar,8432163,46116,6186.39
Sulawesi Tengah,ST,Palu,2831283,68090,2514.84
Sulawesi Tenggara,SG,Kendari,2448081,36757,2416.99
Sulawesi Utara,SA,Manado,2386604,13931,2625.93
```

Sebelum kita *import* isi file ini, saya akan menginput satu data ke dalam tabel provinsi:

```
INSERT INTO provinsi VALUES ('Sulawesi Tengah', 'ST', 'Palu', NULL, NULL, NULL);

SELECT * FROM provinsi;
+-----+-----+-----+-----+-----+
| nama_prov | kode_iso | ibukota | populasi | luas | apbd |
+-----+-----+-----+-----+-----+
| Sulawesi Tengah | ST | Palu | NULL | NULL | NULL |
+-----+-----+-----+-----+-----+
```

Sekarang, mari kita test input data dari file `nama_provinsi_k.txt`:

```
LOAD DATA INFILE 'C:\\xampp\\mysql\\file_belajar\\nama_provinsi_1.txt'
INTO TABLE provinsi
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\\r\\n';
```

```
ERROR 1062 (23000): Duplicate entry 'ST' for key 'PRIMARY'
```

Pesan error muncul karena ditemukan duplikasi data ST untuk kolom `kode_iso`. Hasilnya, tidak ada data baru yang input ke tabel `provinsi`. Ini merupakan prilaku default dari query `LOAD DATA INFILE` dalam menangani data yang berulang.

Jika ditambahkan perintah `REPLACE`, baris yang bentrok tersebut akan diupdate dengan nilai baru yang berasal dari file teks. Mari kita coba:

```
LOAD DATA INFILE 'C:\\xampp\\mysql\\file_belajar\\nama_provinsi_1.txt'
REPLACE INTO TABLE provinsi
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\\r\\n';
```

```
Query OK, 6 rows affected (0.10 sec)
Records: 5 Deleted: 1 Skipped: 0 Warnings: 0
```

```
SELECT * FROM provinsi;
+-----+-----+-----+-----+-----+
| nama_prov | kode_iso | ibukota | populasi | luas | apbd |
+-----+-----+-----+-----+-----+
| Sulawesi Utara | SA | Manado | 2386604 | 13931.00 | 2625.93 |
| Sulawesi Tenggara | SG | Kendari | 2448081 | 36757.00 | 2416.99 |
| Sulawesi Selatan | SN | Makassar | 8432163 | 46116.00 | 6186.39 |
| Sulawesi Barat | SR | Mamuju | 1258090 | 16787.00 | 1388.27 |
| Sulawesi Tengah | ST | Palu | 2831283 | 68090.00 | 2514.84 |
+-----+-----+-----+-----+-----+
```

Sekarang file teks `nama_provinsi_1.txt` berhasil diinput ke dalam tabel `provinsi`. Khusus untuk baris provinsi Sulawesi Tengah, kolom yang sebelumnya berisi nilai `NULL` sudah diupdate sesuai data dari file `nama_provinsi_1.txt`.

Apabila perintah yang dipakai adalah IGNORE, maka baris yang bentrok tersebut akan diloloskan (diabaikan) seperti contoh berikut:

```
TRUNCATE provinsi;

INSERT INTO provinsi VALUES ('Sulawesi Tengah', 'ST', 'Palu', NULL, NULL, NULL);

SELECT * FROM provinsi;
+-----+-----+-----+-----+-----+
| nama_prov | kode_iso | ibukota | populasi | luas | apbd |
+-----+-----+-----+-----+-----+
| Sulawesi Tengah | ST | Palu | NULL | NULL | NULL |
+-----+-----+-----+-----+-----+

LOAD DATA INFILE 'C:\\xampp\\mysql\\file_belajar\\nama_provinsi_1.txt'
IGNORE INTO TABLE provinsi
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\\r\\n';

Query OK, 4 rows affected, 1 warning (0.12 sec)
Records: 5 Deleted: 0 Skipped: 1 Warnings: 1

SELECT * FROM provinsi;
+-----+-----+-----+-----+-----+
| nama_prov | kode_iso | ibukota | populasi | luas | apbd |
+-----+-----+-----+-----+-----+
| Sulawesi Utara | SA | Manado | 2386604 | 13931.00 | 2625.93 |
| Sulawesi Tenggara | SG | Kendari | 2448081 | 36757.00 | 2416.99 |
| Sulawesi Selatan | SN | Makassar | 8432163 | 46116.00 | 6186.39 |
| Sulawesi Barat | SR | Mamuju | 1258090 | 16787.00 | 1388.27 |
| Sulawesi Tengah | ST | Palu | NULL | NULL | NULL |
+-----+-----+-----+-----+-----+
```

Kali ini baris Sulawesi Tengah tidak akan diupdate, dimana MySQL akan meloloskan teks tersebut sesuai dengan perintah IGNORE.

LOAD DATA INFILE ... (col1, col2, col3)

Dari semua contoh query LOAD DATA INFILE yang sudah kita jalankan, jumlah kolom pada file teks harus sama dengan jumlah kolom di tabel provinsi. Akan tetapi kita juga bisa menentukan kolom mana saja yang akan diisi dan kolom mana yang boleh diabaikan. Caranya, tulis nama kolom diakhir query LOAD DATA INFILE.

Sebagai contoh praktek, saya memiliki file `nama_provinsi_m.txt` dengan isi teks sebagai berikut:

nama_provinsi_m.txt

```
Provinsi;Singkatan ISO;Ibukota;APBD 2014 (miliar rupiah)
#####

```

```
Sumatera Barat,SB,Padang,3887.84
Sumatera Selatan,SS,Palembang,7697.99
Sumatera Utara,SU,Medan,8565.53

```

Isi file teks ini hanya menyediakan 4 kolom data, padahal tabel provinsi memiliki 6 kolom. Jika diinput langsung ke dalam tabel, akan tampil pesan error karena jumlah kolom tidak sesuai.

Agar teks ini bisa diinput ke dalam tabel provinsi, kita harus tulis kolom apa saja yang akan menampung ke-4 nilai tersebut. Berikut contoh penggunaannya:

```
TRUNCATE provinsi;
```

```
LOAD DATA INFILE 'C:\\xampp\\mysql\\file_belajar\\nama_provinsi_m.txt'
INTO TABLE provinsi
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\\r\\n'
IGNORE 2 LINES (nama_prov, kode_iso, ibukota, apbd);
```

```
Query OK, 3 rows affected (0.07 sec)
Records: 3 Deleted: 0 Skipped: 0 Warnings: 0
```

```
SELECT * FROM provinsi;
```

nama_prov	kode_iso	ibukota	populasi	luas	apbd
Sumatera Barat	SB	Padang	NULL	NULL	3887.84
Sumatera Selatan	SS	Palembang	NULL	NULL	7697.99
Sumatera Utara	SU	Medan	NULL	NULL	8565.53

Dengan menuliskan 4 nama kolom di akhir query LOAD DATA INFILE, yakni baris (nama_prov, kode_iso, ibukota, apbd), ke-4 nilai yang ada di file nama_provinsi_m.txt akan diinput ke dalam 4 kolom yang sudah dituliskan.

Untuk kolom yang tidak ada nilai akan di set ke nilai default atau NULL jika kolom tersebut tidak memiliki nilai default.

18.2 Import Data dari File CSV

File CSV (*Comma Separated Values*) merupakan salah satu file yang sering dipakai untuk proses export dan import data. File ini memiliki format mirip seperti file teks yang sudah kita coba sepanjang bab ini.

File CSV bisa dibuat menggunakan aplikasi *spreadsheet* seperti **Microsoft Excel** atau aplikasi sejenis. Selama isinya berupa file teks yang dipisah-pisah dengan karakter tertentu (biasanya karakter koma), file CSV bisa dibaca oleh query `LOAD DATA INFILE`.

Sebagai contoh kasus, saya akan jabarkan cara input data secara massal (*bulk*) yang di dapat dari internet. Contoh ini bisa anda terapkan jika ingin menginput banyak nilai ke dalam tabel MySQL.

Saya akan menginput data tabel yang di dapat dari halaman wikipedia berikut: [Daftar provinsi di Indonesia³](https://id.wikipedia.org/wiki/Daftar_provinsi_di_Indonesia).

Pulau	Provinsi	Singkatan ISO	Ibukota	Diresmikan sebagai Provinsi	Populasi (Proyeksi BPS 2014)	Luas Total (km²)	Kepadatan penduduk (Populasi / Luas; jiwa / km²)	APBD 2014 (miliar rupiah)	PDRB 2014 (triliun rupiah)
Sumatera	Aceh	AC	Banda Aceh	7 Desember 1959	4.906.835	57.365	86	15.572,50	130,45
Kepulauan Nusa Tenggara	Bali	BA	Denpasar	14 Agustus 1958	4.104.900	5.561	738	5.461,18	156,45

Gambar: Data seluruh provinsi yang ada di Indonesia

Dalam halaman ini, terdapat tabel besar yang berisi data provinsi yang ada di Indonesia. Data ini sebenarnya sudah kita pakai sepanjang pembahasan query `LOAD DATA INFILE`.

Untuk beberapa situs, ada yang sudah menyediakan link untuk mendownload file dalam format CSV. Tapi kalau tidak, kita terpaksa buat secara manual.

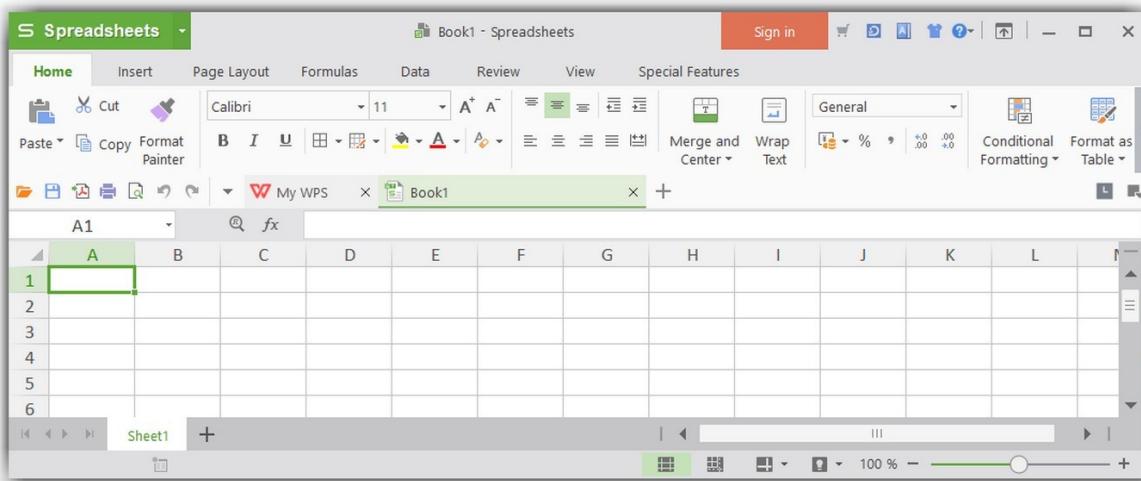
Untuk membuat file CSV, bisa menggunakan aplikasi spreadsheet seperti **Microsoft Excel**. Sebagai alternatif, saya menggunakan aplikasi **WPS Spreadsheets** yang menjadi bagian dari **WPS Office**.

WPS Office sendiri adalah alternatif aplikasi office yang sangat mirip seperti **Microsoft Office**. Di dalamnya terdapat **WPS Writer** (mirip Microsoft Word), **WPS Spreadsheets** (mirip Microsoft Excel), serta **WPS Presentation** (mirip seperti Microsoft Power Point).

WPS Office bisa dipakai secara gratis dengan beberapa batasan (plus iklan yang tidak terlalu mengganggu). Aplikasi ini layak dicoba sebagai pengganti Microsoft Office, terutama untuk menghindari penggunaan aplikasi bajakan (agar lebih nyaman dan lebih berkah).

Jika tertarik, WPS Office bisa di download dari situs resminya di: <https://www.wps.com/office-free>.

³https://id.wikipedia.org/wiki/Daftar_provinsi_di_Indonesia



Gambar: Tampilan WPS Spreadsheets

Baik, kembali ke halaman wikipedia, saya akan *blok* seluruh isi tabel (klik dan tahan tombol mouse). Untuk mem-blok seluruh tabel, mulai dari kiri atas tabel, yakni diatas tabel sampai kanan bawah. Pastikan seluruh tabel sudah ter-seleksi.

Pulau	Provinsi	Singkatan ISO	Ibukota	Diresmikan sebagai Provinsi	Populasi (Proyeksi BPS 2014)	Luas Total (km²)	Kepadatan penduduk (Populasi / Luas; jiwa / km²)	APE 2014 (mil rupiah)
Jawa	Yogyakarta	YO	Yogyakarta	4 Maret 1950	3.553.100	3.133	1.134	3.662
Kalimantan	Kalimantan Utara	KU	Tanjungselor	25 Oktober 2012	618.384	71.177	9	2.299
Sumatera	Kepulauan Riau	KR	Tanjung Pinang	25 Oktober 2002	1.917.415	8.084	237	3.944
Jawa	Jawa Timur	JI	Surabaya	4 Juli 1950	38.610.202	47.921	806	18.786
Kepulauan Maluku	Maluku Utara	MU	Sofifi	4 Oktober 1999	1.138.667	42.960	27	1.599

Gambar: Blok tabel dari kiri atas ke kanan bawah

Lalu *paste* isian tabel ke Notepad++ terlebih dahulu karena ada beberapa bagian yang perlu dirapikan.

	Pulau	Provinsi	Singkatan ISO	Ibukota	Diresmikan sebagai Provinsi	Populasi	(APBD)
1	Pulau	Provinsi	Singkatan ISO	Ibukota	Diresmikan sebagai Provinsi	Populasi	(APBD)
2	Pulau	Provinsi	Singkatan ISO	Ibukota	Diresmikan sebagai Provinsi	Populasi	(APBD)
3	(Proyeksi BPS 2014)	YO	Yogyakarta	4 Maret 1950	3.553.100	3.133	1.134
4	Jawa	Yogyakarta	YO	Yogyakarta	4 Maret 1950	3.553.100	3.133
5	Kalimantan	Kalimantan Utara	KU	Tanjungselor	25 Oktober 2012	618.384	71.177
6	Sumatera	Kepulauan Riau	KR	Tanjung Pinang	25 Oktober 2002	1.917.415	8.084
7	Jawa	Jawa Timur	JI	Surabaya	4 Juli 1950	38.610.202	47.921
8	Kepulauan Maluku	Maluku Utara	MU	Sofifi	4 Oktober 1999	1.138.667	42.960
9	Jawa	Banten	BT	Serang	4 Oktober 2000	11.704.877	9.019
10	Jawa	Jawa Tengah	JT	Semarang	4 Juli 1950	33.522.663	33.987
11	Kalimantan	Kalimantan Timur	KI	Samarinda	2 Juli 1958	3.351.432	194.849

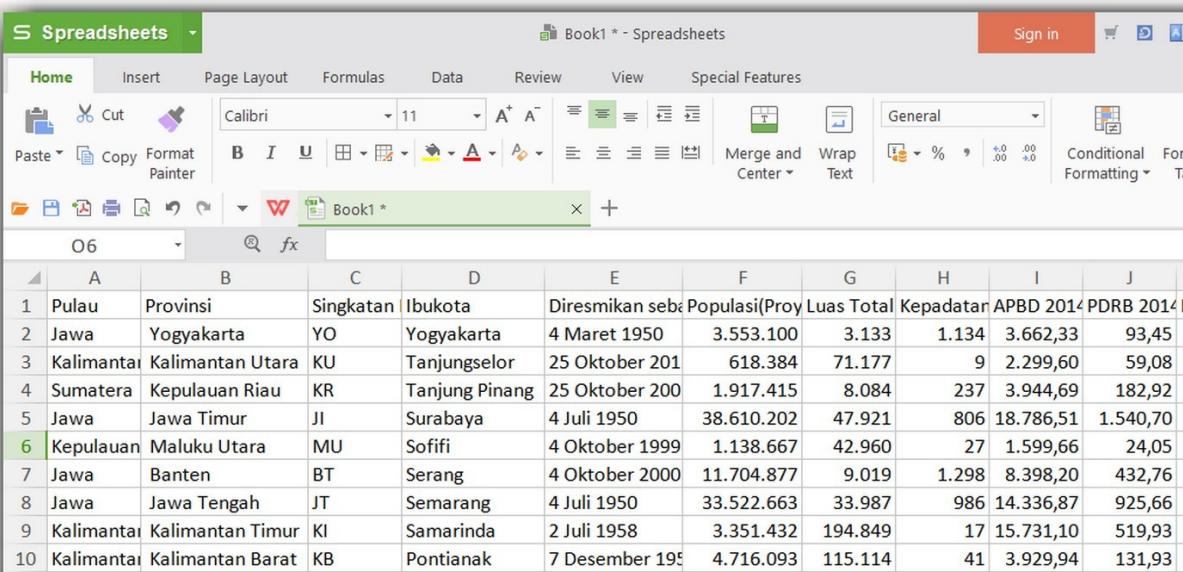
Gambar: Hapus baris pertama dan naikkan baris 3 ke baris 2

Pertama, hapus baris teks “Daftar Provinsi” di baris pertama (1), kemudian naikkan baris ketiga ke baris kedua (2). Ini supaya judul kolom memanjang dalam 1 baris dan sesuai dengan tiap kolom di bawahnya. Hasil akhir adalah sebagai berikut:

1	Pulau	Provinsi	Singkatan ISO	Ibukota	Diresmikan sebagai Provinsi	Populasi	(APBD)
2	Jawa	Yogyakarta	YO	Yogyakarta	4 Maret 1950	3.553.100	3.133
3	Kalimantan	Kalimantan Utara	KU	Tanjungselor	25 Oktober 2012	618.384	71.177
4	Sumatera	Kepulauan Riau	KR	Tanjung Pinang	25 Oktober 2002	1.917.415	8.084
5	Jawa	Jawa Timur	JI	Surabaya	4 Juli 1950	38.610.202	47.921
6	Kepulauan Maluku	Maluku Utara	MU	Sofifi	4 Oktober 1999	1.138.667	42.960
7	Jawa	Banten	BT	Serang	4 Oktober 2000	11.704.877	9.019
8	Jawa	Jawa Tengah	JT	Semarang	4 Juli 1950	33.522.663	33.987
9	Kalimantan	Kalimantan Timur	KI	Samarinda	2 Juli 1958	3.351.432	194.849

Gambar: Baris pertama berisi judul dari setiap kolom

Selanjutnya, blok seluruh teks di Notepad++ lalu paste-kan ke WPS Spreadsheet:

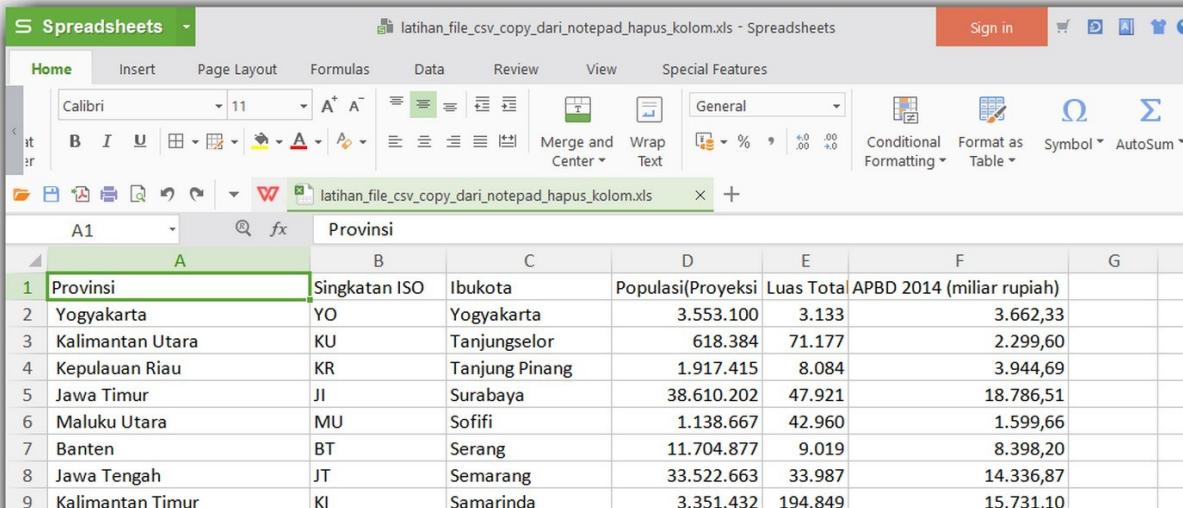


	A	B	C	D	E	F	G	H	I	J
1	Pulau	Provinsi	Singkatan	Ibukota	Diresmikan seb:	Populasi(Proy)	Luas Total	Kepadatan	APBD 2014	PDRB 2014
2	Jawa	Yogyakarta	YO	Yogyakarta	4 Maret 1950	3.553.100	3.133	1.134	3.662,33	93,45
3	Kalimanta	Kalimantan Utara	KU	Tanjungselor	25 Oktober 201	618.384	71.177	9	2.299,60	59,08
4	Sumatera	Kepulauan Riau	KR	Tanjung Pinang	25 Oktober 200	1.917.415	8.084	237	3.944,69	182,92
5	Jawa	Jawa Timur	JI	Surabaya	4 Juli 1950	38.610.202	47.921	806	18.786,51	1.540,70
6	Kepulauan	Maluku Utara	MU	Sofifi	4 Oktober 1999	1.138.667	42.960	27	1.599,66	24,05
7	Jawa	Banten	BT	Serang	4 Oktober 2000	11.704.877	9.019	1.298	8.398,20	432,76
8	Jawa	Jawa Tengah	JT	Semarang	4 Juli 1950	33.522.663	33.987	986	14.336,87	925,66
9	Kalimanta	Kalimantan Timur	KI	Samarinda	2 Juli 1958	3.351.432	194.849	17	15.731,10	519,93
10	Kalimanta	Kalimantan Barat	KB	Pontianak	7 Desember 195	4.716.093	115.114	41	3.929,94	131,93

Gambar: Tampilan di dalam WPS Spreadsheet

Sampai disini, data kita sudah terlihat seperti tabel. Anda bisa memberi bingkai atau men-format tampilan agar lebih rapi, namun kita tidak butuh efek tampilan karena yang diperlukan hanya data saja.

Untuk tabel provinsi, saya tidak butuh semua kolom, jadi beberapa kolom yang tidak perlu akan dihapus. Jika ada data yang ingin diubah juga bisa diedit disini, misalnya ingin menambah beberapa baris atau kolom baru.

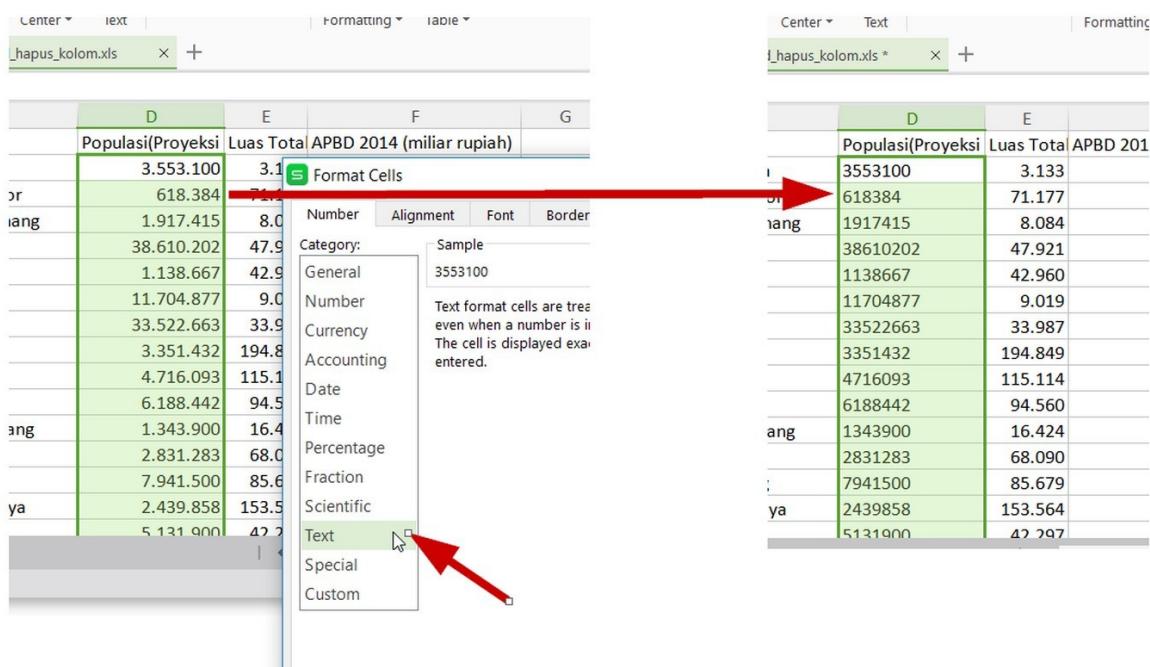


	A	B	C	D	E	F	G
1	Provinsi	Singkatan ISO	Ibukota	Populasi(Proyeksi)	Luas Total	APBD 2014 (miliar rupiah)	
2	Yogyakarta	YO	Yogyakarta	3.553.100	3.133	3.662,33	
3	Kalimantan Utara	KU	Tanjungselor	618.384	71.177	2.299,60	
4	Kepulauan Riau	KR	Tanjung Pinang	1.917.415	8.084	3.944,69	
5	Jawa Timur	JI	Surabaya	38.610.202	47.921	18.786,51	
6	Maluku Utara	MU	Sofifi	1.138.667	42.960	1.599,66	
7	Banten	BT	Serang	11.704.877	9.019	8.398,20	
8	Jawa Tengah	JT	Semarang	33.522.663	33.987	14.336,87	
9	Kalimantan Timur	KI	Samarinda	3.351.432	194.849	15.731,10	

Gambar: Sisa 6 kolom yang diperlukan untuk tabel provinsi

Setelah menghapus kolom yang tidak diperlukan, langkah berikutnya adalah menyamakan format tampilan data. Seperti yang anda lihat dari tampilan diatas, untuk kolom populasi, luas dan apbd terdapat tanda titik sebagai pemisah ribuan. Selain itu kolom ini juga menggunakan tanda koma sebagai pemisah pecahan.

Format ini tidak sesuai dengan MySQL. Di dalam MySQL, tanda titik untuk angka akan dianggap sebagai pemisah pecahan. Kita harus edit agar datanya sesuai dengan aturan MySQL.



Gambar: Mengubah format tampilan agar tidak ditampilkan dengan pemisah ribuan

Cara “pembersihan” ini bergantung kepada jenis data serta aplikasi yang dipakai. Dalam kasus kali ini, saya cukup mengubah format tampilan kolom angka menjadi teks. Hasil akhirnya berbentuk seperti ini:

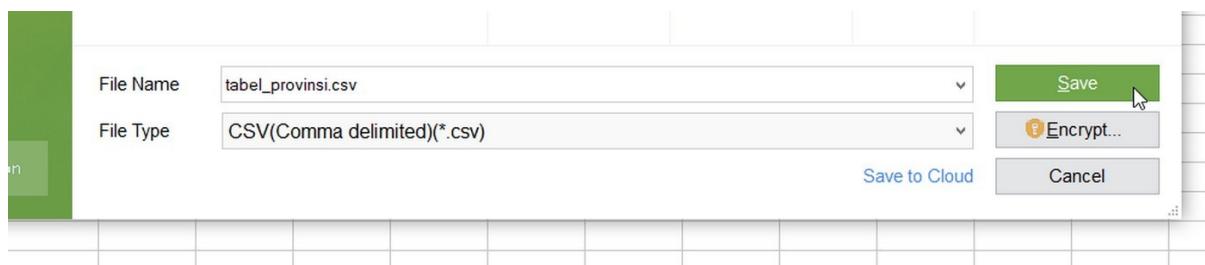
The screenshot shows a WPS Spreadsheets window with a toolbar at the top. The data is presented in a table with columns A through F. The numbers in the columns now lack the thousands separator, appearing as 3553100, 618384, etc. The table structure is identical to the one in the previous Excel screenshot.

A	B	C	D	E	F
1 Provinsi	Singkatan ISO	Ibukota	Populasi(Proyeksi)	Luas Total APBD 2014 (miliar rupiah)	
2 Yogyakarta	YO	Yogyakarta	3553100	3133	3662.33
3 Kalimantan Utara	KU	Tanjungselor	618384	71177	2299.6
4 Kepulauan Riau	KR	Tanjung Pinang	1917415	8084	3944.69
5 Jawa Timur	JI	Surabaya	38610202	47921	18786.51
6 Maluku Utara	MU	Sofifi	1138667	42960	1599.66
7 Banten	BT	Serang	11704877	9019	8398.2
8 Jawa Tengah	JT	Semarang	33522663	33987	14336.87
9 Kalimantan Timur	KI	Samarinda	3351432	194849	15731.1
10 Kalimantan Barat	KB	Pontianak	4716093	115114	3929.94
11 Riau	RI	Pekanbaru	6188442	94560	9425.83
12 Kepulauan Bangka Belitung	BB	Pangkalpinang	1343900	16424	2276.18
13 Sulawesi Tengah	ST	Palu	2831283	68090	2514.84
14 Sumatera Selatan	SS	Palembang	7941500	85679	7697.99
15 Kalimantan Tengah	KT	Balikpapan	2120950	152561	2510.01

Gambar: Pembersihan data sudah selesai

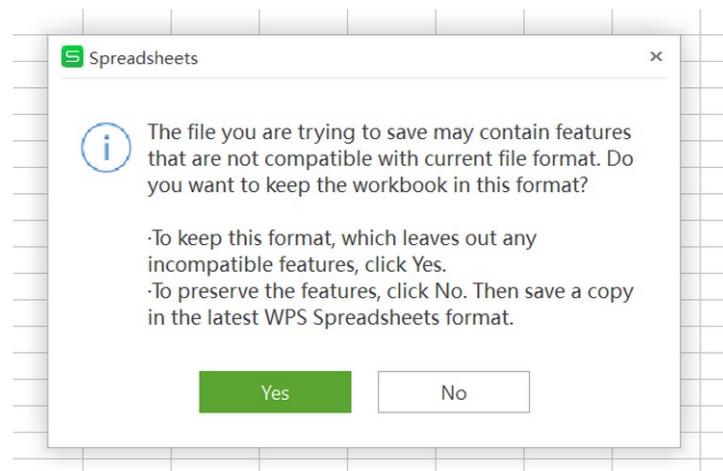
Langkah terakhir di aplikasi WPS Spreadsheets adalah men-save data diatas sebagai file CSV.

Pilihan ini ada di kolom “File Type” pada jendela “Save As”. Saya akan menyimpan tabel ini sebagai tabel_provinsi.csv ke dalam folder file_belajar.



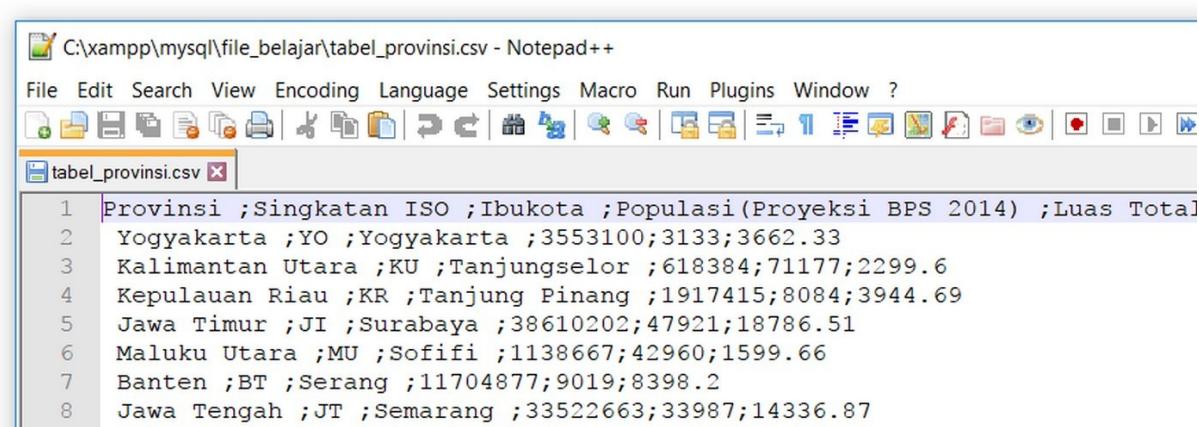
Gambar: Simpan sebagai tabel_provinsi.csv

Pada saat menyimpan file sebagai CSV, bisa jadi ada jendela peringatan bahwa file CSV tidak mendukung beberapa fitur dari WPS. Klik saja tombol YES.



Gambar: Jendela peringatan keterbatasan fitur di file CSV

Sekarang file CSV kita sudah selesai. Untuk memeriksa isinya, silahkan buka tabel_provinsi.csv menggunakan Notepad++.



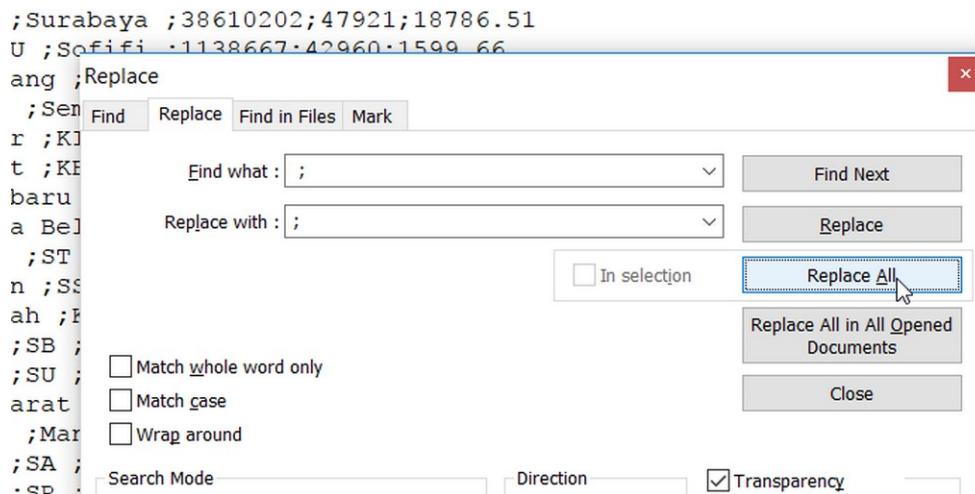
Gambar: Tampilan file tabel_provinsi.csv di Notepad++

Di Notepad++ terlihat tampilan teks seperti yang sudah kita praktekkan sepanjang pembahasan query LOAD DATA INFILE. WPS Spreadsheets menggunakan tanda semi colon atau titik koma (

;) sebagai pemisah antar kolom.

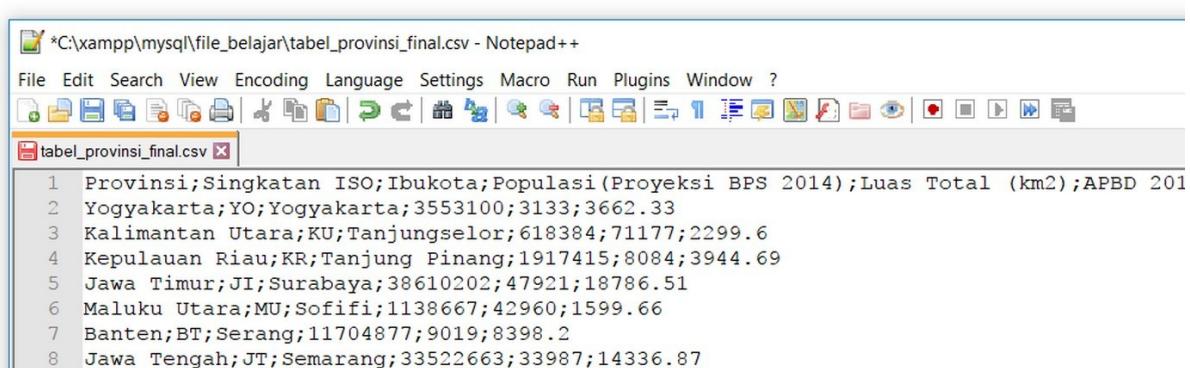
Biasanya file CSV menggunakan tanda koma (,) sebagai pemisah kolom, akan tetapi hal ini tidak menjadi masalah karena kita tinggal menyesuaikan baris perintah FIELDS TERMINATED BY.

Dari tampilan diatas juga terlihat bahwa terdapat tambahan spasi di awal dan akhir data untuk beberapa kolom. Tambahan spasi ini bisa menjadi masalah karena akan ikut menjadi bagian dari data. Proses penghapusan spasi bisa lebih mudah menggunakan fitur pencarian (*find and replace*) dari Notepad++, atau bisa juga dihapus secara manual.



Gambar: Cari semua karakter spasi+titik koma, kemudian ganti menjadi titik koma saja

File yang sudah bersih dari spasi akan saya simpan sebagai tabel_provinsi_final.csv



Gambar: Tampilan file tabel_provinsi_final.csv di Notepad++

Langkah terakhir, jalankan query LOAD DATA INFILE. Tentu saja kita harus membuat tabel provinsi terlebih dahulu:

```

DROP TABLE IF EXISTS provinsi;
Query OK, 0 rows affected (0.33 sec)

CREATE TABLE provinsi (
    nama_prov VARCHAR(50),
    kode_iso CHAR(2),
    ibukota VARCHAR (50),
    populasi INT,
    luas DEC(8,2),
    apbd DEC(8,2)
);
Query OK, 0 rows affected (0.18 sec)

LOAD DATA INFILE 'C:\\xampp\\mysql\\file_belajar\\tabel_provinsi_final.csv'
INTO TABLE provinsi
FIELDS TERMINATED BY ';'
LINES TERMINATED BY '\\r\\n'
IGNORE 1 LINES;

Query OK, 34 rows affected (0.14 sec)
Records: 34 Deleted: 0 Skipped: 0 Warnings: 0

SELECT * FROM provinsi;
+-----+-----+-----+-----+-----+
| nama_prov | kode_iso | ibukota | populasi | luas |
+-----+-----+-----+-----+-----+
| Yogyakarta | YO | Yogyakarta | 3553100 | 3133.00 | ... |
| Kalimantan Utara | KU | Tanjungselor | 618384 | 71177.00 | ... |
| Kepulauan Riau | KR | Tanjung Pinang | 1917415 | 8084.00 | ... |
| Jawa Timur | JI | Surabaya | 38610202 | 47921.00 | ... |
| ... | .. | ... | ... | ... | ... |
| Jawa Barat | JB | Bandung | 46029668 | 35245.00 | ... |
| Lampung | LA | Bandar Lampung | 8026191 | 35376.00 | ... |
| Aceh | AC | Banda Aceh | 4906835 | 57365.00 | ... |
| Maluku | MA | Ambon | 1657409 | 49350.00 | ... |
+-----+-----+-----+-----+-----+
34 rows in set (0.00 sec)

```

Done! Kita sudah berhasil meng-import data external ke dalam tabel MySQL.

Prosesnya memang sedikit panjang karena kita perlu menyusun data agar sesuai dengan tipe data MySQL. Ini bisa jauh lebih cepat jika file CSV yang sudah terformat sesuai bentuk tipe data MySQL.



Jika anda butuh data random / acak dalam format CSV bisa kunjungi situs-situs data generator seperti [mockaroo⁴](http://www.mockaroo.com), [freedatagenerator⁵](http://www.freedatagenerator.com), atau [generatedata⁶](https://www.generatedata.com).

18.3 Export Data dengan SELECT ... INTO OUTFILE

Query `SELECT...INTO OUTFILE` merupakan kebalikan dari query `LOAD DATA INFILE`. Query ini berfungsi untuk membuat atau *menggenerate* file teks yang berasal dari tabel MySQL. Berikut format dasar dari query `SELECT...INTO OUTFILE`:

```
SELECT nama_kolom1, nama_kolom2, ...
INTO OUTFILE 'lokasi_file', [setting...]
FROM nama_tabel
```

Kita bisa menentukan kolom apa saja yang akan dipilih. Nama kolom ditulis setelah perintah `SELECT`.

Berbagai perintah tambahan yang sudah kita bahas di query `LOAD DATA INFILE` juga bisa dipakai untuk query `SELECT...INTO OUTFILE` ini, yang dituliskan di bagian `[setting...]`.

Sebagai contoh praktik, saya akan men-export isi dari tabel `provinsi` ke dalam file `hasil_provinsi_a.txt`. Berikut query yang digunakan:

```
SELECT * INTO OUTFILE 'C:\\xampp\\mysql\\file_belajar\\hasil_provinsi_a.txt'
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY ''
LINES TERMINATED BY '\\r\\n'
FROM provinsi;
```

Query OK, 34 rows affected (0.00 sec)

Query `SELECT * INTO OUTFILE` akan memilih seluruh kolom di dalam tabel `provinsi`. File `hasil_provinsi_a.txt` nantinya berada di folder `file_belajar`, sesuai dengan alamat file yang dituliskan.

Di baris kedua, terdapat perintah `FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '' LINES TERMINATED BY '\\r\\n'`. Artinya, kolom tabel akan dipisahkan dengan tanda koma, kemudian karakter `\r\n` dipakai sebagai pemisah antar baris.

Khusus untuk perintah `OPTIONALLY ENCLOSED BY ''`, artinya hasil teks untuk data dengan tipe *string* akan diapit dengan tanda kutip dua, sedangkan tipe data *number* tidak. Prilaku seperti di dapat dengan tambahan perintah `OPTIONALLY`.

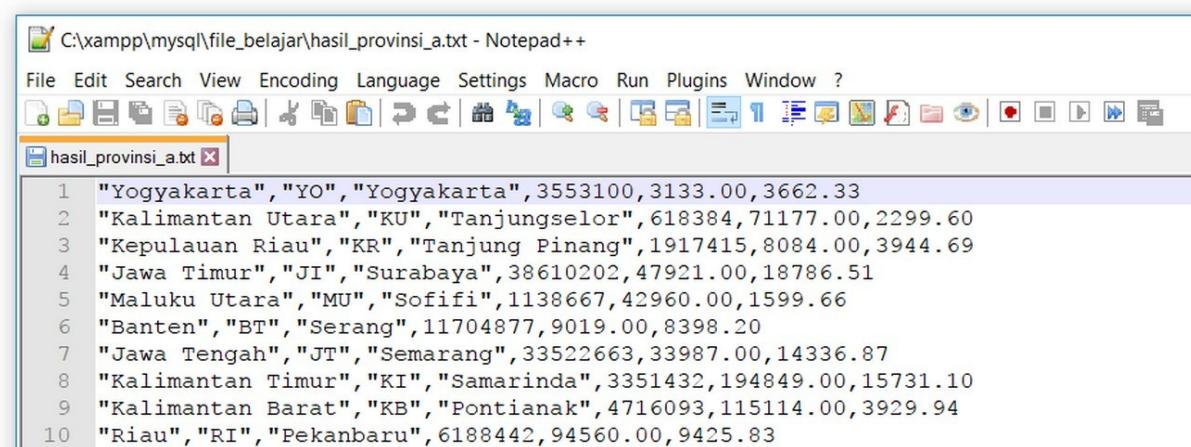
Seandainya saya menggunakan perintah `ENCLOSED BY ''` saja tanpa `OPTIONALLY`, maka seluruh data akan diapit dengan tanda kutip dua, termasuk tipe data *number*.

⁴<http://www.mockaroo.com>

⁵<http://www.freedatagenerator.com>

⁶<https://www.generatedata.com>

Sebagai pembuktian, silahkan buka folder `file_belajar`, dan cari file `hasil_provinsi_a.txt` hasil query diatas.



```

C:\xampp\mysql\file_belajar\hasil_provinsi_a.txt - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
hasil_provinsi_a.txt
1 "Yogyakarta", "YO", "Yogyakarta", 3553100, 3133.00, 3662.33
2 "Kalimantan Utara", "KU", "Tanjungselor", 618384, 71177.00, 2299.60
3 "Kepulauan Riau", "KR", "Tanjung Pinang", 1917415, 8084.00, 3944.69
4 "Jawa Timur", "JI", "Surabaya", 38610202, 47921.00, 18786.51
5 "Maluku Utara", "MU", "Sofifi", 1138667, 42960.00, 1599.66
6 "Banten", "BT", "Serang", 11704877, 9019.00, 8398.20
7 "Jawa Tengah", "JT", "Semarang", 33522663, 33987.00, 14336.87
8 "Kalimantan Timur", "KI", "Samarinda", 3351432, 194849.00, 15731.10
9 "Kalimantan Barat", "KB", "Pontianak", 4716093, 115114.00, 3929.94
10 "Riau", "RI", "Pekanbaru", 6188442, 94560.00, 9425.83

```

Gambar: Tampilan isi file `hasil_provinsi_a.txt` dari query `SELECT...INTO OUTFILE`

Terlihat setiap data kolom dipisah dengan tanda koma. Setiap baris berada di baris baru (efek dari '`\r\n`'). Serta data dengan tipe data *string* akan diapit dengan tanda kutip dua.

Untuk contoh kedua, saya akan membuat file `hasil_provinsi_b.txt` dengan query sebagai berikut:

```

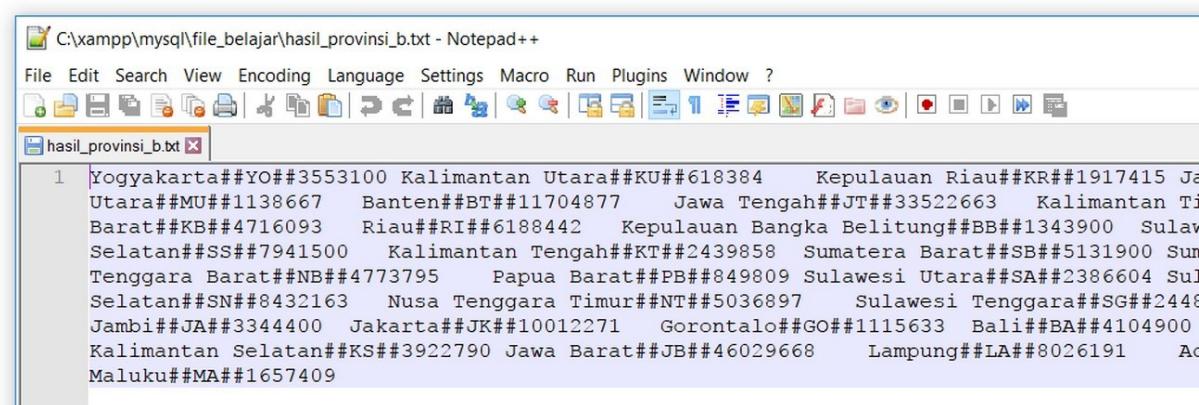
SELECT nama_prov, kode_iso, populasi
INTO OUTFILE 'C:\\xampp\\mysql\\file_belajar\\hasil_provinsi_b.txt'
FIELDS TERMINATED BY '##'
LINES TERMINATED BY '\\t'
FROM provinsi;

```

Query OK, 34 rows affected (0.00 sec)

Disini saya hanya memilih 3 kolom saja dari tabel provinsi. Tanda ## dipakai sebagai pemisah antar kolom, dan setiap baris dipisah dengan karakter tab `\t`.

Hasilnya, file `hasil_provinsi_b.txt` terdiri dari 1 baris panjang. Hanya saja Notepad++ akan menampilkan baris tersebut ke dalam beberapa baris karena fitur “word wrap” aktif secara default. Fitur ini otomatis menampilkan baris yang terlalu panjang ke baris di bawahnya dan terlihat menjadi beberapa baris (meskipun itu sebenarnya hanya 1 baris).



Gambar: Tampilan isi file `hasil_provinsi_b.txt` dari query `SELECT...INTO OUTFILE`

Bagaimana jika ingin menggenerate atau membuat file dengan format CSV? Tidak ada masalah, kita tinggal mengganti nama file saja. File CSV pada dasarnya merupakan file teks biasa dengan extension .csv.

Saya bisa membuat sebuah file CSV dengan nama file `hasil_provinsi_c.csv` menggunakan query berikut ini:

```
SELECT * INTO OUTFILE 'C:\\xampp\\mysql\\file_belajar\\hasil_provinsi_c.csv'
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY "'"
LINES TERMINATED BY '\\r\\n'
FROM provinsi;
```

Query OK, 34 rows affected (0.00 sec)

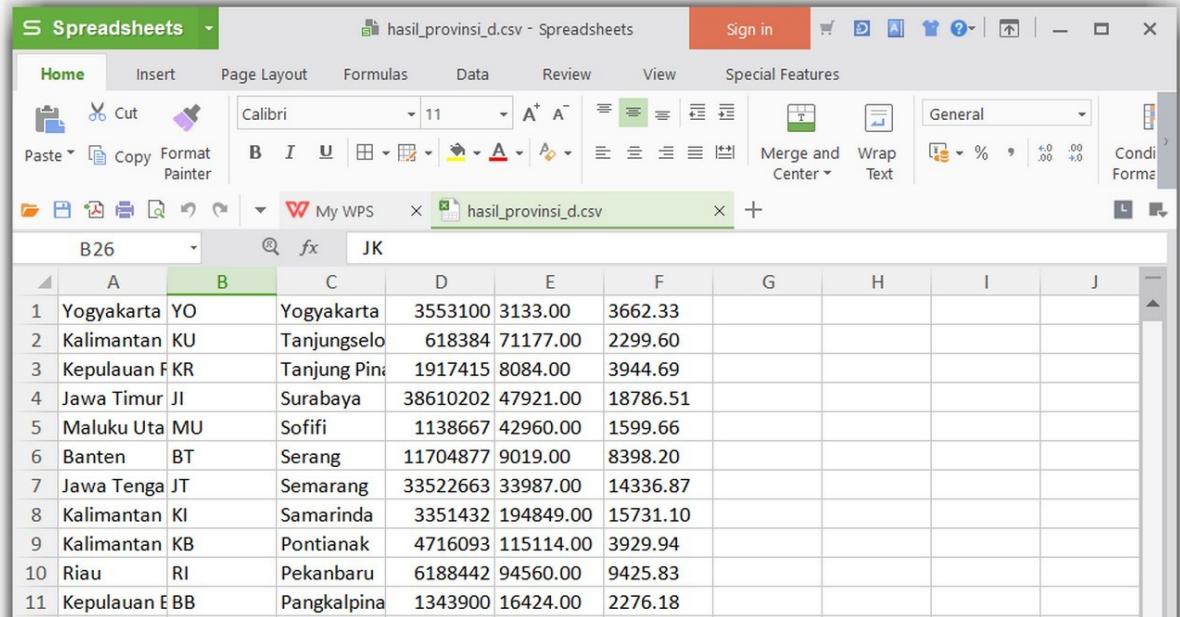
Hasilnya, file `hasil_provinsi_c.csv` berisi file teks dalam format CSV yang bisa dibuka diaplikasi seperti **Microsoft Excel**.

Khusus untuk aplikasi **WPS Spreadsheets** yang saya pakai, ternyata menggunakan tanda titik koma (;) sebagai pemisah kolom untuk file CSV. Agar hasilnya sesuai, saya tinggal mengganti perintah `FIELDS TERMINATED BY ','` menjadi `FIELDS TERMINATED BY ';'.` Berikut querynya:

```
SELECT * INTO OUTFILE 'C:\\xampp\\mysql\\file_belajar\\hasil_provinsi_d.csv'
FIELDS TERMINATED BY ';' OPTIONALLY ENCLOSED BY "'"
LINES TERMINATED BY '\\r\\n'
FROM provinsi;
```

Query OK, 34 rows affected (0.00 sec)

Sekarang, file `hasil_provinsi_d.csv` sudah bisa dibuka langsung menggunakan WPS Spreadsheets. Setiap kolom data otomatis akan menempati 1 sel tabel.



	A	B	C	D	E	F	G	H	I	J
1	Yogyakarta	YO	Yogyakarta	3553100	3133.00	3662.33				
2	Kalimantan	KU	Tanjungselo	618384	71177.00	2299.60				
3	Kepulauan	FR	Tanjung Pinang	1917415	8084.00	3944.69				
4	Jawa Timur	JI	Surabaya	38610202	47921.00	18786.51				
5	Maluku Utara	MU	Sofifi	1138667	42960.00	1599.66				
6	Banten	BT	Serang	11704877	9019.00	8398.20				
7	Jawa Tengah	JT	Semarang	33522663	33987.00	14336.87				
8	Kalimantan	KI	Samarinda	3351432	194849.00	15731.10				
9	Kalimantan	KB	Pontianak	4716093	115114.00	3929.94				
10	Riau	RI	Pekanbaru	6188442	94560.00	9425.83				
11	Kepulauan Riau	BB	Pangkalpinang	1343900	16424.00	2276.18				

Gambar: File CVS hasil export dari MySQL sudah bisa langsung dibuka dari aplikasi WPS Spreadsheets

Jika anda menggunakan Microsoft Excel, tetap gunakan perintah `FIELDS TERMINATED BY ','` karena Excel menggunakan tanda koma sebagai pemisah kolom.

18.4 Export Data dengan Mysqldump

Selain menggunakan query `SELECT... INTO OUTFILE`, kita juga bisa menggunakan `mysqldump` sebagai cara untuk mengexport tabel MySQL.

Bedanya, `mysqldump` akan menghasilkan file yang berisi query MySQL, bukan data “mentah” seperti pada `SELECT... INTO OUTFILE`. File hasil `mysqldump` cocok dipakai sebagai sarana backup tabel dan database. Atau bisa juga jika kita ingin memindahkan tabel / database ke tempat lain.

`Mysqldump` sendiri sebenarnya sebuah file `exe` terpisah yang ada di dalam folder `mysql\bin`. Cara penggunaannya mirip seperti menjalankan MySQL Client, yakni dengan mengakses folder `mysql\bin\mysqldump` dari cmd Windows. Jika saat ini anda berada di dalam cmd MySQL Client, ketik perintah `exit` untuk keluar.

Terdapat beberapa format penulisan dari perintah `mysqldump`. Diantaranya adalah sebagai berikut:

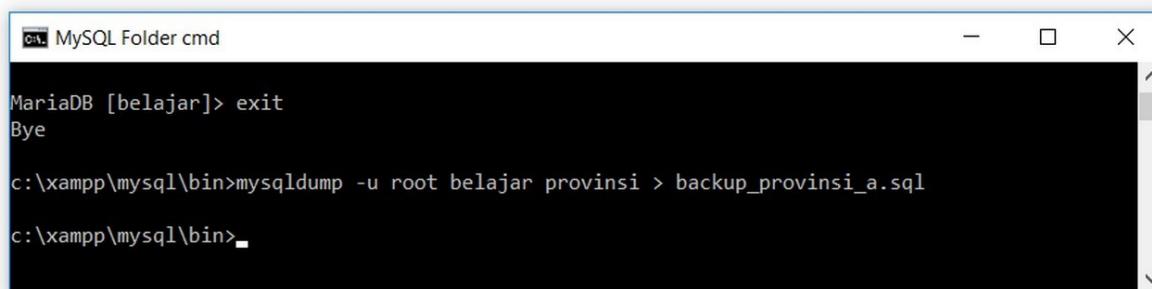
```
shell> mysqldump [options] nama_database [nama_tabel ...] > file_output.sql
shell> mysqldump [options] --databases nama_database ... > file_output.sql
shell> mysqldump [options] --all-databases > nama_file_output.sql
```

Perintah di baris pertama dipakai untuk mengexport satu atau beberapa tabel. Perintah di baris kedua untuk mengexport satu atau beberapa database. Perintah di baris ketiga dipakai untuk mengexport seluruh database yang ada di MySQL Server. Kita akan membahasnya satu persatu.

Mysqldump memiliki banyak sekali pilihan pengaturan (*options*), terdapat lebih dari 100 konfigurasi yang bisa dipilih. Jika anda tertarik bisa membacanya ke dokumentasi resmi MySQL: [mysqldump — A Database Backup Program⁷](https://dev.mysql.com/doc/refman/5.7/en/mysqldump.html). Kita hanya akan membahas penggunaan **mysqldump** secara umum.

Mari mulai praktik. Silahkan buka cmd windows namun kali ini jangan masuk ke MySQL Client, cukup sampai ke folder `mysql\bin`. Setelah itu ketik perintah berikut:

```
mysqldump -u root belajar provinsi > backup_provinsi_a.sql
```



```
MariaDB [belajar]> exit
Bye

c:\xampp\mysql\bin>mysqldump -u root belajar provinsi > backup_provinsi_a.sql
c:\xampp\mysql\bin>
```

Gambar: Menjalankan **mysqldump** untuk membuat file `backup_provinsi_a.sql`

Sama seperti masuk ke MySQL Client (`mysql.exe`), kita harus menuliskan **username** ketika menjalankan **mysqldump**. Dalam hal ini perintah `-u root` dipakai untuk mendapatkan hak akses sebagai user **root**. Jika user ini memiliki password, kita juga harus menulis passwordnya.

Setelah penulisan **username**, berikutnya terdapat perintah `belajar provinsi > backup_provinsi_a.sql`. Artinya, saya ingin mengexport tabel `provinsi` yang ada di dalam database `belajar` ke sebuah file external yang diberi nama `backup_provinsi_a.sql`.

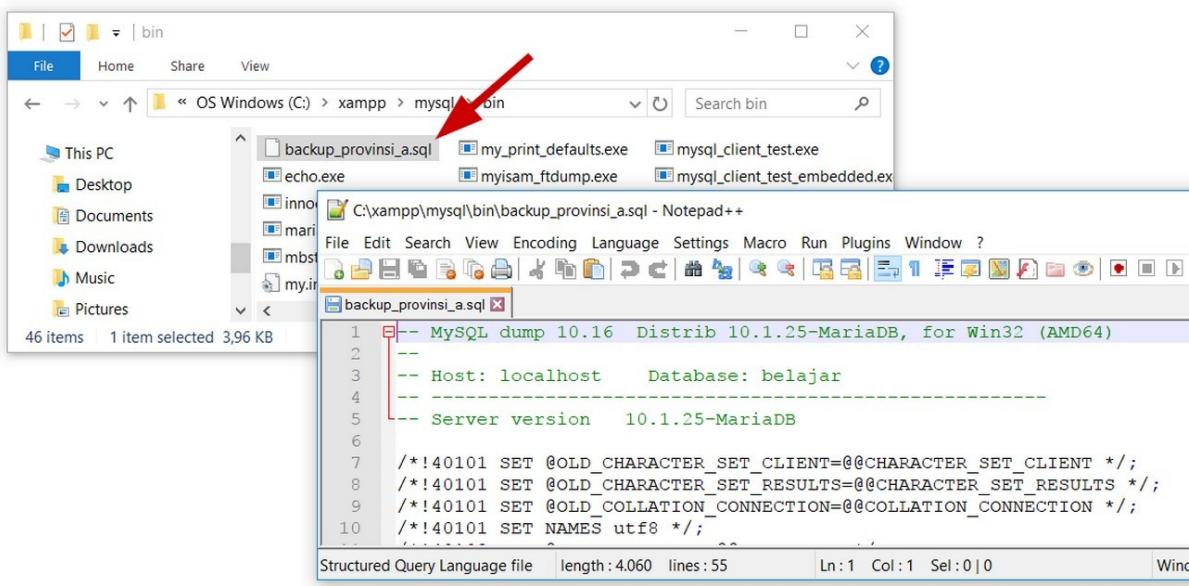
Jika jendela cmd Windows tidak menampilkan pesan error apapun (dan cursor akan pindah ke baris berikutnya), berarti tidak ada masalah. Pertanyaannya, dimanakah lokasi file `backup_provinsi_a.sql` ini berada?

Secara default, file `backup_provinsi_a.sql` akan dibuat di dalam folder yang sama dengan file **mysqldump.exe**, yakni di folder `mysql\bin`. Silahkan buka folder ini dan anda akan mendapatkan sebuah file `backup_provinsi_a.sql`.

Perhatikan extension yang saya pakai untuk file ini, yakni `.sql`. Sama seperti file CSV, file yang dihasilkan dari **mysqldump.exe** hanyalah file teks biasa. Akhiran `.sql` dipakai untuk membedakannya dengan file teks biasa. Dengan akhiran `.sql` kita bisa tau bahwa isi file teks ini adalah perintah SQL.

Karena `backup_provinsi_a.sql` berbentuk file teks biasa, kita bisa melihat isinya menggunakan Notepad++. Berikut hasil yang saya dapatkan:

⁷<https://dev.mysql.com/doc/refman/5.7/en/mysqldump.html>



Gambar: Isi file `backup_provinsi_a.sql` yang didapat dari perintah `mysqldump`

Isi file `backup_provinsi_a.sql` berupa perintah query SQL biasa seperti yang kita pakai untuk membuat tabel. Di dalamnya terdapat query `DROP TABLE IF EXISTS provinsi`, `CREATE TABLE provinsi`, serta `INSERT INTO provinsi`. Semua query ini dipakai untuk menggenerate kembali tabel `provinsi`.

Selain itu juga terdapat beberapa perintah dan keterangan tambahan di dalam file `backup_provinsi_a.sql`. Nantinya, dengan satu perintah sederhana kita bisa membuat ulang tabel `provinsi` menggunakan file `backup_provinsi_a.sql` ini (akan dibahas sesaat lagi).

Apabila diinginkan, kita juga bisa membuat file dengan extension `.txt`. Nantinya file ini juga bisa dipakai untuk menggenerate tabel `provinsi`. Jadi kita tidak harus menggunakan extension `.sql` untuk `mysqldump`. Dalam query berikut ini saya akan membuat file `backup_provinsi_b.txt`:

```
mysqldump -u root belajar provinsi > backup_provinsi_b.txt
```

Jika anda menjalankan perintah diatas, akan tercipta file `backup_provinsi_b.txt` di folder `mysql\bin`. Bagaimana jika kita ingin file hasil `mysqldump` disimpan di folder lain? Misalnya ke folder `file_belajar`?

Caranya, cukup dengan menulis alamat file tersebut. Sebagai contoh, kode program berikut akan membuat file `backup_provinsi_c.sql` di folder `file_belajar`:

```
mysqldump -u root belajar provinsi >
C:\xampp\mysql\file_belajar\backup_provinsi_c.sql
```

Perintah `mysqldump` yang kita jalankan sebelum ini hanya akan mengexport 1 tabel saja. Bagaimana dengan 2 atau 3 tabel? Tinggal menulis nama tabel tersebut setelah nama database, dipisah dengan tanda spasi.

Sebagai contoh, saya ingin membuat backup untuk tabel `provinsi`, `mahasiswa` dan `universitas`. Berikut querynya:

```
mysqldump -u root belajar provinsi mahasiswa universitas >  
C:\xampp\mysql\file_belajar\backup_3_tabel.sql
```

File backup_3_tabel.sql akan berisi query SQL untuk menggenerate 3 tabel, yakni provinsi, mahasiswa, dan universitas yang ada di dalam database belajar.

Untuk mem-backup satu database secara langsung, bisa menggunakan query berikut:

```
mysqldump -u root belajar >  
C:\xampp\mysql\file_belajar\backup_database_belajar.sql
```

Sekarang file backup_database_belajar.sql akan berisi kode query SQL untuk membuat ulang seluruh tabel yang ada di dalam database belajar.

Alternatifnya, bisa menggunakan query berikut:

```
mysqldump -u root --databases belajar >  
C:\xampp\mysql\file_belajar\backup_database_belajar.sql
```

Perintah mysqldump seperti ini cocok digunakan untuk proses backup database secara reguler. Selain itu, MySQL juga menyediakan perintah **mysqldump** untuk membackup seluruh database dan tabel yang ada di MySQL Server saat ini. Perintahnya adalah:

```
mysqldump -u root --all-databases >  
C:\xampp\mysql\file_belajar\backup_all_database.sql
```

Disini file backup_all_database.sql akan berisi kode query SQL untuk membuat ulang seluruh database. File backup ini cocok jika kita ingin pindah atau migrasi ke server lain atau sekedar backup untuk seluruh database.

18.5 Import Data dengan Mysqldump

Setelah mengexport atau membackup data menggunakan **mysqldump**, saatnya kita pelajari cara mengimport data tersebut kembali ke dalam MySQL.

Dalam materi sebelumnya, kita sudah menjalankan beberapa perintah **mysqldump**. Hasilnya, terdapat setidaknya 3 file yang berisi backup data:

- backup_provinsi_c.sql: berisi backup tabel provinsi.
- backup_3_tabel.sql: berisi backup tabel provinsi, mahasiswa dan universitas
- backup_database_belajar.sql: berisi backup database belajar (dan semua tabel di dalamnya).

Ketiga file ini akan kita pakai untuk praktek import data.

Silahkan masuk ke MySQL Client sebagai user **root** dan pilih database **belajar**. Hapus tabel provinsi karena kita akan coba generate ulang dari file external:

```
DROP TABLE provinsi;
Query OK, 0 rows affected (0.22 sec)

SELECT * FROM provinsi;
-- ERROR 1146 (42S02): Table 'belajar.provinsi' doesn't exist
```

Sekarang tabel provinsi sudah tidak ada. Kita akan buat ulang menggunakan file backup_provinsi_c.sql yang berada di folder file_belajar. Perintah querynya adalah sebagai berikut:

```
source C:\xampp\mysql\file_belajar\backup_provinsi_c.sql
```

Gambar: Import tabel provinsi dari file backup_provinsi_c.sql

Setelah di jalankan, akan tampil beberapa kali baris Query OK, 0 rows affected (0.00 sec). Ini adalah hasil *feedback* saat seluruh perintah yang ada di dalam file backup_provinsi_c.sql dijalankan.

Setelah proses selesai, mari periksa kembali tabel provinsi:

```
SELECT * FROM provinsi;
+-----+-----+-----+-----+-----+
| nama_prov | kode_iso | ibukota | populasi | luas | apbd |
+-----+-----+-----+-----+-----+
| Yogyakarta | YO | Yogyakarta | 3553100 | 3133.00 | ... |
| Kalimantan Utara | KU | Tanjungselor | 618384 | 71177.00 | ... |
| ... | .. | ... | ... | ... | ... |
| Aceh | AC | Banda Aceh | 4906835 | 57365.00 | ... |
| Maluku | MA | Ambon | 1657409 | 49350.00 | ... |
+-----+-----+-----+-----+-----+
34 rows in set (0.00 sec)
```

Tabel provinsi sudah hadir kembali, lengkap dengan seluruh isi data.

Percobaan kedua, saya akan membuat ulang 3 tabel: provinsi, mahasiswa dan universitas dari file backup_3_tabel.sql. Cara yang dipakai juga sama, yakni dengan perintah source. Berikut querynya:

```
DROP TABLE provinsi, mahasiswa, universitas;
Query OK, 0 rows affected (0.47 sec)

SELECT * FROM provinsi;
-- ERROR 1146 (42S02): Table 'belajar.provinsi' doesn't exist

SELECT * FROM mahasiswa;
-- ERROR 1146 (42S02): Table 'belajar.mahasiswa' doesn't exist

SELECT * FROM universitas;
--- ERROR 1146 (42S02): Table 'belajar.universitas' doesn't exist

source C:\xampp\mysql\file_belajar\backup_3_tabel.sql
Query OK, 0 rows affected (0.00 sec)
...
Query OK, 0 rows affected (0.00 sec)

SELECT * FROM provinsi;
...
SELECT * FROM mahasiswa;
...
SELECT * FROM universitas;
```

Di baris paling awal, saya menghapus ketiga tabel menggunakan query `DROP TABLE`. Kemudian saya menjalankan query `SELECT` untuk memastikan ketiga tabel tersebut sudah tidak ada.

Setelah itu, perintah `source C:\xampp\mysql\file_belajar\backup_3_tabel.sql` akan menggenerate ulang ketiga tabel. Hasilnya, tabel `provinsi`, `mahasiswa` dan `universitas` kembali hadir.

Percobaan ketiga, saya akan hapus database `belajar` dan nantinya di generate ulang dari file `backup_database_belajar.sql`.

```
DROP DATABASE belajar;
-- ERROR 1010 (HY000): Error dropping database
-- (can't rmdir '..\belajar', errno: 41 "Directory not empty")
```

Jika anda mencoba menjalankan semua query yang ada di dalam bab ini, error diatas juga akan tampil. Apa yang terjadi?

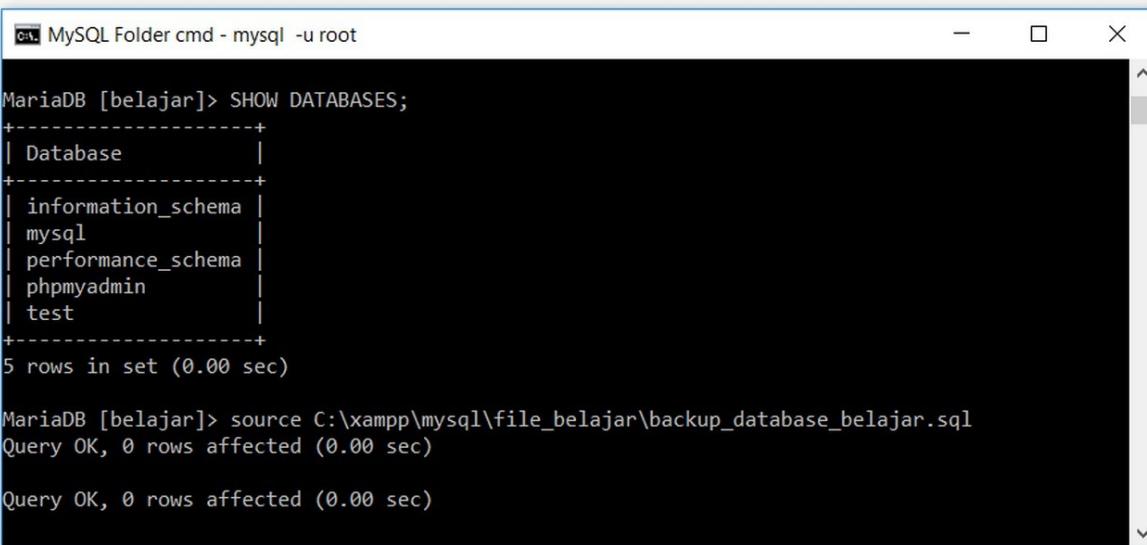
Proses penghapusan database gagal berjalan karena terdapat file yang bukan bagian dari data MySQL di dalam folder `mysql\data\belajar`. Silahkan buka folder ini dan periksa apakah ada file lain di dalamnya.

Pada saat pembahasan materi `LOAD DATA INFILE`, saya menyimpan file `nama_provinsi_a.txt` di dalam folder `mysql\data\belajar`. File `nama_provinsi_a.txt` inilah yang menjadi penyebab query `DROP DATABASE` menghasilkan error.

Hapus manual file `nama_provinsi_a.txt` dari Windows Explorer, lalu jalankan kembali query `DROP DATABASE belajar`. Untuk memastikan database `belajar` sudah terhapus, jalankan query `SHOW DATABASES`.

Sekarang, kita akan generate ulang database `belajar` dari file `backup_database_belajar.sql` dengan perintah berikut:

```
source C:\xampp\mysql\file_belajar\backup_database_belajar.sql
```



The screenshot shows a terminal window titled "MySQL Folder cmd - mysql -u root". It displays the following SQL session:

```
MariaDB [belajar]> SHOW DATABASES;
+-----+
| Database      |
+-----+
| information_schema |
| mysql          |
| performance_schema |
| phpmyadmin     |
| test           |
+-----+
5 rows in set (0.00 sec)

MariaDB [belajar]> source C:\xampp\mysql\file_belajar\backup_database_belajar.sql
Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.00 sec)
```

Gambar: Menggenerate ulang database belajar

Setelah menjalankan perintah diatas, jalankan kembali query `SHOW DATABASES`. Database `belajar` akan kembali tampil. Silahkan periksa apakah di dalam database ini juga terdapat seluruh tabel yang menjadi anggota dari database `belajar`.

Alternatif lain untuk mengimport file SQL adalah menulisnya diluar MySQL client. Untuk bahan praktek, saya akan hapus kembali database `belajar`, lalu keluar dari MySQL client:

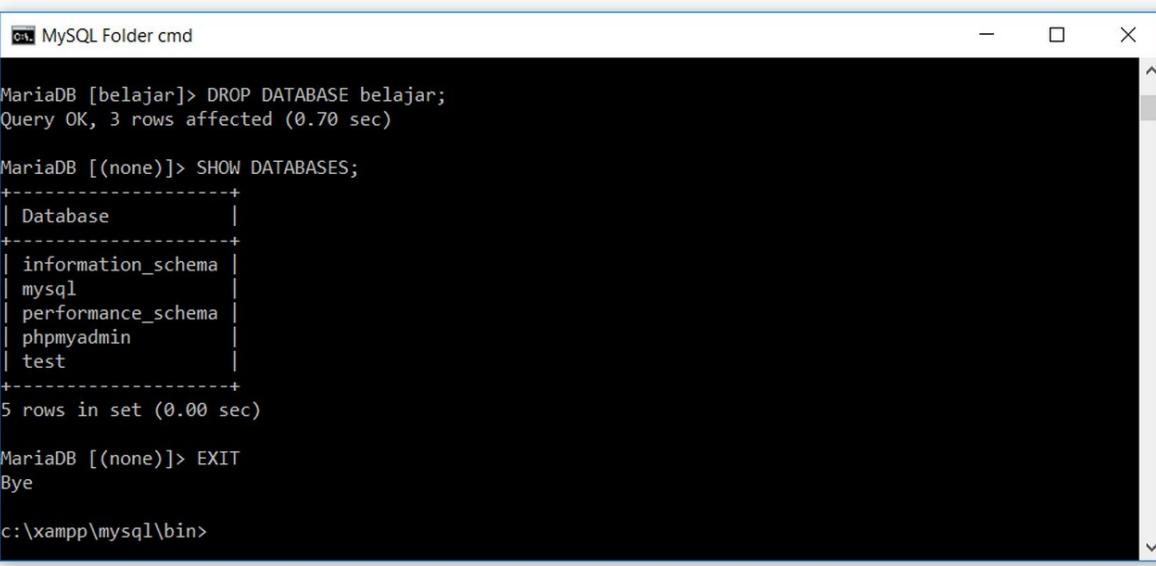
```
DROP DATABASE belajar;
Query OK, 3 rows affected (0.70 sec)
```

```
SHOW DATABASES;
+-----+
| Database      |
+-----+
| information_schema |
| mysql          |
| performance_schema |
| phpmyadmin     |
| test           |
+-----+
```

```
5 rows in set (0.00 sec)
```

EXIT

-- Bye



```
MariaDB [belajar]> DROP DATABASE belajar;
Query OK, 3 rows affected (0.70 sec)

MariaDB [(none)]> SHOW DATABASES;
+-----+
| Database      |
+-----+
| information_schema |
| mysql          |
| performance_schema |
| phpmyadmin     |
| test           |
+-----+
5 rows in set (0.00 sec)

MariaDB [(none)]> EXIT
Bye

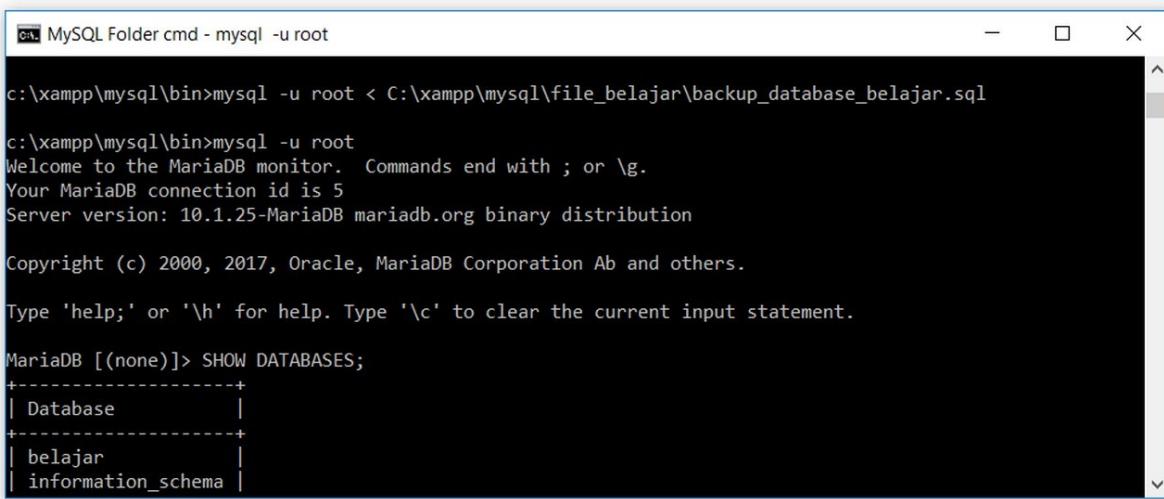
c:\xampp\mysql\bin>
```

Gambar: Hapus database belajar lalu keluar dari MySQL client

Sekarang kita sudah berada di luar MySQL client dan berada di folder `mysql\bin` di dalam cmd Windows.

Untuk mengimport file `backup_database_belajar.sql`, jalankan perintah berikut:

```
mysql -u root < C:\xampp\mysql\file_belajar\backup_database_belajar.sql
```



```
c:\xampp\mysql\bin>mysql -u root < C:\xampp\mysql\file_belajar\backup_database_belajar.sql

c:\xampp\mysql\bin>mysql -u root
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 5
Server version: 10.1.25-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> SHOW DATABASES;
+-----+
| Database      |
+-----+
| belajar       |
| information_schema |
+-----+
```

Gambar: Import file SQL dari luar MySQL Client

Setelah itu silahkan masuk kembali ke dalam MySQL Client dan periksa apakah database `belajar` sudah terbentuk kembali atau belum.

Dalam bab ini kita telah membahas cara import data ke dalam MySQL serta cara export data dari MySQL ke dalam file external. Membuat file SQL menggunakan **mysqldump** sering dipakai sebagai sarana backup atau untuk memindahkan tabel ke server lain.

Jika anda ingin bertanya di group atau forum diskusi, sertakan file hasil **mysqldump** agar rekan yang lain bisa membuat ulang tabel dengan mudah.

19. Pembuatan User dan Hak Akses (Privilege)

Sepanjang pembahasan dari awal buku, saya hanya menggunakan 1 user MySQL, yakni **root**.

User **root** dikenal juga sebagai **super user**, yaitu user yang memiliki hak akses paling tinggi. Seseorang yang login sebagai *superuser* bisa melakukan semua hal, termasuk menghapus seluruh database yang ada serta mematikan MySQL server.

Dalam operasional aplikasi, sebaiknya user **root** ini tidak digunakan. Kita bisa membuat user lain yang secara khusus hanya bisa mengakses database dan tabel yang diperlukan saja. MySQL menyediakan berbagai query yang untuk manajemen user ini, diantaranya `CREATE USER`, `DROP USER`, `RENAME USER`, dll.

Selain itu setiap user juga bisa dibatasi berdasarkan query yang dijalankan, misalnya hanya bisa mengakses data tanpa bisa mengubahnya. Untuk keperluan ini tersedia query `GRANT` dan `REVOKE`.

Dalam bab ini kita akan bahas cara pembuatan user dan hak akses (*privilege*) di dalam MySQL.

19.1 Melihat Daftar User MySQL

Secara default, MariaDB dan MySQL memiliki **root** sebagai user bawaan. Selain itu juga tersedia user lain, terutama di dalam MariaDB bawaan XAMPP yang saya gunakan.

Untuk melihat apa saja user yang saat ini tersedia di dalam MySQL, kita bisa mengakses tabel `user` di database `mysql`. Database `mysql` berisi cukup banyak tabel yang menyimpan berbagai pengaturan MySQL server.

Jika anda menjalankan query `SHOW DATABASES`, database `mysql` ini bisa terlihat:

```
SHOW DATABASES;
+-----+
| Database |
+-----+
| belajar  |
| information_schema |
| mysql    |
| performance_schema |
| phpmyadmin |
| test     |
+-----+
```

Mari kita pilih database `mysql` kemudian lihat apa saja tabel yang ada di dalamnya:

```
USE mysql;
Database changed

SHOW TABLES;
+-----+
| Tables_in_mysql |
+-----+
| column_stats    |
| columns_priv    |
| db               |
| ...              |
| time_zone_transition_type |
| user             |
+-----+
34 rows in set (0.00 sec)
```

Terdapat sekitar 34 tabel di dalam database `mysql`. Tabel -tabel ini menyimpan berbagai settingan terkait MySQL. Sebaiknya tabel-tabel ini tidak diutak-atik karena bisa berdampak ke pengaturan MySQL secara keseluruhan.

Daftar user yang ada di dalam MySQL disimpan di dalam tabel `user`. Mari kita lihat isinya:

```
SELECT * FROM user \G
***** 1. row *****
      Host: localhost
      User: root
      Password:
      Select_priv: Y
      Insert_priv: Y
      Update_priv: Y
      Delete_priv: Y
      Create_priv: Y
      Drop_priv: Y
      Reload_priv: Y
      Shutdown_priv: Y
      Process_priv: Y
      File_priv: Y
      ....: ...
```

Tabel `user` memiliki 46 kolom! Karena jumlah kolom ini sangat banyak, saya menggunakan akhiran `\G` agar hasilnya tampil memanjang ke bawah. Semua kolom yang ada di tabel `user` berkaitan dengan hak akses dan pembatasan user.

Yang menjadi fokus utama kita saat ini ada di 3 kolom saja, yakni kolom `User`, `Password` dan `Host` dari tabel `user`. Ketiga kolom ini berisi data tentang **nama user** yang tersedia di dalam sistem, **password** dari user tersebut, serta alamat **host**.

Berikut perintah yang bisa dipakai untuk melihat 3 kolom ini:

```
SELECT User, Password, Host FROM user;
+-----+-----+
| User | Password | Host      |
+-----+-----+
| root |          | localhost |
| root |          | 127.0.0.1 |
| root |          | ::1       |
|      |          | localhost |
| pma  |          | localhost |
+-----+
```



Jika anda tidak memilih database `mysql` dengan perintah `USE DATABASE mysql`, query diatas juga bisa dijalankan dengan perintah `SELECT User, Password, Host FROM mysql.user`.

Inilah daftar **seluruh user** yang ada di MariaDB bawaan XAMPP. Jika anda mengakses MySQL atau MariaDB yang diinstall secara *standalone* (bukan bersama XAMPP), hasilnya akan berbeda. Tapi yang pasti user **root** selalu ada.

Di dalam MySQL, lokasi tempat user mengakses ke server merupakan bagian dari user itu sendiri. Inilah yang menjadi alasan kenapa terdapat 3 nama user **root** dalam tabel `user`. Namanya sama-sama **root** tapi terdapat perbedaan di isian kolom `host`, yakni **localhost**, **127.0.0.1**, dan **::1**.

Kolom **Host** berisi alamat komputer dari mana seorang user diperbolehkan login ke server. MySQL bisa membatasi seorang user hanya bisa login dari komputer tertentu saja.

Alamat **localhost** adalah sebutan untuk *komputer saat ini*. Alamat host **127.0.0.1**, dan **::1** juga merujuk ke komputer localhost. **127.0.0.1** merupakan alamat IP versi 4, sedangkan **::1** adalah alamat IP versi 6.

Karena user **root** memiliki nilai host **localhost**, **127.0.0.1**, dan **::1**, artinya user **root** hanya bisa login ke MySQL server dari komputer localhost saja. Seandainya saya mengkoneksikan 2 buah komputer menggunakan jaringan LAN, user **root** tidak bisa login dari komputer lain.

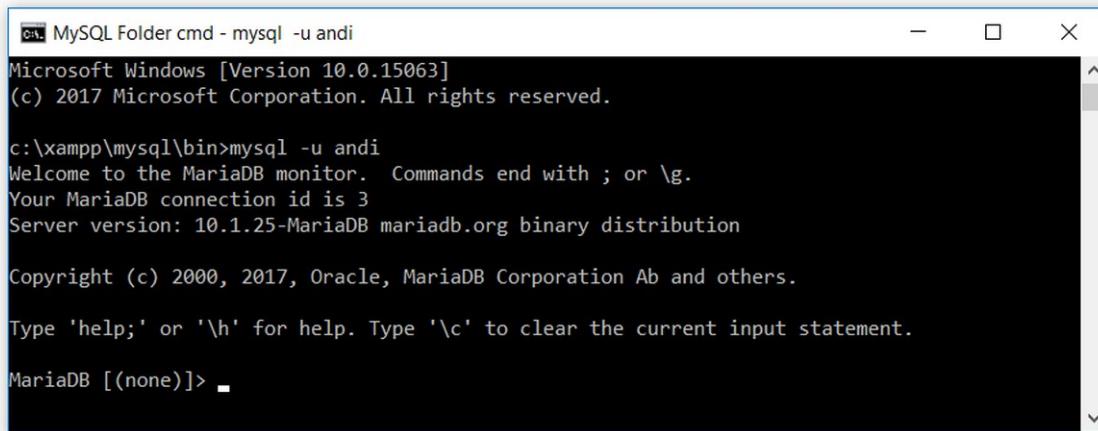
Ke-3 baris user **root** ini memiliki nilai kosong di kolom `Password`. Artinya, user **root** tidak memiliki password apapun. Ini merupakan nilai default bawaan XAMPP.

Di baris ke-4 nilai kolom `user` kosong, `password` kosong, namun kolom `Host` berisi alamat `localhost`, apa maksudnya?

Baris inilah penyebab sebagian besar kasus salah login ke dalam MySQL. Kolom `user` dan `password` tidak berisi apa-apa, efeknya MySQL membolehkan user apa pun untuk login ke dalam MySQL Server **tanpa password** (dikenal juga sebagai user *anonym*). Dengan syarat, user tersebut hanya bisa login di komputer localhost.

Sebagai percobaan, silahkan buka cmd Windows kedua, atau exit dari user **root** saat ini. Kemudian test masuk sebagai user **andi**:

```
c:\xampp\mysql\bin>mysql -u andi
```



```
MySQL Folder cmd - mysql -u andi
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

c:\xampp\mysql\bin>mysql -u andi
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 3
Server version: 10.1.25-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

Gambar: Login ke MySQL sebagai andi

Kita bisa masuk! Bahkan, sembarang user pun juga tetap bisa masuk. Anda bisa mencoba memakai nama user lain seperti rani, joko, atau aaa, dan semuanya bisa login. Inilah maksud dari baris kosong di tabel user, yakni berupa sebuah user *anonym*.

Akan tetapi, user anonym ini memiliki hak akses yang dibatasi. Kita bisa cek dengan cara menjalankan query SHOW DATABASES:

```
SHOW DATABASES;
+-----+
| Database      |
+-----+
| information_schema |
| test          |
+-----+
2 rows in set (0.00 sec)
```

Hanya tampil dua buah database. Database belajar tidak terlihat karena hak akses dari user anonym memang dibatasi.

MySQL juga memiliki sebuah fungsi khusus untuk menampilkan nama user saat ini, yakni SELECT USER():

```
SELECT USER();
+-----+
| USER()      |
+-----+
| andi@localhost |
+-----+
1 row in set (0.00 sec)
```

Terlihat bahwa user yang sedang dipakai adalah andi yang berada di localhost.

User anonym seperti andi ini bisa menjadi masalah. Contoh kasusnya sebagai berikut:

```
mysql -u root;
Welcome to the MariaDB monitor. Commands end with ; or \g.
```

```
SHOW DATABASES;
+-----+
| Database      |
+-----+
| information_schema |
| test          |
+-----+
2 rows in set (0.00 sec)
```

```
c:\xampp\mysql\bin>mysql -u root;
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 4
Server version: 10.1.25-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> SHOW DATABASES;
+-----+
| Database      |
+-----+
| information_schema |
| test          |
+-----+
2 rows in set (0.00 sec)
```

Gambar: Login sebagai user root ?

Bisakah anda menjelaskan kenapa database yang tampil hanya 2? Padahal saya login sebagai **root**. Kenapa tidak tampak database belajar? Saya bisa jamin bahwa database tersebut ada.

Kasus seperti ini sering di tanyakan di kolom komentar duniaIlkom maupun forum-forum terkait MySQL. Jawabannya ada di tambahan tanda titik koma setelah root. User yang login diatas bukanlah root, tetapi root; .

Karena MariaDB bawaan XAMPP memiliki user `anonym`, user `root`; tetapi bisa masuk dengan hak akses yang dibatasi (sama seperti user `andi` sebelumnya). Jika kita menjalankan query `SELECT USER()`, hal ini bisa langsung terkonfirmasi:

```
SELECT USER();
+-----+
| USER()      |
+-----+
| root;@localhost |
+-----+
1 row in set (0.00 sec)
```

Dari hasil diatas terlihat user yang sedang dipakai saat adalah `root`; yang berada di komputer `localhost`. Jika kita ingin masuk sebagai `root`, seharusnya yang tampil adalah `root@localhost` (tanpa tambahan titik koma).

Kembali ke tabel `user`, di baris terakhir terdapat user `pma`. User `pma` ini merupakan user khusus yang digunakan oleh aplikasi `phpMyAdmin` bawaan XAMPP.

19.2 Cara Login ke MySQL Server

Cara login ke MySQL server yang kita pakai selama ini merupakan penulisan singkat. MySQL Server sebenarnya perlu berbagai informasi lain yang jika tidak ditulis akan menggunakan nilai default dari MySQL.

Ketika sebuah user login ke MySQL server, setidaknya butuh 4 data:

- Nama user.
- Password user, jika tidak ditulis akan dianggap tanpa password (kosong).
- Alamat host MySQL Server (alamat IP komputer tempat MySQL Server berada). Jika host tidak ditulis, dianggap sebagai `localhost`.
- Nomor port jaringan, jika tidak ditulis akan menggunakan port default MySQL: `3306`.

Untuk menginput ke-4 data ini, kita menggunakan cara login khusus dengan format sebagai berikut:

```
mysql -u nama_user -ppassword -h alamat_host -P nomor_port
```

Atau bisa juga menggunakan format yang panjang:

```
mysql --user=nama_user --password=password --host=alamat_host
--port=nomor_port
```

Sebagai contoh, sepanjang buku ini saya login dengan format perintah:

```
c:\xampp\mysql\bin>mysql -u root
```

Ini sama artinya dengan:

```
c:\xampp\mysql\bin>mysql -u root -p
```

```
c:\xampp\mysql\bin>mysql -u root -h localhost
```

```
c:\xampp\mysql\bin>mysql -u root -h localhost -P 3306
```

Atau dengan penulisan panjang perintahnya seperti ini:

```
c:\xampp\mysql\bin>mysql --user=root
```

```
c:\xampp\mysql\bin>mysql --user=root --host=localhost
```

```
c:\xampp\mysql\bin>mysql --user=root --host=localhost --port=3306
```

Khusus untuk pengisian password belum bisa dipakai karena user root belum memiliki password. Kita akan lihat prakteknya sebentar lagi.

Selain ke-4 data ini (nama user, password, host dan port), sebenarnya masih ada informasi login lain yang relatif jarang digunakan. Lengkapnya bisa lihat ke dokumentasi resmi MySQL: [Connecting to the MySQL Server¹](#)

19.3 Membuat User MySQL

Untuk membuat user MySQL, tersedia query `CREATE USER`. Query ini memiliki berbagai pengaturan dan perintah tambahan yang akan kita bahas secara bertahap.

Sebagai contoh pertama, saya ingin membuat user MySQL baru dengan nama budi, berikut perintahnya:

```
CREATE USER 'budi';
-- Query OK, 0 rows affected (0.00 sec)
```

Untuk memastikan user budi sudah terbentuk, bisa dengan cara memeriksa isi tabel `user`:

¹<https://dev.mysql.com/doc/refman/5.5/en/connecting.html>

```
SELECT User, Host, Password FROM user;

+-----+-----+-----+
| User | Host      | Password |
+-----+-----+-----+
| root | localhost |          |
| root | 127.0.0.1 |          |
| root | ::1       |          |
|      | localhost |          |
| pma  | localhost |          |
| budi | %        |          |
+-----+-----+-----+
6 rows in set (0.00 sec)
```

Di baris terakhir terdapat nilai budi di kolom User. Artinya, user budi sudah terbentuk.

Perhatikan pula isi kolom Host untuk budi, nilainya berupa tanda persen (%). Tanda ini adalah karakter ‘wildcard’ yang berarti nilai host tidak ditentukan. User budi yang baru saja kita buat bisa login ke dalam MySQL dari komputer mana saja.

Untuk menghapus sebuah user, tersedia query `DROP USER`. Sebagai contoh saya ingin menghapus user budi yang baru saja kita buat:

```
DROP USER 'budi';
-- Query OK, 0 rows affected (0.04 sec)
```

```
SELECT User, Host, Password FROM user;

+-----+-----+-----+
| User | Host      | Password |
+-----+-----+-----+
| root | localhost |          |
| root | 127.0.0.1 |          |
| root | ::1       |          |
|      | localhost |          |
| pma  | localhost |          |
+-----+-----+-----+
```

Sekarang user budi sudah berhasil dihapus.

Cara lain untuk menghapus user adalah dengan query `DELETE`, yakni menghapus manual baris yang ada di dalam tabel user. Berikut contoh prakteknya:

```
CREATE USER 'budi';
-- Query OK, 0 rows affected (0.00 sec)
```

```
SELECT User, Host, Password FROM user;
+-----+-----+
| User | Host      | Password |
+-----+-----+
| root | localhost |          |
| root | 127.0.0.1 |          |
| root | ::1       |          |
|      | localhost |          |
| pma  | localhost |          |
| budi | %        |          |
+-----+-----+
6 rows in set (0.00 sec)
```

```
DELETE FROM user WHERE User='budi';
-- Query OK, 1 row affected (0.06 sec)
```

```
FLUSH PRIVILEGES;
-- Query OK, 0 rows affected (0.00 sec)
```

```
SELECT User, Host, Password FROM user;
+-----+-----+
| User | Host      | Password |
+-----+-----+
| root | localhost |          |
| root | 127.0.0.1 |          |
| root | ::1       |          |
|      | localhost |          |
| pma  | localhost |          |
+-----+-----+
5 rows in set (0.00 sec)
```

Di awal saya membuat ulang user budi dengan query `CREATE USER 'budi'`. Kemudian menampilkan isi tabel `user` sekedar memastikan user tersebut sudah ada.

Untuk menghapus user budi, kali ini saya menggunakan query `DELETE FROM user WHERE User='budi'`. Perintah ini akan menghapus 1 baris di tabel `user` dimana kolom `User` bernilai budi.

Di baris berikutnya terdapat query `FLUSH PRIVILEGES`. Ini adalah query khusus yang berfungsi untuk me-reload ulang tabel `user` (serta beberapa tabel lain). Fungsinya agar perubahan yang dibuat bisa langsung diterapkan oleh MySQL.

Query `FLUSH PRIVILEGES` ini hanya perlu jika kita mengedit manual tabel `user` menggunakan query `INSERT`, `UPDATE`, dan `DELETE`.

Apabila query FLUSH PRIVILEGES tidak dijalankan, perubahan di dalam tabel user tidak berefek sampai MySQL server di restart ulang. User budi yang baru saja kita hapus masih bisa login di tempat lain karena MySQL “belum tahu” bahwa tabel user sudah diubah secara manual.

Jika perubahan user dan hak akses dilakukan dengan query khusus pembuatan user seperti CREATE USER atau DROP USER, query FLUSH PRIVILEGES tidak perlu di jalankan.

Dalam kebanyakan situasi, lebih disarankan menghapus user menggunakan query DROP USER dibandingkan menghapus tabel user secara manual.

Akan tetapi terdapat beberapa kasus dimana kita hanya bisa menggunakan query DELETE untuk menghapus user, salah satunya untuk menghapus user *anonym* yang ada di dalam MariaDB bawaan XAMPP.

Seperti yang terlihat di tabel user, user *anonym* ini tidak memiliki nama (kolom nama berisi string kosong), karena itu kita tidak bisa menggunakan query DROP USER.

Untuk menghapus user *anonym* ini, bisa menggunakan query berikut:

```
DELETE FROM user WHERE User='';
-- Query OK, 1 row affected (0.00 sec)

FLUSH PRIVILEGES;
-- Query OK, 0 rows affected (0.00 sec)

SELECT User, Host, Password FROM user;
+-----+-----+-----+
| User | Host      | Password |
+-----+-----+-----+
| root | localhost |          |
| root | 127.0.0.1 |          |
| root | ::1       |          |
| pma  | localhost |          |
+-----+-----+-----+
4 rows in set (0.00 sec)
```

Sekarang user *anonym* sudah terhapus dan yang tersedia hanyalah user **root** serta user **pma** yang digunakan khusus oleh phpMyAdmin.

Untuk mengubah nama sebuah user, tersedia query RENAME USER. Sebagai contoh saya ingin menukar nama user budi menjadi andi:

```

CREATE USER 'budi';
-- Query OK, 0 rows affected (0.00 sec)

RENAME USER 'budi' TO 'andi';
-- Query OK, 0 rows affected (0.04 sec)

SELECT User, Host, Password FROM user;
+-----+-----+
| User | Host      | Password |
+-----+-----+
| root | localhost |          |
| root | 127.0.0.1 |          |
| root | ::1       |          |
| pma  | localhost |          |
| andi | %        |          |
+-----+-----+

```

Sekarang user budi sudah berubah menjadi andi.

Sebelum masuk ke pembahasan berikutnya, saya akan menghapus user andi ini:

```

DROP USER 'andi';
-- Query OK, 0 rows affected (0.00 sec)

```

19.4 Membuat User dengan Batasan Host

User budi yang kita buat sebelumnya tidak memiliki batasan host. Maksudnya, user budi bisa login ke MySQL server dari komputer mana saja selama terhubung ke server (misalnya terhubung ke jaringan LAN atau WIFI yang sama).

Dari segi keamanan, ini tidak bagus. Karena siapapun bisa membawa laptop sendiri kemudian mencari colokan kabel LAN dan ia bisa login sebagai user budi. Kita bisa meningkatkan keamanan dengan cara membatasi komputer mana saja yang boleh dipakai oleh user budi.

Untuk membuat batasan host, query CREATE USER bisa ditulis sebagai berikut:

```
CREATE USER 'namauser'@'lokasihost'
```

Misalkan saya ingin user budi hanya bisa login dari komputer dengan IP Address 192.168.0.2, perintahnya adalah sebagai berikut:

```
CREATE USER 'budi'@'192.168.0.2';
-- Query OK, 0 rows affected (0.00 sec)

SELECT User, Host, Password FROM user;
+-----+-----+
| User | Host      | Password |
+-----+-----+
| root | localhost |          |
| root | 127.0.0.1 |          |
| root | ::1       |          |
| budi | 192.168.0.2 |          |
| pma  | localhost |          |
+-----+-----+
```

Sekarang, user budi hanya bisa login dari komputer dengan alamat IP 192.168.0.2. User budi tidak bisa login dari komputer lain, termasuk dari localhost (komputer yang kita gunakan saat ini).

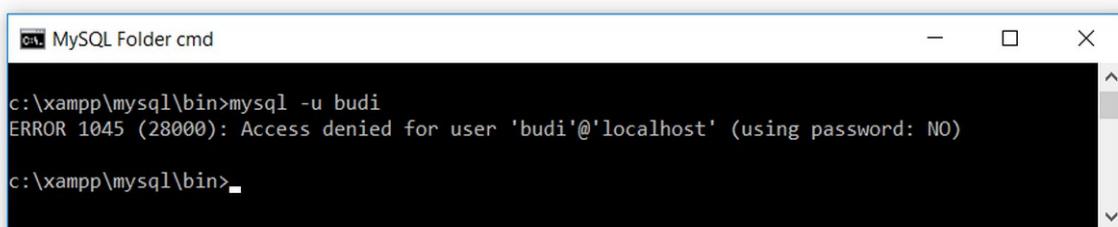


Tanda kutip dalam pembuatan nama user dan host sebenarnya opsional, kita bisa saja menulis CREATE USER budi@192.168.0.2.

Untuk menguji user budi, silahkan buka cmd Windows kedua dan coba login sebagai user budi:

```
c:\xampp\mysql\bin>mysql -u budi
```

```
-- ERROR 1045 (28000): Access denied for user 'budi'@'localhost'
-- (using password: NO)
```



Gambar: Gagal login sebagai user budi

User budi tidak bisa login karena kita mengaksesnya dari komputer localhost. User budi yang baru saja dibuat hanya bisa diakses dari komputer dengan IP 192.168.0.2, sesuai dengan query pembuatan user tersebut.



Jika ternyata anda bisa masuk sebagai user budi, bisa jadi ini karena user *anonym* bawaan MySQL/MariaDB dari XAMPP belum dihapus.

Jadi, bagaimana caranya agar user budi bisa diakses dari komputer localhost? Kita bisa tambahkan hak akses host untuk user budi dengan query berikut ini:

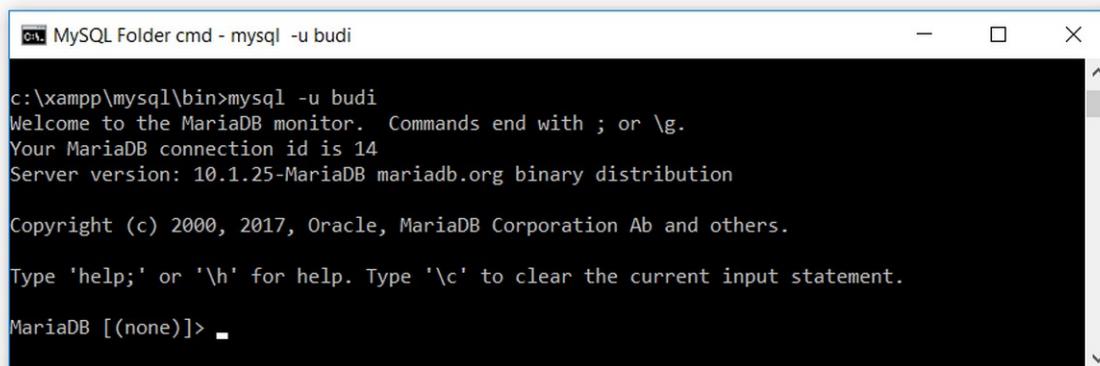
```
CREATE USER 'budi'@'localhost';
-- Query OK, 0 rows affected (0.00 sec)

SELECT User, Host, Password FROM user;
+-----+-----+
| User | Host      | Password |
+-----+-----+
| root | localhost |          |
| root | 127.0.0.1 |          |
| root | ::1       |          |
| budi | 192.168.0.2 |          |
| pma  | localhost |          |
| budi | localhost |          |
+-----+-----+
6 rows in set (0.00 sec)
```

Saya membuat lagi user budi, namun kali ini alamat host-nya berisi **localhost**.

Perhatikan isi tabel user, user budi terdaftar 2 kali dengan alamat host yang berbeda. Artinya, user budi bisa masuk ke MySQL server dari komputer dengan IP 192.168.0.2 atau dari komputer localhost. Dalam prakteknya nanti, kedua user budi ini bisa memiliki hak akses yang berbeda.

Untuk uji coba, silahkan buka jendela cmd Windows kedua lalu login lagi sebagai budi:



```
c:\xampp\mysql\bin>mysql -u budi
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 14
Server version: 10.1.25-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

Gambar: Sukses login sebagai user budi



User budi yang baru saja dibuat memiliki hak akses yang sangat terbatas, karena saya memang belum memberikan hak akses apapun. Jika anda menjalankan query SHOW DATABASES, database belajar juga tidak akan terlihat (hak aksesnya sama seperti user *anonym*).

Jika kita ingin user budi bisa diakses dari komputer dengan alamat host lain, tinggal menjalankan query CREATE USER dan tulis nama host-nya.

Untuk menghapus user budi, kita tidak bisa hanya menjalankan query DROP USER 'budi' karena akan menghasilkan error:

```
DROP USER 'budi';
-- ERROR 1396 (HY000): Operation DROP USER failed for 'budi'@'%'
```

Hal ini terjadi karena user budi sekarang memiliki info tambahan, yakni **host**. Query `DROP USER` harus lebih spesifik dengan menuliskan nama hostnya. MySQL perlu informasi tambahan terkait user budi yang mana yang ingin dihapus, apakah yang ada di `localhost` atau di `192.168.0.2`.

Untuk menghapus user budi di `192.168.0.2`, perintahnya adalah sebagai berikut:

```
DROP USER 'budi'@'192.168.0.2';
-- Query OK, 0 rows affected (0.00 sec)
```

```
SELECT User, Host, Password FROM user;
+-----+-----+-----+
| User | Host      | Password |
+-----+-----+-----+
| root | localhost |          |
| root | 127.0.0.1 |          |
| root | ::1       |          |
| pma  | localhost |          |
| budi | localhost |          |
+-----+-----+-----+
5 rows in set (0.00 sec)
```

Terlihat bahwa user budi yang tinggal hanya yang terdaftar di `localhost`. Untuk menghapus user budi di `localhost` caranya juga sama:

```
DROP USER 'budi'@'localhost';
-- Query OK, 0 rows affected (0.00 sec)
```

```
SELECT User, Host, Password FROM user;
+-----+-----+-----+
| User | Host      | Password |
+-----+-----+-----+
| root | localhost |          |
| root | 127.0.0.1 |          |
| root | ::1       |          |
| pma  | localhost |          |
+-----+-----+-----+
4 rows in set (0.00 sec)
```

Karakter wildcard persen (%) bisa digunakan sebagai lokasi host, seperti contoh berikut:

```
CREATE USER 'budi'@'%';
Query OK, 0 rows affected (0.00 sec)

SELECT User, Host, Password FROM user;
+-----+-----+
| User | Host      | Password |
+-----+-----+
| root | localhost |          |
| root | 127.0.0.1 |          |
| root | ::1       |          |
| pma  | localhost |          |
| budi | %        |          |
+-----+
5 rows in set (0.00 sec)
```



Jika alamat host mengandung karakter persen %, alamat tersebut harus ditulis di dalam tanda kutip.

Tanda persen (%) di bagian host merupakan nilai default dari query CREATE USER sehingga dua query berikut bermakna sama:

```
CREATE USER 'budi';

CREATE USER 'budi'@'%';
```

Untuk menghapus user budi dengan host %, bisa dengan query berikut:

```
DROP USER 'budi'@'%';
-- Query OK, 0 rows affected (0.00 sec)

SELECT User, Host, Password FROM user;
+-----+-----+
| User | Host      | Password |
+-----+-----+
| root | localhost |          |
| root | 127.0.0.1 |          |
| root | ::1       |          |
| pma  | localhost |          |
+-----+
4 rows in set (0.00 sec)
```

Untuk pembatasan hak akses yang lebih kompleks, kita bisa menulis karakter wildcard % sebagai bagian dari alamat IP Address, seperti contoh berikut:

```
CREATE USER 'budi'@'192.168.0.%';
-- Query OK, 0 rows affected (0.00 sec)

SELECT User, Host, Password FROM user;
+-----+-----+
| User | Host      | Password |
+-----+-----+
| root | localhost |          |
| root | 127.0.0.1 |          |
| root | ::1       |          |
| pma  | localhost |          |
| budi | 192.168.0.% |          |
+-----+
5 rows in set (0.00 sec)
```

Sekarang user budi bisa diakses dari seluruh komputer dalam satu *netmask*, yakni dari komputer dengan IP 192.168.0.1 hingga 192.168.0.255. Kembali, tanda kutip harus ditambahkan saat penulisan host karena mengandung karakter wildcard %.

Bagaimana jika kita ingin mengubah Host sebuah user? Bisa dengan manual update tabel user menggunakan query UPDATE, kemudian jalankan perintah FLUSH PRIVILEGES. Berikut contohnya:

```
UPDATE user SET Host = '172.16.0.5' WHERE User = 'budi';
-- Query OK, 1 row affected (0.18 sec)
-- Rows matched: 1  Changed: 1  Warnings: 0
```

```
FLUSH PRIVILEGES;
-- Query OK, 0 rows affected (0.00 sec)
```

```
SELECT User, Host, Password FROM user;
+-----+-----+
| User | Host      | Password |
+-----+-----+
| root | localhost |          |
| root | 127.0.0.1 |          |
| root | ::1       |          |
| pma  | localhost |          |
| budi | 172.16.0.5 |          |
+-----+
```

Sekarang alamat host untuk user budi sudah berubah menjadi 172.16.0.5.

Sebelum lanjut ke materi berikutnya, saya akan hapus user budi ini:

```
DROP USER 'budi'@'172.16.0.5';
-- Query OK, 0 rows affected (0.00 sec)
```

19.5 Membuat User dengan Password

Untuk membuat user dengan password, kita bisa menambahkan perintah IDENTIFIED BY ke dalam query CREATE USER. Berikut format dasar penulisannya:

```
CREATE USER 'namauuser'@'lokasihost' IDENTIFIED BY 'password'
```

Sebagai contoh, saya ingin membuat user **budi** dengan password **rahasia**:

```
CREATE USER 'budi'@'localhost' IDENTIFIED BY 'rahasia';
-- Query OK, 0 rows affected (0.03 sec)
```

```
SELECT User, Host, Password FROM user;
+-----+-----+
| User | Host      | Password          |
+-----+-----+
| root | localhost |                  |
| root | 127.0.0.1 |                  |
| root | ::1       |                  |
| pma  | localhost |                  |
| budi | localhost | *3E5287812B7D1F947439AC45E73935377A3ADEF7 |
+-----+-----+
```

Sekarang, setiap kali user **budi** login dia harus memasukkan password **rahasia**.

Tapi apa maksud karakter ‘aneh’ di kolom password untuk user budi? Bukankah kita membuat password berupa string **rahasia**?

Karakter aneh tersebut, yakni *3E5287812B7D1F947439AC45E73935377A3ADEF7 adalah hasil dari proses **hashing** untuk string **rahasia**.

Secara sederhana, hashing adalah metode khusus yang dipakai untuk mengkonversi sebuah string menjadi karakter “acak” yang memiliki panjang tetap. Tujuannya, jika ada *hacker* yang berhasil masuk ke database mysql, dia tidak bisa membaca password untuk setiap user.

Secara bawaan, panjang karakter hashing untuk password di dalam MySQL sebanyak 41 karakter. Berapapun panjang password yang diinput, hasil hashingnya tetap 41 karakter.

Hashing bersifat pengkodean satu arah, dimana karakter acak hasil hashing tidak bisa dikembalikan lagi menjadi karakter awal. Secara teori tidak ada cara untuk mengkonversi kode *3E5287812B7D1F947439AC45E73935377A3ADEF7 menjadi string **rahasia**.



Dalam prakteknya, terdapat beberapa metode khusus untuk ‘membobol’ hashing, misalnya membuat *rainbow tabel*. **Rainbow tabel** adalah daftar tabel hasil hashing untuk string-string umum. Oleh karena itu kita selalu disarankan untuk membuat password yang susah ditebak dengan kombinasi huruf, angka, dan huruf besar.

Hashing merupakan salah satu materi yang menjadi bagian dari kriptografi. **Kriptografi** sendiri adalah ilmu yang mempelajari cara menyembunyikan pesan agar tidak bisa dibaca oleh pihak yang tidak diinginkan.

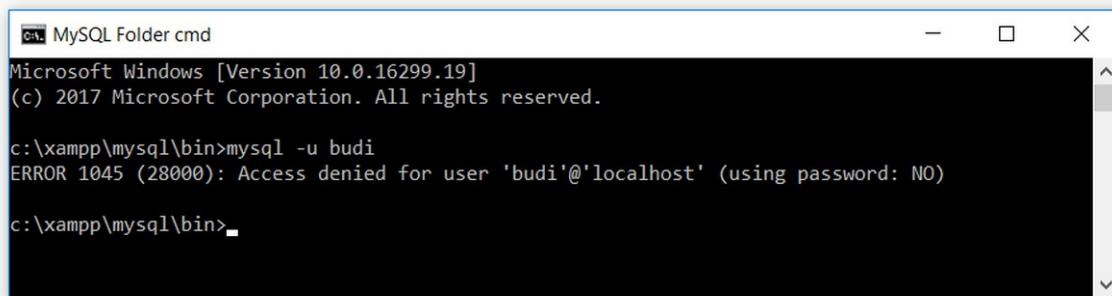
Bahasan tentang hashing serta kriptografi cukup kompleks, namun setidaknya kita bisa memahami bahwa di tabel `user`, password yang disimpan bukanlah karakter “mentah”, tapi hasil dari hashing.

Untuk menghasilkan karakter hashing ini, tersedia fungsi `PASSWORD()`. Sebagai contoh, jika saya menjalankan perintah `SELECT PASSWORD('rahasia')`, hasilnya sama dengan karakter yang ada di dalam kolom `password` untuk user `budi`:

```
SELECT PASSWORD('rahasia');
+-----+
| PASSWORD('rahasia') |
+-----+
| *3E5287812B7D1F947439AC45E73935377A3ADEF7 |
+-----+
```

Mari kita praktikkan cara masuk ke mysql menggunakan user `budi` yang memiliki password. Silahkan buka jendela cmd kedua, kemudian login dengan perintah berikut:

```
c:\xampp\mysql\bin>mysql -u budi
-- ERROR 1045 (28000): Access denied for user 'budi'@'localhost'
-- (using password: NO)
```



Gambar: Pesan error saat masuk sebagai user budi

Error diatas terjadi karena saya masuk tanpa menulis password untuk user `budi`.

Untuk login dengan password, terdapat beberapa cara. Pertama, kita bisa menulis langsung tambahan kode `-password` berserta nama user seperti contoh berikut:

```
c:\xampp\mysql\bin>mysql -u budi -prahasia
```

```
c:\xampp\mysql\bin>mysql -u budi -prahasia
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 4
Server version: 10.1.25-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

Gambar: Masuk sebagai user budi dengan password **rahasia**

Jika ditulis seperti ini, tidak boleh ada spasi antara **-p** dengan **isi password**. Yang harus ditulis adalah **-prahasia**, bukan **-p rahasia**.

Kelemahan dari cara login diatas adalah, karakter password bisa terlihat langsung di layar. Orang lain yang kebetulan ada di samping/belakang kita bisa melihat password tersebut.

Cara lain, bisa menggunakan alternatif penulisan kedua, yakni hanya menulis **-p** saja. Layar cmd akan meminta isian password di baris berikutnya:

```
c:\xampp\mysql\bin>mysql -u budi -p
Enter password: *****
```

```
c:\xampp\mysql\bin>mysql -u budi -p
Enter password: *****
Welcome to the MariaDB monitor. Commands end with ; or \g.
Your MariaDB connection id is 5
Server version: 10.1.25-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2017, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

Gambar: Isian password diinput di baris kedua

Setelah menulis **mysql -u budi -p**, cursor cmd akan berhenti sesaat menunggu kita mengetik isian password. Tampilan password akan disamarkan dengan karakter bintang.

Bagaimana jika kita ingin menukar password dari user budi? Tersedia query **SET PASSWORD FOR**. Berikut contoh penggunaannya:

```
SET PASSWORD FOR 'budi'@'localhost' = PASSWORD('qwerty');
-- Query OK, 0 rows affected (0.07 sec)
```

```
SELECT User, Host, Password FROM user;
+-----+-----+
| User | Host      | Password          |
+-----+-----+
| root | localhost |                  |
| root | 127.0.0.1 |                  |
| root | ::1       |                  |
| pma  | localhost |                  |
| budi | localhost | *AA1420F182E88B9E5F874F6FBE7459291E8F4601 |
+-----+-----+
```

Sekarang password untuk budi sudah berubah jadi `qwerty`.



Jika anda ingin memberikan password kepada user root caranya juga sama. Namun agar seragam dan tidak bentrok dengan aplikasi phpMyAdmin, saya akan biarkan password root tanpa password.

Alternatif lain, kita bisa update secara manual kolom password dari tabel user menggunakan query UPDATE :

```
UPDATE User SET Password = PASSWORD('rahasia') WHERE User = 'budi';
-- Query OK, 1 row affected (0.10 sec)
-- Rows matched: 1  Changed: 1  Warnings: 0
```

```
FLUSH PRIVILEGES;
-- Query OK, 0 rows affected (0.00 sec)
```

```
SELECT User, Host, Password FROM user;
+-----+-----+
| User | Host      | Password          |
+-----+-----+
| root | localhost |                  |
| root | 127.0.0.1 |                  |
| root | ::1       |                  |
| pma  | localhost |                  |
| budi | localhost | *3E5287812B7D1F947439AC45E73935377A3ADEF7 |
+-----+-----+
```

Karena kita mengubah isi tabel User secara manual, perintah FLUSH PRIVILEGES harus dijalankan setelah query UPDATE.

Ingin menghapus password dari user budi? Tinggal berikan string kosong ke dalam fungsi `PASSWORD()`, seperti contoh berikut:

```

UPDATE User SET Password = PASSWORD('') WHERE User = 'budi';
-- Query OK, 1 row affected (0.00 sec)
-- Rows matched: 1  Changed: 1  Warnings: 0

FLUSH PRIVILEGES;
-- Query OK, 0 rows affected (0.00 sec)

SELECT User, Host, Password FROM user;
+-----+-----+-----+
| User | Host      | Password |
+-----+-----+-----+
| root | localhost |          |
| root | 127.0.0.1 |          |
| root | ::1       |          |
| pma  | localhost |          |
| budi | localhost |          |
+-----+-----+-----+

```

Hasilnya, user budi tidak lagi memiliki password.

Khusus untuk versi terbaru MySQL Server, yakni dari versi MySQL 5.7.6 ke atas, tersedia query ALTER USER dengan contoh sebagai berikut:

```
ALTER USER nama_user IDENTIFIED BY 'string_password';
```

Namun query diatas tidak bisa berjalan di versi MariaDB yang saya gunakan.

Sebelum lanjut ke materi berikutnya, saya akan mengembalikan password qwerty untuk user budi:

```

SET PASSWORD FOR 'budi'@'localhost' = PASSWORD('qwerty');
-- Query OK, 0 rows affected (0.02 sec)

```

19.6 Mengenal Hak Akses (Privilege)

Dalam materi database, istilah **privilege** berarti hak yang dimiliki oleh sebuah user. Secara sederhana **privilege** ini juga bisa disebut sebagai **hak akses**. User yang baru saja kita buat belum bisa melakukan apa-apa di dalam MySQL. User tersebut harus diberikan hak akses (privilege) terlebih dahulu.

MySQL menyediakan fitur yang cukup detail mengenai privilege. Pembatasannya mulai dari query apa saja yang boleh dijalankan, database atau tabel apa saja yang bisa diakses, hingga kolom mana saja yang bisa ditampilkan.

Di dalam MySQL, proses pembuatan hak akses menggunakan query GRANT. Berikut format dasarnya:

```
GRANT jenis_hak_akses (kolom)
ON [nama_tabel.nama database]
TO 'namauuser@host' [IDENTIFIED BY 'password']
[WITH GRANT OPTION]
```

Kita akan bahas query GRANT ini secara bertahap.

Sebagai bahan praktek, saya membuka 2 jendela cmd Windows untuk 2 buah user: **root** dan **budi**. User **root** dipakai untuk menulis query GRANT, sedangkan **user budi** dipakai untuk melihat efek dari query GRANT tersebut. Kedua jendela ini akan kita akses secara bergantian.

The screenshot shows two separate command-line windows. The top window is titled 'MySQL Folder cmd - mysql -u budi -pqwerty'. It displays the MySQL monitor welcome message, connection information (connection id 3), and server version (10.1.25-MariaDB mariadb.org binary distribution). The bottom window is titled 'MySQL Folder cmd - mysql -u root'. It also displays the MySQL monitor welcome message, connection information (connection id 2), and server version (10.1.25-MariaDB mariadb.org binary distribution). Two red arrows point from the text 'Welcome to the MariaDB monitor.' in each window towards the respective MySQL client sessions.

Gambar: Dua buah jendela cmd mysql client untuk user root dan budi



User **budi** ini berasal dari materi sebelumnya, atau anda bisa menjalankan query CREATE USER 'budi'@'localhost' IDENTIFIED BY 'qwerty'.

Untuk melihat apa saja hak atau privilege dari sebuah user, bisa dengan menjalankan query SHOW GRANTS.

Berikut hasil yang didapat dari query SHOW GRANTS ketika dijalankan dari user **root**:

```
SHOW GRANTS;
+-----+
| Grants for root@localhost
+-----+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' WITH GRANT OPTION |
| GRANT PROXY ON ''@''%'' TO 'root'@'localhost' WITH GRANT OPTION |
+-----+
```

Artinya, user **root** memiliki semua hak akses (ALL PRIVILEGES) untuk semua database dan tabel (ON .) serta bisa memberikan hak akses ini ke user lain (WITH GRANT OPTION).

Dan berikut hasil yang didapat ketika dijalankan dari user **budi**:

```
SHOW GRANTS;
+-----+
| Grants for budi@localhost
+-----+
| GRANT USAGE ON *.* TO 'budi'@'localhost' IDENTIFIED BY PASSWORD '*AA1...' |
+-----+
```

Bagian GRANT USAGE sebenarnya berarti *tidak memiliki hak akses* atau *no privileges*. Artinya, user **budi** tidak memiliki privilege untuk database dan tabel apapun di dalam MySQL. Kata **USAGE** disini bisa dimaknai sebagai hak untuk masuk ke dalam MySQL.

Menggunakan query GRANT dari user **root**, kita akan berikan hak akses (privileges) kepada user **budi**. Setiap hak akses yang diberikan akan terlihat dari query GRANT USAGE.

19.7 Membuat Hak Akses (Privilege)

Dengan hak akses yang dimiliki user **budi** saat ini, database **belajar** tidak bisa diakses. Bahkan dari query SHOW DATABASES, database **belajar** tidak terlihat. Ini karena user **budi** tidak memiliki privileges untuk itu.

Sebagai contoh pertama, saya ingin memberikan hak akses kepada user **budi** agar bisa melakukan apapun ke dalam tabel **mahasiswa** yang ada di database **belajar**. Caranya, dari user **root** ketik kode berikut:

```
GRANT ALL ON belajar.mahasiswa TO budi@localhost;
-- Query OK, 0 rows affected (0.04 sec)
```

Query GRANT diatas bisa dibagi menjadi 3 bagian:

- Query yang dibolehkan: GRANT ALL. Artinya kita mengizinkan user **budi** untuk menjalankan semua query.
- Tabel atau database yang diizinkan: ON **belajar.mahasiswa**. Artinya, user **budi** bisa menjalankan semua query (**ALL**) untuk tabel **mahasiswa** yang berada di database **belajar**.
- Untuk user: TO **budi@localhost**. Inilah user yang diberi izin.

Setelah query dijalankan, buka jendela cmd untuk user **budi** dan periksa menggunakan query SHOW GRANTS:

```
SHOW GRANTS;
+-----+
| Grants for budi@localhost
+-----+
| GRANT USAGE ON *.* TO 'budi'@'localhost' IDENTIFIED BY PASSWORD '*AA1...' |
| GRANT ALL PRIVILEGES ON `belajar`.`mahasiswa` TO 'budi'@'localhost'           |
+-----+
```

Sekarang terlihat tambahan baris baru, yakni GRANT ALL PRIVILEGES ON belajar.mahasiswa. Dengan tambahan baris ini, user **budi** sudah diberikan hak akses untuk tabel mahasiswa di database belajar.

Jika kita menjalankan query SHOW DATABASES dari user **budi**, database belajar sudah terlihat:

```
SHOW DATABASES;
+-----+
| Database
+-----+
| belajar
| information_schema
| test
+-----+

USE belajar;
-- Database changed

SHOW TABLES;
+-----+
| Tables_in_belajar
+-----+
| mahasiswa
+-----+
```

Sebenarnya di dalam database belajar terdapat banyak tabel, tapi yang bisa diakses oleh user **budi** hanya tabel **mahasiswa**. Ini sesuai dengan hak akses yang kita berikan sebelumnya.

User **budi** memiliki ALL PRIVILEGES untuk tabel **mahasiswa**. Dengan demikian kita bisa menjalankan semua query ke dalam tabel ini. Berikut percobaannya:

```
SELECT * FROM mahasiswa;
+-----+-----+-----+-----+
| nim      | nama           | asal      | jurusan       |
+-----+-----+-----+-----+
| 17080305 | Rina Kumala Sari | Jakarta   | Akuntansi    |
| 17090113 | Riana Putria    | Padang    | Kimia         |
| 17090222 | Sari Citra Lestari | Jakarta   | Manajemen    |
| 17140143 | Rudi Permana    | Bandung   | Ilmu Komputer|
+-----+-----+-----+-----+
UPDATE mahasiswa SET asal = 'Semarang' WHERE nim = '17080305';
-- Query OK, 1 row affected (0.10 sec)
-- Rows matched: 1  Changed: 1  Warnings: 0
```

```
SELECT * FROM mahasiswa;
+-----+-----+-----+-----+
| nim      | nama           | asal      | jurusan       |
+-----+-----+-----+-----+
| 17080305 | Rina Kumala Sari | Semarang  | Akuntansi    |
| 17090113 | Riana Putria    | Padang    | Kimia         |
| 17090222 | Sari Citra Lestari | Jakarta   | Manajemen    |
| 17140143 | Rudi Permana    | Bandung   | Ilmu Komputer|
+-----+-----+-----+-----+
```

Query SELECT dan UPDATE sukses dijalankan oleh user **budi** ke dalam tabel **mahasiswa**.

Berikutnya, mari kita coba membuat query yang lebih terbatas untuk user **budi**. Beralih ke cmd user **root**, lalu jalankan query berikut:

```
GRANT SELECT ON belajar.universitas TO budi@localhost;
-- Query OK, 0 rows affected (0.01 sec)
```

Query diatas berarti saya memberikan hak akses query SELECT untuk tabel **universitas** di database **belajar** kepada user **budi**. Perhatikan bahwa hak aksesnya bukan lagi GRANT ALL, tetapi GRANT SELECT.

Buka cmd user **budi** dan mari periksa hasil query SHOW GRANTS:

```
SHOW GRANTS;
+-----+
| Grants for budi@localhost
+-----+
| GRANT USAGE ON *.* TO 'budi'@'localhost' IDENTIFIED BY PASSWORD '*AA1...'
| GRANT ALL PRIVILEGES ON `belajar`.`mahasiswa` TO 'budi'@'localhost'
| GRANT SELECT ON `belajar`.`universitas` TO 'budi'@'localhost'
+-----+
```

Di baris ketiga muncul hak akses baru, yakni penggunaan query SELECT untuk tabel universitas. Mari kita coba:

```
SHOW TABLES;
+-----+
| Tables_in_belajar |
+-----+
| mahasiswa          |
| universitas         |
+-----+

SELECT * FROM universitas;
+-----+
| jurusan      | tgl_berdiri | nama_dekan
+-----+
| Akuntansi    | 1985-03-19  | Maya Fitrianti, M.M.
| Farmasi       | 1997-05-30  | Prof. Silvia Nst, M.Farm.
| Ilmu Komputer | 2003-02-23 | Dr. Syahrial, M.Kom.
| Kimia         | 1987-07-12  | Prof. Mulyono, M.Sc.
+-----+

DELETE FROM universitas WHERE jurusan = 'Akuntansi';
-- ERROR 1142 (42000): DELETE command denied to user 'budi'@'localhost'
-- for table 'universitas'

UPDATE universitas SET nama_dekan = 'Prof. Pane Tambunan'
WHERE jurusan = 'Akuntansi';
-- ERROR 1142 (42000): UPDATE command denied to user 'budi'@'localhost'
-- for table 'universitas'
```

Dari hasil query SHOW TABLES, user **budi** sudah diberikan hak akses untuk tabel `mahasiswa` dan `universitas`. Query `SELECT * FROM universitas` berhasil dijalankan.

Tapi tidak halnya dengan query `DELETE` dan `UPDATE`, pada saat menjalankan kedua perintah ini, hasilnya berupa pesan error: *command denied*. Hal ini terjadi karena privilege atau hak yang dimiliki user **budi** hanyalah query `SELECT` saja. Query selain `SELECT` tidak bisa dijalankan untuk tabel `universitas`.

Pembatasan query seperti ini cukup penting untuk dipahami. Kita bisa membuat sebuah user yang hanya bisa melihat data saja, namun tidak bisa mengubah atau menghapus data yang ada.

Bagaimana jika ternyata user **budi** “naik pangkat” dan memperoleh izin untuk mengubah data yang ada di dalam tabel `universitas`? Kita tinggal menambahkan hak akses baru. Silahkan buka cmd Windows dengan user **root**, lalu jalankan perintah berikut:

```
GRANT UPDATE, DELETE ON belajar.universitas TO budi@localhost;
-- Query OK, 0 rows affected (0.01 sec)
```

Dalam perintah ini, saya memberikan hak akses query UPDATE dan DELETE kepada user **budi** untuk tabel `universitas`.



Jika anda ingin menambahkan query lain, tinggal ditulis sebelum perintah `ON`. Misalnya user **budi** juga diberikan hak untuk query `ALTER`, maka penulisannya adalah: `GRANT UPDATE, DELETE, ALTER ON belajar.universitas ...`

Buka cmd Windows untuk user **budi** dan periksa dari query `SHOW GRANTS`:

```
SHOW GRANTS;
+-----+
| Grants for budi@localhost
+-----+
| GRANT USAGE ON *.* TO 'budi'@'localhost' IDENTIFIED BY PASSWORD '*AA1... '
| GRANT ALL PRIVILEGES ON `belajar`.`mahasiswa` TO 'budi'@'localhost'
| GRANT SELECT, UPDATE, DELETE ON `belajar`.`universitas` TO 'budi'@'...' |
+-----+
```

Di baris ketiga sekarang tercantum 3 query yang diperbolehkan, yakni `SELECT`, `UPDATE` dan `DELETE` untuk tabel `universitas`.

Untuk uji coba, anda bisa jalankan query `DELETE` dan `UPDATE` sebelumnya dan perhatikan apakah kali ini user **budi** diizinkan untuk melakukan hal tersebut.

Untuk pembatasan yang lebih presisi, kita bisa melakukannya hingga ke level kolom. Maksudnya, seorang user bisa dibatasi hanya bisa melihat kolom tertentu saja di dalam sebuah tabel.

Sebagai tabel sample, saya akan menggunakan tabel `provinsi` yang sudah kita buat pada bab sebelum ini. Secara keseluruhan, tabel `provinsi` memiliki 6 kolom:

```
SELECT * FROM provinsi;
+-----+-----+-----+-----+-----+
| nama_prov | kode_iso | ibukota      | populasi | luas       | apbd      |
+-----+-----+-----+-----+-----+
| Yogyakarta | YO        | Yogyakarta   | 3553100  | 3133.00    | 3662.33   |
| ...        | ..        | ...          | ...       | ...        | ...       |
| Aceh       | AC        | Banda Aceh   | 4906835  | 57365.00   | 15572.50   |
| Maluku    | MA        | Ambon        | 1657409  | 49350.00   | 2013.83   |
+-----+-----+-----+-----+-----+
34 rows in set (0.03 sec)
```

Hasil tampilan diatas hanya bisa dijalankan dari user **root** karena user **budi** belum diberikan hak akses untuk tabel **provinsi**.

Sekarang saya ingin agar user **budi** bisa mengakses tabel **provinsi**, namun dibatasi hanya bisa melihat kolom **nama_prov** dan **apbd** saja. User **budi** tidak diperbolehkan melihat kolom lain. Selain itu user **budi** juga diizinkan untuk mengupdate kolom **apbd**.

Buka cmd Windows untuk user **root**, dan berikut cara penulisan query **GRANT** yang diperlukan:

```
GRANT SELECT (nama_prov, apbd), UPDATE (apbd) ON belajar.provinsi
TO budi@localhost;
-- Query OK, 0 rows affected (0.00 sec)
```

Untuk membatasi kolom apa saja yang bisa diakses, tulis nama kolom setelah nama query. Dalam perintah diatas, **GRANT SELECT (nama_prov, apbd), UPDATE (apbd)** artinya query **SELECT** hanya diperbolehkan untuk kolom **nama_prov** dan **apbd**, serta query **UPDATE** hanya bisa untuk kolom **apbd** saja.

Mari kita coba akses tabel **provinsi** dari user **budi**:

```
SHOW TABLES;
+-----+
| Tables_in_belajar |
+-----+
| mahasiswa         |
| provinsi          |
| universitas       |
+-----+
3 rows in set (0.00 sec)
```

```
SELECT * FROM provinsi;
-- ERROR 1142 (42000): SELECT command denied to user 'budi'@'localhost'
-- for table 'provinsi'
```

Apa yang terjadi? Kenapa query **SELECT** tetap di tolak? Padahal hak aksesnya sudah diberikan kepada user **budi**.

Alasan dari error diatas adalah karena penggunaan query `SELECT *` yang akan mengakses **seluruh kolom** dari tabel provinsi. User **budi** tidak memiliki hak untuk seluruh kolom, tapi hanya kolom `nama_prov` dan `apbd` saja. Oleh karena itu, query `SELECT` harus ditulis sebagai berikut:

```
SELECT nama_prov, apbd FROM provinsi;
+-----+-----+
| nama_prov      | apbd    |
+-----+-----+
| Yogyakarta     | 3662.33 |
| Kalimantan Utara | 2299.60 |
| ...            | ...     |
| Aceh           | 15572.50 |
| Maluku         | 2013.83 |
+-----+-----+
34 rows in set (0.00 sec)
```

User **budi** juga memiliki privilege untuk query `UPDATE`, tapi hanya untuk kolom `apbd` saja:

```
UPDATE provinsi SET apbd = 4000.99 WHERE nama_prov = 'Yogyakarta';
-- Query OK, 1 row affected (0.18 sec)
-- Rows matched: 1  Changed: 1  Warnings: 0
```

```
SELECT nama_prov, apbd FROM provinsi;
+-----+-----+
| nama_prov      | apbd    |
+-----+-----+
| Yogyakarta     | 4000.99 |
| Kalimantan Utara | 2299.60 |
| ...            | ...     |
| Maluku         | 2013.83 |
+-----+-----+
34 rows in set (0.00 sec)
```

```
DELETE FROM provinsi WHERE nama_prov = 'Yogyakarta';
-- ERROR 1142 (42000): DELETE command denied to user 'budi'@'localhost'
-- for table 'provinsi'
```

Di baris terakhir saya mencoba menjalankan query `DELETE`. Seperti yang bisa kita perkirakan, MySQL akan menolak perintah ini karena user **budi** tidak memiliki hak akses untuk query tersebut.

19.8 Menghapus Hak Akses (Privilege)

Hak akses yang sudah diberikan kepada sebuah user bisa dihapus atau dicabut. Untuk keperluan ini tersedia query `REVOKE`. Cara penulisan query `REVOKE` mirip seperti query `GRANT`, dimana kita menulis apa saja privilege yang ingin dicabut.

Berikut format dasar penulisan query REVOKE:

```
REVOKE jenis_hak_akses (kolom)
ON [nama_tabel.nama database]
FROM 'namauser@host' [IDENTIFIED BY 'password']
```

Jika pada saat pembuatan hak akses kita menulis GRANT . . . TO, maka untuk menghapus hak akses perintahnya adalah REVOKE . . . FROM.

Kita akan praktik menghapus beberapa privilege dari user **budi**. Tapi sebelum itu mari lihat apa saja hak akses yang bisa dilakukan oleh **budi** saat ini:

```
SHOW GRANTS;
```

+-----+	
Grants for budi@localhost	
+-----+	
GRANT USAGE ON *.* TO 'budi'@'localhost' IDENTIFIED BY PASSWORD '*AA1...'	
GRANT SELECT (nama_prov, apbd), UPDATE (apbd) ON `belajar`.`provinsi` . . .	
GRANT SELECT, UPDATE, DELETE ON `belajar`.`universitas` TO 'budi'@'...'	
GRANT ALL PRIVILEGES ON `belajar`.`mahasiswa` TO 'budi'@'localhost'	
+-----+	

Perhatikan baris kedua dari hasil query SHOW GRANTS, user **budi** bisa menjalankan query SELECT ke kolom **nama_prov** dan **apbd**, serta query UPDATE untuk kolom **apbd** di tabel **provinsi**.

Saya ingin menghapus hak akses SELECT ini. Caranya, jalankan perintah berikut dari user **root**:

```
REVOKE SELECT(nama_prov) ON belajar.provinsi FROM budi@localhost;
-- Query OK, 0 rows affected (0.00 sec)
```

Query diatas akan mencabut hak akses user **budi** untuk query SELECT pada kolom **nama_prov** dari tabel **provinsi**. Akibat dari perintah ini, hak akses user **budi** akan berkurang. Berikut hasil dari query SHOW GRANTS:

```
SHOW GRANTS;
```

+-----+	
Grants for budi@localhost	
+-----+	
GRANT USAGE ON *.* TO 'budi'@'localhost' IDENTIFIED BY PASSWORD '*AA1...'	
GRANT SELECT (apbd), UPDATE (apbd) ON `belajar`.`provinsi` TO 'budi'@'...'	
GRANT SELECT, UPDATE, DELETE ON `belajar`.`universitas` TO 'budi'@'...'	
GRANT ALL PRIVILEGES ON `belajar`.`mahasiswa` TO 'budi'@'localhost'	
+-----+	

Sekarang di baris kedua hanya ada GRANT SELECT (apbd). Artinya, user **budi** tidak bisa lagi mengakses kolom **nama_prov** karena hak aksesnya sudah dicabut. Mari coba akses tabel **provinsi** ini:

```
SELECT nama_prov, apbd FROM provinsi;
-- ERROR 1143 (42000): SELECT command denied to user 'budi'@'localhost'
-- for column 'nama_prov' in table 'provinsi'

SELECT apbd FROM provinsi;
+-----+
| apbd    |
+-----+
| 4000.99 |
| 2299.60 |
| ...     |
| 2013.83 |
+-----+
34 rows in set (0.03 sec)
```

Kolom apdb tetap bisa diakses karena user **budi** masih memiliki hak akses untuk itu.

Sebagai contoh selanjutnya, saya ingin menghapus seluruh hak akses user **budi** untuk tabel provinsi. Silahkan buka user **root** lalu jalankan query berikut:

```
REVOKE ALL ON belajar.provinsi FROM budi@localhost;
-- Query OK, 0 rows affected (0.00 sec)
```

Sama seperti GRANT ALL, query REVOKE ALL akan menghapus hak akses untuk semua query. Mari kita cek dari user **budi**:

```
SHOW GRANTS;
+-----+
| Grants for budi@localhost                                |
+-----+
| GRANT USAGE ON *.* TO 'budi'@'localhost' IDENTIFIED BY PASSWORD '*AA1...' |
| GRANT ALL PRIVILEGES ON `belajar`.`mahasiswa` TO 'budi'@'localhost'          |
| GRANT SELECT, UPDATE, DELETE ON `belajar`.`universitas` TO 'budi'@'...'      |
+-----+
```

Tidak terlihat lagi hak akses untuk tabel provinsi. Artinya, user budi tidak bisa lagi mengakses tabel tersebut.

Sebagai latihan, bisakah anda membuat query untuk menghapus seluruh hak akses user **budi** dari tabel **mahasiswa** dan **universitas**?

Baik, berikut querynya:

```
REVOKE ALL ON belajar.universitas FROM budi@localhost;
-- Query OK, 0 rows affected (0.00 sec)
```

```
REVOKE ALL ON belajar.mahasiswa FROM budi@localhost;
-- Query OK, 0 rows affected (0.00 sec)
```

Sekarang, semua hak akses untuk user **budi** sudah dihapus, kecuali GRANT USAGE yang merupakan hak akses bawaan dari sebuah user (hak untuk login ke MySQL Server):

```
SHOW GRANTS;
+-----+
| Grants for budi@localhost |
+-----+
| GRANT USAGE ON *.* TO 'budi'@'localhost' IDENTIFIED BY PASSWORD '*AA1...' |
+-----+
```

19.9 Permissible Privileges

Dalam pembahasan tentang query GRANT, kita sudah menggunakan beberapa pembatasan query seperti GRANT SELECT, GRANT UPDATE, serta penulisan singkat GRANT ALL yang berarti “semua query”.

Tentu saja kita juga bisa menulis query lain yang diperbolehkan, atau dikenal sebagai **permissible privileges**. Misalnya saya ingin user **budi** diperbolehkan untuk membuat user lain, maka hak aksesnya adalah GRANT CREATE USER. Atau user budi diizinkan untuk membuat index, maka hak aksesnya adalah GRANT INDEX.

Terdapat sekitar 31 *permissible privileges* yang bisa diberikan ke seorang user. Berikut daftarnya:

- ALTER
- ALTER ROUTINE
- CREATE
- CREATE ROUTINE
- CREATE TABLESPACE
- CREATE TEMPORARY TABLES
- CREATE USER
- CREATE VIEW
- DELETE
- DROP
- EVENT
- EXECUTE
- FILE
- GRANT OPTION
- INDEX

- INSERT
- LOCK TABLES
- PROCESS
- PROXY
- REFERENCES
- RELOAD
- REPLICATION CLIENT
- REPLICATION SLAVE
- SELECT
- SHOW DATABASES
- SHOW VIEW
- SHUTDOWN
- SUPER
- TRIGGER
- UPDATE
- USAGE

Sebagian besar dari query diatas sudah kita bahas dalam bab-bab sebelumnya. Beberapa diantaranya termasuk kategori advanced yang memang tidak dibahas dalam buku ini seperti REPLICATION CLIENT dan REPLICATION SLAVE.

Jika anda tertarik untuk mempelajari lebih lanjut tentang arti setiap query serta efeknya ke hak akses, bisa dibaca-baca ke [MySQL Reference Manual: GRANT Syntax²](#)

19.10 Hak Akses untuk Seluruh Tabel dan Database

Dalam materi query GRANT sebelum ini, saya membuat satu persatu hak akses untuk setiap tabel. Dengan menggunakan tanda bintang (*) sebagai pengganti nama tabel dan database, kita bisa membuat hak akses untuk seluruh tabel dan seluruh database. Cara penggunaannya mirip seperti penulisan Identifier Qualifiers.

Sebagai contoh, jika saya ingin user budi bisa menjalankan query SELECT dan DELETE untuk seluruh tabel di dalam database belajar, query GRANT-nya adalah sebagai berikut:

```
GRANT SELECT, DELETE ON belajar.* TO budi@localhost;
-- Query OK, 0 rows affected (0.00 sec)
```

Berikut hasil dari query SHOW GRANTS dari user budi:

²<https://dev.mysql.com/doc/refman/5.7/en/grant.html>

```
SHOW GRANTS;
+-----+
| Grants for budi@localhost
+-----+
| GRANT USAGE ON *.* TO 'budi'@'localhost' IDENTIFIED BY PASSWORD '*AA1...'
| GRANT SELECT, DELETE ON `belajar`.* TO 'budi'@'localhost'
+-----+
```

Sekarang, user **budi** bisa melihat seluruh tabel di dalam database **belajar**.

Akan tetapi, perubahan hak akses ini tidak langsung efektif. Jika tetap tidak bisa mengakses database **belajar**, bisa keluar dulu dan login kembali sebagai **budi**.

Contoh lain, saya ingin user **budi** bisa menggunakan seluruh query untuk seluruh database. Query GRANT-nya adalah sebagai berikut:

```
GRANT ALL ON *.* TO budi@localhost;
-- Query OK, 0 rows affected (0.00 sec)
```

Dengan perintah ini, user **budi** bisa melakukan hampir semua hal di dalam MySQL (hak akses mirip seperti **root**). Dia bisa menjalankan query apa saja untuk setiap database, termasuk perintah management server seperti membuat dan menghapus user (CREATE USER dan DROP USER).

Namun ada 1 hal yang belum bisa dilakukan user **budi**, dia tidak bisa menjalankan query GRANT. Untuk hak akses ini, perlu sebuah perintah khusus yang akan kita bahas sesaat lagi.

Untuk menghapus hak akses yang ditulis menggunakan tanda bintang, cara penulisannya juga sama:

```
REVOKE SELECT, DELETE ON belajar.* FROM budi@localhost;
-- Query OK, 0 rows affected (0.00 sec)
```

```
REVOKE ALL ON *.* FROM budi@localhost;
-- Query OK, 0 rows affected (0.00 sec)
```

Kedua query diatas akan menghapus hak akses yang sebelumnya kita berikan kepada user **budi**.

19.11 Hak akses WITH GRANT OPTION

Dalam contoh sebelum ini, user **budi** saya berikan hak akses GRANT ALL ON *.*. Query ini membuat budi bisa melakukan ‘hampir’ segalanya di dalam MySQL. Kecuali satu hal, yakni memberikan hak akses kepada user lain (menjalankan query GRANT).

Perintah WITH GRANT OPTION membolehkan sebuah user untuk memberikan privileges yang dipunyai kepada user lain. Misalkan user **budi** memiliki hak akses untuk menjalankan query SELECT di tabel **mahasiswa**. Ia bisa saja memberikan hak akses tersebut kepada user **alex**.



Mengizinkan seorang user untuk memberikan hak akses ke user lain tidak terlalu sering dipakai karena bisa berdampak besar. Akan tetapi jika kewenangan ini ingin diberikan, kita harus menambahkan perintah `WITH GRANT OPTION` pada saat pemberian hak akses.

Mari kita coba dengan contoh praktik. Pertama, saya akan memberikan hak akses `GRANT ALL` kepada user **budi**. Jalankan query berikut dari user **root**:

```
GRANT ALL ON *.* TO budi@localhost;
-- Query OK, 0 rows affected (0.00 sec)
```

Dengan query diatas, user **budi** bisa melakukan banyak hal, termasuk membuat user lain:

```
SHOW GRANTS;
+-----+
| Grants for budi@localhost
+-----+
| GRANT ALL PRIVILEGES ON *.* TO 'budi'@'localhost' IDENTIFIED BY ... |
+-----+
```

```
CREATE USER alex@localhost;
Query OK, 0 rows affected (0.00 sec)
```

```
SELECT User, Host, Password FROM mysql.user;
+-----+
| User | Host      | Password          |
+-----+
| root | localhost |                  |
| root | 127.0.0.1 |                  |
| root | ::1       |                  |
| pma  | localhost |                  |
| budi | localhost | *AA1420F182E88B9E5F874F6FBE7459291E8F4601 |
| alex | localhost |                  |
+-----+
```

```
GRANT SELECT ON belajar.universitas TO alex@localhost;
-- ERROR 1142 (42000): GRANT command denied to user 'budi'@'localhost'
-- for table 'universitas'
```

Seluruh query diatas saya jalankan dari user **budi**. Karena ia sudah memiliki hak akses `GRANT ALL`, maka kita bisa menjalankan query `CREATE USER` untuk membuat user **alex**. Akan tetapi seperti yang terlihat, user **budi** tidak bisa menjalankan query `GRANT`.

Untuk keperluan ini, kembali ke user **root** lalu jalankan query berikut:

```
GRANT ALL ON *.* TO budi@localhost WITH GRANT OPTION;
-- Query OK, 0 rows affected (0.00 sec)
```

Perintah diatas akan memberikan hak akses untuk seluruh query kepada user **budi** termasuk query GRANT. Kembali ke cmd Windows untuk user **budi** dan jalankan perintah berikut:

```
exit
-- Bye

c:\xampp\mysql\bin>mysql -u budi -pqwerty

SHOW GRANTS;
+-----+
| Grants for budi@localhost |
+-----+
| GRANT ALL PRIVILEGES ON *.* TO 'budi'@'localhost' ... WITH GRANT OPTION |
+-----+

GRANT SELECT ON belajar.universitas TO alex@localhost;
-- Query OK, 0 rows affected (0.00 sec)
```

Di awal saya keluar dulu (exit) agar pengaturan GRANT bisa berlaku untuk user **budi**. Diakhir hasil query SHOW GRANTS terdapat tambahan WITH GRANT OPTION. Inilah hak akses yang harus dimiliki seorang user agar bisa menjalankan query GRANT.

Yang juga harus menjadi catatan, jika ada 2 user memiliki hak akses berbeda namun bisa menjalankan query GRANT, user tersebut bisa saling memberikan hak akses (yang seharusnya hanya berlaku untuk diri sendiri saja). Karena itu selalu pertimbangkan efek samping dari perintah WITH GRANT OPTION.

19.12 SQL Mode: NO_AUTO_CREATE_USER

Di dalam SQL mode, terdapat sebuah pengaturan yang bernama NO_AUTO_CREATE_USER. Jika anda mengikuti materi tentang SQL mode di buku ini, pengaturan NO_AUTO_CREATE_USER sudah berlaku efektif.

Untuk memeriksanya, silahkan jalankan query berikut:

```
SELECT @@GLOBAL.sql_mode;
+-----+
| @@GLOBAL.sql_mode |
+-----+
| STRICT_ALL_TABLES,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+-----+
```

Jika mode NO_AUTO_CREATE_USER aktif, query GRANT tidak akan membuat user baru jika user yang di berikan hak akses tidak ditemukan. Namun jika pengaturan ini tidak ada, user baru akan otomatis dibuat. Agar pengertian ini bisa lebih jelas, kita akan lihat contoh prakteknya.

Pertama, saya akan coba membuat query GRANT kepada user yang belum ada di MySQL:

```
GRANT SELECT ON belajar.universitas TO rani@localhost;
-- ERROR 1133 (28000): Can't find any matching row in the user table
```

Error diatas terjadi karena MySQL tidak menemukan user **rani** yang memang belum ada. Sekarang, saya akan ubah pengaturan SQL mode dan menjalankan query yang sama:

```
SET GLOBAL sql_mode = 'STRICT_ALL_TABLES,NO_ENGINE_SUBSTITUTION';
-- Query OK, 0 rows affected (0.00 sec)
```

```
SELECT @@GLOBAL.sql_mode;
+-----+
| @@GLOBAL.sql_mode           |
+-----+
| STRICT_ALL_TABLES,NO_ENGINE_SUBSTITUTION |
+-----+
```

```
GRANT SELECT ON belajar.universitas TO rani@localhost;
-- Query OK, 0 rows affected (0.00 sec)
```

```
SELECT User, Host, Password FROM mysql.user;
+-----+-----+-----+
| User | Host      | Password          |
+-----+-----+-----+
| root | localhost |                |
| root | 127.0.0.1 |                |
| root | ::1       |                |
| pma  | localhost |                |
| rani | localhost |                |
| budi | localhost | *AA1420F182E88B9E5F874F6FBE7459291E8F4601 |
| alex | localhost |                |
+-----+-----+-----+
```

Setelah mengubah pengaturan SQL mode, saya menjalankan kembali query GRANT dan kali ini tidak ada error. Penyebabnya, query GRANT secara tidak langsung akan membuat user **rani**. Hal ini bisa terlihat dari isi tabel user dari database mysql.

Hal seperti ini bisa menjadi celah keamanan. Karena jika kita tidak teliti, sebuah user bisa tercipta dengan tidak sengaja. Pengaturan SQL mode NO_AUTO_CREATE_USER bisa mencegah kejadian seperti ini.

Akan tetapi, meskipun NO_AUTO_CREATE_USER aktif, query GRANT sebenarnya tetap bisa membuat user secara tidak langsung. Berikut contoh kasusnya:

```

SET GLOBAL sql_mode =
'NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION,STRICT_ALL_TABLES';
-- Query OK, 0 rows affected (1.13 sec)

SELECT @@GLOBAL.sql_mode;
+-----+
| @@GLOBAL.sql_mode |
+-----+
| STRICT_ALL_TABLES,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+-----+

GRANT ALL ON belajar.universitas TO sari@localhost;
-- ERROR 1133 (28000): Can't find any matching row in the user table

GRANT ALL ON belajar.universitas TO sari@localhost IDENTIFIED BY '112233';
-- Query OK, 0 rows affected (0.09 sec)

SELECT User, Host, Password FROM mysql.user;
+-----+
| User | Host      | Password          |
+-----+
| root | localhost |                  |
| root | 127.0.0.1 |                  |
| root | ::1       |                  |
| pma  | localhost |                  |
| rani | localhost |                  |
| budi | localhost | *AA1420F182E88B9E5F874F6FBE7459291E8F4601 |
| alex | localhost |                  |
| sari | localhost | *C42CF059802456312318BB928C3334F1A6133AB4 |
+-----+

```

Disini saya mengaktifkan kembali settingan NO_AUTO_CREATE_USER. Akibatnya query GRANT ALL ON belajar.universitas TO sari@localhost tidak bisa dijalankan.

Akan tetapi ketika ditambahkan perintah IDENTIFIED BY '112233', query GRANT bisa berjalan dan user **rani** secara tidak langsung juga akan dibuat.

Efek dari query GRANT seperti ini juga berlalu secara timbal balik. Ketika kita menghapus sebuah user, hak akses yang dimiliki user tersebut juga otomatis terhapus.

Menutup materi bab ini, saya akan menghapus user yang sudah kita buat:

```
DROP user budi@localhost;
DROP user alex@localhost;
DROP user rani@localhost;
DROP user sari@localhost;

SELECT User, Host, Password FROM mysql.user;
+-----+-----+-----+
| User | Host      | Password |
+-----+-----+-----+
| root | localhost |          |
| root | 127.0.0.1 |          |
| root | ::1       |          |
| pma  | localhost |          |
+-----+-----+-----+
```

Sekarang user yang tersisa hanya **root** dan **pma** yang digunakan khusus oleh aplikasi **phpMyAdmin**. Namun tentu saja anda boleh jika ingin menambahkan user baru sesuai kebutuhan.

Dalam bab ini kita telah mempelajari cara manajemen user di dalam MySQL, mulai dari pembuatan user hingga pembatasan hak akses. Yang juga perlu dipahami adalah, user di dalam MySQL belum tentu bersesuaian untuk user di dalam aplikasi.

Sebagai contoh, di website sistem informasi biasanya ada halaman login untuk masuk ke dalam website. Halaman login ini mengharuskan sebuah user dibuat terlebih dahulu, misalnya dari halaman register. User yang dipakai untuk login ke aplikasi, tidak harus dibuatkan user MySQLnya. Satu user MySQL bisa dipakai bersama-sama.

Jika saya ingin merancang sebuah sistem informasi kampus, ratusan mahasiswa bisa login ke dalam aplikasi. Setiap mahasiswa memiliki akun terpisah di dalam website, tapi bisa saja saya hanya menyediakan 1 user MySQL dengan nama **mahasiswa** yang dipakai oleh ratusan mahasiswa secara bersamaan. Hal ini karena setiap mahasiswa akan memiliki akses yang sama ke dalam database.

Dalam bab berikutnya, kita akan membahas tentang fungsi bawaan MySQL yang bisa dipakai untuk proses mengolah data.

20. MySQL Operator dan Function

Data yang sudah diinput ke dalam database biasanya butuh pemrosesan lebih lanjut. Misalnya ketika akan ditampilkan, kita bisa mengubah data tersebut ke bentuk dan format tertentu.

Contoh yang cukup sering adalah ketika ingin menampilkan nilai angka yang memiliki pecahan. Misalnya angka yang ada di dalam database tersimpan 5.5487632, pada saat data ditampilkan kita bisa menyederhanakannya ke dalam 2 tempat desimal, yakni sebagai 5.55. Tetapi data asli yang ada di database tetap 5.5487632.

Atau di dalam sebuah tabel terdapat pemisahan antara kolom nama depan dengan kolom nama belakang. Pada saat ditampilkan, kedua kolom tersebut bisa disatukan. Untuk mengolah data seperti ini, tersedia berbagai *operator* dan *function* bawaan MySQL.

Operator adalah instruksi khusus untuk menghasilkan nilai akhir. Contohnya seperti proses penambahan (+) dan pengurangan (-). Dalam bab ini kita akan membahas **operator aritmatika** yang terdapat di dalam MySQL. Operator aritmatika ini digunakan untuk tipe data **numeric**.

Selain itu juga akan dibahas tentang **function**. Meskipun MySQL dan MariaDB bukanlah sebuah bahasa pemrograman, tersedia berbagai function bawaan yang bisa dipakai untuk mengolah data. Di dalam MySQL terdapat ratusan function siap pakai yang daftar lengkapnya bisa dilihat ke: [MySQL Manual: Function and Operator Reference¹](#).

Dalam bab ini kita akan pelajari cara menjalankan function secara umum. Kemudian diikuti oleh 3 bab yang khusus membahas function bawaan MySQL untuk tipe data **numeric**, **string** dan **date**.

20.1 Operator Aritmatika MySQL

Sama seperti bahasa pemrograman pada umumnya, kita bisa melakukan operasi aritmatika di dalam MySQL. Berikut operator yang tersedia:

Operator	Keterangan
+	Penambahan
-	Pengurangan
*	Perkalian
/	Pembagian
DIV	Pembagian (angka bulat)
%, MOD	Sisa hasil bagi

Operator ini bisa digunakan langsung ke dalam tipe data angka (numeric), atau ke suatu kolom di sebuah tabel. Berikut percobaannya:

¹<https://dev.mysql.com/doc/refman/5.7/en/func-op-summary-ref.html>

```
SELECT 10 + 5;
```

```
+-----+
| 10 + 5 |
+-----+
|      15 |
+-----+
```

```
SELECT 10 - 5;
```

```
+-----+
| 10 - 5 |
+-----+
|      5 |
+-----+
```

```
SELECT 10 * 5;
```

```
+-----+
| 10 * 5 |
+-----+
|      50 |
+-----+
```

```
SELECT 10 / 5;
```

```
+-----+
| 10 / 5 |
+-----+
| 2.0000 |
+-----+
```

Untuk perhitungan yang kompleks, kita bisa menambahkan tanda kurung agar sebuah operasi bisa diproses sebelum operasi lainnya:

```
SELECT ((9 + 3) * 2) / 4;
```

```
+-----+
| ((9 + 3) * 2) / 4 |
+-----+
|       6.0000 |
+-----+
```

Selain operator kabataku (kali, bagi, tambah dan kurang), juga terdapat operator DIV dan MOD.

Operator DIV adalah pembagian angka bulat. Hasil dari operator DIV mirip seperti pembagian (/), tapi akan selalu berbentuk angka bulat. Jika hasil pembagian tersebut menghasilkan nilai pecahan, angka di belakang koma akan dibuang. Berikut perbandingan antara hasil operator pembagian biasa dengan DIV:

```
SELECT 10 / 3;
```

```
+-----+
| 10 / 3 |
+-----+
| 3.3333 |
+-----+
```

```
SELECT 10 DIV 3;
```

```
+-----+
| 10 DIV 3 |
+-----+
|      3 |
+-----+
```

```
SELECT 100 / 7;
```

```
+-----+
| 100 / 7 |
+-----+
| 14.2857 |
+-----+
```

```
SELECT 100 DIV 7;
```

```
+-----+
| 100 DIV 7 |
+-----+
|      14 |
+-----+
```

Sedangkan **MOD** adalah operator yang menghasilkan nilai sisa hasil bagi. Berikut contohnya:

```
SELECT 10 MOD 4;
```

```
+-----+
| 10 MOD 4 |
+-----+
|      2 |
+-----+
```

```
SELECT 100 MOD 15;
```

```
+-----+
| 100 MOD 15 |
+-----+
|      10 |
+-----+
```

Operasi **10 MOD 4** menghasilkan angka 2 karena angka 10 tidak habis dibagi 4. Angka terdekat yang habis dibagi 4 adalah 8, sehingga terdapat sisa 2.

Operasi `100 MOD 15` menghasilkan angka 10 karena angka 100 tidak habis dibagi 15. Angka terdekat yang habis dibagi 15 adalah 90, sehingga ada terdapat 10.

Di dalam MySQL, operator **MOD** juga memiliki penulisan lain, yakni tanda persen %. Fungsinya sama seperti **MOD**:

```
SELECT 10 % 4;
```

```
+-----+
| 10 % 4 |
+-----+
|      2 |
+-----+
```

```
SELECT 100 % 15;
```

```
+-----+
| 100 % 15 |
+-----+
|      10 |
+-----+
```

Dalam beberapa contoh sebelum ini saya memakai angka langsung ke dalam query **SELECT**. Penggunaan seperti ini sangat sederhana dan tentunya akan lebih mudah jika menggunakan kalkulator.

Namun kita juga bisa menggunakan operator diatas secara “massal” ke dalam kolom tabel. Untuk praktek ini saya mempersiapkan sebuah tabel `nilai_mahasiswa`, dengan query pembuatan tabel sebagai berikut:

```
USE belajar;
-- Database changed
```

```
CREATE TABLE nilai_mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    semester_1 DECIMAL(4,2),
    semester_2 DECIMAL(4,2),
    semester_3 DECIMAL(4,2)
) ENGINE = InnoDB;
-- Query OK, 0 rows affected (0.24 sec)
```

```
INSERT INTO nilai_mahasiswa VALUES
('17090113', 'Riana Putria', 3.12, 2.98, 3.45),
('17140143', 'Rudi Permana', 2.56, 3.14, 3.22),
('17080305', 'Rina Kumala Sari', 3.45, 2.56, 3.67),
('17140119', 'Sandri Fatmala', 2.12, 2.78, 2.56),
('17090308', 'Christine Wijaya', 3.78, 3.23, 3.11);
-- Query OK, 5 rows affected (0.03 sec)
```

```
-- Records: 5  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM nilai_mahasiswa;
```

nim	nama	semester_1	semester_2	semester_3
17080305	Rina Kumala Sari	3.45	2.56	3.67
17090113	Riana Putria	3.12	2.98	3.45
17090308	Christine Wijaya	3.78	3.23	3.11
17140119	Sandri Fatmala	2.12	2.78	2.56
17140143	Rudi Permana	2.56	3.14	3.22

```
-- 5 rows in set (0.00 sec)
```

Tabel `nilai_mahasiswa` berisi nilai mahasiswa untuk 3 semester. Mari kita terapkan penggunaan operator aritmatika untuk menghasilkan informasi dalam bentuk lain.

Contoh pertama, saya ingin menghitung total jumlah nilai dari setiap mahasiswa untuk 3 semester. Artinya nilai akhir adalah hasil dari `semester_1 + semester_2 + semester_3`. Berikut query yang bisa dipakai:

```
SELECT semester_1 + semester_2 + semester_3 FROM nilai_mahasiswa;
```

semester_1 + semester_2 + semester_3
9.68
9.55
10.12
7.46
8.92

Dengan query diatas kita bisa menampilkan nilai total untuk seluruh tabel `nilai_mahasiswa`. Anda juga bisa perhatikan bahwa judul kolom dari tampilan diatas berasal dari operator yang kita tulis.

Agar tampilannya lebih rapi, judul kolom ini bisa diubah dengan query AS:

```
SELECT semester_1 + semester_2 + semester_3 AS 'Total 3 Semester'
FROM nilai_mahasiswa;
```

```
+-----+
| Total 3 Semester |
+-----+
| 9.68 |
| 9.55 |
| 10.12 |
| 7.46 |
| 8.92 |
+-----+
```

Sekarang judul kolom sudah lebih rapi.

Tidak berhenti sampai disana, agar lebih informatif akan lebih bagus jika ditampilkan juga kolom nama mahasiswa:

```
SELECT
    nama,
    (semester_1 + semester_2 + semester_3) AS 'Total 3 Semester'
FROM nilai_mahasiswa;
```

```
+-----+
| nama | Total 3 Semester |
+-----+
| Rina Kumala Sari | 9.68 |
| Riana Putria | 9.55 |
| Christine Wijaya | 10.12 |
| Sandri Fatmala | 7.46 |
| Rudi Permana | 8.92 |
+-----+
```

Disini saya menambahkan tanda kurung untuk menghitung nilai Total 3 Semester agar lebih jelas.

Sebagai latihan, bisakah anda mengubah kolom Total 3 Semester menjadi IPK. IPK merupakan singkatan dari *Indeks Prestasi Kumulatif*, atau bisa dibilang sebagai nilai IP rata-rata yang dimiliki mahasiswa. Nilai IPK ini didapat dari total IP seluruh semester, lalu dibagi dengan banyaknya semester.

Selain itu tampilkan pula nomor nim mahasiswa. Sehingga kolom yang akan tampil adalah: nim, nama, serta IPK.

Baik, berikut query yang saya gunakan:

```
SELECT
nim,
nama,
((semester_1 + semester_2 + semester_3)/3) AS 'IPK'
FROM nilai_mahasiswa;
```

nim	nama	IPK
17080305	Rina Kumala Sari	3.226667
17090113	Riana Putria	3.183333
17090308	Christine Wijaya	3.373333
17140119	Sandri Fatmala	2.486667
17140143	Rudi Permana	2.973333

Dari beberapa percobaan ini kita sudah bisa melihat fungsi dari operator di dalam MySQL. Kita bisa mengolah data dan menampilkannya dalam bentuk kolom. Kolom tersebut belum tentu ada di dalam tabel karena merupakan hasil dari operasi antar kolom. Bahkan bisa saja kolom tersebut adalah hasil dari operasi antar tabel.

Menggunakan *function*, kita bisa mengolah data dengan lebih jauh lagi.

20.2 Cara Pemanggilan Function MySQL

Di dalam programming, **function** adalah kode program yang berfungsi untuk membantu program utama. Sebagian besar function membutuhkan nilai awal (*input*) dan menghasilkan nilai akhir (*output*).

Jika anda sudah pernah mempelajari bahasa pemrograman lain seperti C, C++, PHP atau JavaScript, tentu tidak asing dengan konsep *function*.

Function atau dalam bahasa indonesia disebut juga sebagai **fungsi**, dijalankan dengan format berikut:

```
nama_function(argumen1, argumen2, ...)
```

Setelah menulis `nama_function`, di dalam tanda kurung diisi dengan *argumen*. **Argumen** adalah nilai masukan atau nilai input ke dalam sebuah function. Nilai input ini kemudian diproses oleh function untuk menghasilkan nilai akhir (*output*).

Hampir semua function di dalam MySQL bisa dijalankan dengan perintah **SELECT**. Sebagai contoh, dalam perintah berikut ini saya ingin mencari hasil dari 3^2 dengan menggunakan fungsi `POW()`:

```
SELECT POW(3,2);
+-----+
| POW(3,2) |
+-----+
|      9   |
+-----+
1 row in set (0.03 sec)
```

Fungsi `POW()` berguna untuk mencari hasil pemangkatan. `POW(3,2)` dipakai untuk mencari nilai dari 3^2 , yang hasilnya adalah 9.

Function `POW()` disini membutuhkan 2 buah argumen. Argumen pertama berupa angka yang akan dipangkatkan (dalam contoh ini angka 3), dan argumen kedua berupa nilai pangkat (angka 2).

Di antara nama function dengan tanda kurung argumen sebaiknya tidak boleh dipisah dengan spasi, karena di beberapa versi MySQL bisa menghasilkan error. Penulisan yang disarankan adalah `POW(3,2)`, bukan `POW (3,2)`. Akan tetapi spasi sebagai pemisah antar argumen boleh ditulis, seperti `POW(3, 2)`.

Hasil pemanggilan function dengan query `SELECT` ditampilkan dalam bentuk tabel dengan judul kolom berupa teks dari function yang dijalankan. Dalam contoh sebelumnya, judul kolom adalah `POW(3,2)`.

Agar lebih rapi, kita bisa menggunakan perintah alias query `AS` untuk merubah judul kolom tersebut, seperti contoh berikut:

```
SELECT POW(5,4) AS `Hasil Pangkat`;
+-----+
| Hasil Pangkat |
+-----+
|      625   |
+-----+
1 row in set (0.04 sec)
```

Sekarang judul kolom tertulis `Hasil Pangkat`.

Untuk menampilkan data yang kompleks, nilai argumen dari sebuah function bisa berasal dari hasil pemanggilan function lain, seperti contoh berikut:

```
SELECT SQRT(ABS(-49));
+-----+
| SQRT(ABS(-49)) |
+-----+
|      7   |
+-----+
1 row in set (0.03 sec)
```

Angka 7 didapat dari hasil akar kuadrat nilai absolut -49. Fungsi ABS() digunakan untuk mencari hasil nilai absolut, dan fungsi SQRT() dipakai untuk mencari hasil dari akar kuadrat.

Selain menginput langsung nilai ke dalam function, dalam prakteknya kita akan sering memakai nilai yang ada di dalam tabel. Jika ditulis seperti ini, nilai argumen dari function bisa diganti dengan nama kolom, seperti contoh berikut:

```
SELECT UCASE(nama_prov) FROM belajar.provinsi;
+-----+
| UCASE(nama_prov) |
+-----+
| YOGYAKARTA      |
| KALIMANTAN UTARA|
| ...              |
| ACEH             |
| MALUKU           |
+-----+
34 rows in set (0.03 sec)
```

Dalam contoh ini saya menggunakan function UCASE(nama_prov) untuk menampilkan seluruh nama provinsi dalam huruf besar.



Untuk menyingkat penulisan, jika hasil function berupa satu baris saja, saya akan menampilkannya dengan format berikut:

```
SELECT POW(2,8);
-- 256
```

Hasil dari pemanggilan function POW(2,8) adalah 256. Karakter -- di awal hanya sebagai penanda. Penulisan seperti ini semata-mata agar kode programnya tidak terlalu panjang. Jika anda menjalankannya di cmd Windows, hasilnya tetap berupa tabel seperti contoh sebelum ini.

Dalam bab ini kita telah bahas tentang operator aritmatika serta cara penggunaan function. Selanjutnya kita akan lihat apa saja function bawaan MySQL untuk tipe data **numeric**.

21. MySQL Numeric Function

Numeric function adalah fungsi bawaan MySQL untuk mengolah angka (tipe data *integer*, *decimal*, *float* dan *double*). Function ini akan saya bahas secara berkelompok karena sebagian besar mirip satu sama lain.



Jika anda tertarik mempelajari semua **numeric function** bawaan MySQL, bisa ke [MySQL Documentation: Mathematical Functions¹](#) atau [MySQL Functions²](#).

21.1 Function CEIL(), CEILING(), FLOOR(), ROUND() dan TRUNCATE()

Kelima fungsi ini, yakni `CEIL()`, `CEILING()`, `FLOOR()`, `ROUND()` dan `TRUNCATE()` digunakan untuk pembulatan angka pecahan.

Function `CEIL()` dan `CEILING()` dipakai untuk pembulatan ke atas. Keduanya membutuhkan 1 argumen, yakni angka yang akan dibulatkan. Angka 3.01 akan dibulatkan menjadi 4. Angka 3.99 juga akan dibulatkan menjadi 4.

`CEILING()` merupakan *alias* dari function `CEIL()` yang keduanya berfungsi sama. Berikut contoh penggunaan function `CEIL()` dan `CEILING()`:

```
SELECT CEIL(5.23);
-- 6
```

```
SELECT CEIL(5.83);
-- 6
```

```
SELECT CEIL(5.57);
-- 6
```

```
SELECT CEIL(999.57);
-- 1000
```

```
SELECT CEILING(5.23);
-- 6
```

¹<https://dev.mysql.com/doc/refman/5.7/en/mathematical-functions.html>

²https://www.w3schools.com/sql/sql_ref_mysql.asp

Function FLOOR() digunakan untuk pembulatan ke bawah. Fungsi ini juga membutuhkan 1 argumen, yakni angka yang akan dibulatkan. Angka 3.99 akan dibulatkan menjadi 3. Angka 3.01 juga akan dibulatkan menjadi 3.

Berikut contoh penggunaan function FLOOR():

```
SELECT FLOOR(5.23);
-- 5
```

```
SELECT FLOOR(5.83);
-- 5
```

```
SELECT FLOOR(5.57);
-- 5
```

```
SELECT FLOOR(999.57);
-- 999
```

Function ROUND() digunakan untuk pembulatan ke atas atau ke bawah tergantung nilai pecahan. Jika angka pecahan kurang dari 0.5, maka akan dibulatkan ke bawah. Jika angka pecahan lebih atau sama dengan 0.5 maka akan dibulatkan ke atas.

Function ROUND() bisa dipanggil dengan 1 atau 2 argumen. Argumen pertama adalah angka pecahan yang akan dibulatkan, sedangkan argumen kedua berupa jumlah digit desimal yang diinginkan. Jika argumen kedua tidak ditulis, dianggap 0 yang artinya tidak ada angka pecahan di belakang koma.

Berikut contoh pemanggilan fungsi ROUND() dengan 1 argumen:

```
SELECT ROUND(5.23);
-- 5
```

```
SELECT ROUND(5.83);
-- 6
```

```
SELECT ROUND(5.57);
-- 6
```

```
SELECT ROUND(999.57);
-- 1000
```

Berikut contoh pemanggilan fungsi ROUND() dengan 2 argumen:

```
SELECT ROUND(523.23547,2);
-- 523.24
```

```
SELECT ROUND(523.23547,3);
-- 523.235
```

```
SELECT ROUND(523.23547,7);
-- 523.2354700
```

```
SELECT ROUND(523.23547,0);
-- 523
```

```
SELECT ROUND(523.23547,-1);
-- 520
```

```
SELECT ROUND(523.23547,-2);
-- 500
```

Yang cukup unik adalah, argumen kedua dari fungsi ROUND() bisa diisi dengan angka negatif. Jika ditulis seperti ini, fungsi ROUND() akan mengambil angka bulat paling kanan dan diganti dengan angka 0. Sebagai contoh ROUND(523.23547,-2) akan mengambil 2 angka bulat paling kanan untuk diganti menjadi 0, hasilnya 500.

Function TRUNCATE() digunakan untuk memotong angka desimal sesuai dengan jumlah angka di argumen kedua. Tidak ada pembulatan di fungsi TRUNCATE(), sisa angka akan dibuang sepenuhnya. Fungsi TRUNCATE() juga harus dipanggil dengan 2 argumen.

Berikut contoh penggunaan fungsi TRUNCATE():

```
SELECT TRUNCATE(523.23547,2);
-- 523.23
```

```
SELECT TRUNCATE(523.23547,3);
-- 523.235
```

```
SELECT TRUNCATE(523.23547,7);
-- 523.2354700
```

```
SELECT TRUNCATE(523.23547,0);
-- 523
```

```
SELECT TRUNCATE(523.23547,-1);
-- 520
```

```
SELECT TRUNCATE(523.23547,-2);
-- 500
```

Sebagaimana operator, function MySQL juga bisa dipakai ke dalam kolom tabel. Berikut contoh penggunaannya:

SELECT

```
nama,
semester_1,
CEIL(semester_1) AS `CEIL`,
FLOOR(semester_1) AS `FLOOR`,
ROUND(semester_1) AS `ROUND`
```

```
FROM nilai_mahasiswa;
```

nama	semester_1	CEIL	FLOOR	ROUND
Rina Kumala Sari	3.45	4	3	3
Riana Putria	3.12	4	3	3
Christine Wijaya	3.78	4	3	4
Sandri Fatmala	2.12	3	2	2
Rudi Permana	2.56	3	2	3

Disini saya menampilkan hasil fungsi CEIL(), FLOOR() dan ROUND() untuk kolom `semester_1` dari tabel `nilai_mahasiswa`. Setiap fungsi akan menghasilkan nilai yang berbeda tergantung angka di belakang koma.

Fungsi pembulatan ini cocok dipakai untuk menyederhanakan tampilan angka. Contohnya hasil dari perhitungan IPK:

SELECT

```
nim,
nama,
((semester_1 + semester_2 + semester_3)/3) AS 'IPK'
```

```
FROM nilai_mahasiswa;
```

nim	nama	IPK
17080305	Rina Kumala Sari	3.226667
17090113	Riana Putria	3.183333
17090308	Christine Wijaya	3.373333
17140119	Sandri Fatmala	2.486667
17140143	Rudi Permana	2.973333

Kolom IPK berasal dari perhitungan MySQL yang secara default menghasilkan angka pecahan dengan 6 digit di belakang koma. Angka ini tidak terlalu pas, karena biasanya nilai IPK hanya butuh ketelitian hingga 2 angka di belakang koma. Fungsi ROUND() bisa dipakai untuk situasi seperti ini:

SELECT

```

nim,
nama,
ROUND(((semester_1 + semester_2 + semester_3)/3),2) AS 'IPK'
FROM nilai_mahasiswa;

```

nim	nama	IPK
17080305	Rina Kumala Sari	3.23
17090113	Riana Putria	3.18
17090308	Christine Wijaya	3.37
17140119	Sandri Fatmala	2.49
17140143	Rudi Permana	2.97

Sekarang tampilan IPK sudah terlihat rapi dengan pembulatan 2 angka di belakang tanda koma. Dalam query diatas penulisan fungsi ROUND() tampak sedikit rumit karena input untuk fungsi ini berasal dari hasil perhitungan. Terdapat 3 buah tanda kurung untuk memisahkan operasi mana yang akan dijalankan terlebih dahulu.

21.2 Function SIN(), COS(), TAN(), ASIN(), ACOS(), ATAN() dan COT()

Ketujuh fungsi ini digunakan untuk perhitungan trigonometri.

Fungsi SIN() untuk menghitung nilai *sinus*, COS() untuk *cosinus*, TAN() untuk *tangen*, ASIN() untuk *arc sin / inverse sinus*, ACOS() untuk *arc cos/ inverse cosinus*, ATAN() untuk *arc tan/ inverse tangen*, serta COT() untuk *cotangen*.

Semua fungsi ini membutuhkan 1 argumen dalam bentuk *radian*. Berikut contoh penggunaannya:

```
SELECT SIN(0.5);
-- 0.479425538604203
```

```
SELECT COS(0.5);
-- 0.8775825618903728
```

```
SELECT TAN(0.5);
-- 0.5463024898437905
```

```
SELECT ASIN(0.5);
-- 0.5235987755982989
```

```
SELECT ACOS(0.5);
-- 1.0471975511965979
```

```
SELECT ATAN(0.5);
-- 0.4636476090008061
```

```
SELECT COT(0.5);
-- 1.830487721712452
```

Fungsi-fungsi ini baru terpakai untuk perhitungan matematika yang melibatkan rumus trigonometri.

21.3 Function PI(), RADIANS() dan DEGREES()

Ketiga fungsi ini berkaitan dengan perhitungan sudut dan konversi antara *radian* dengan derajat.

Function PI() tidak membutuhkan argumen dan akan mengembalikan nilai konstanta pi matematika, yakni 3.14. Function RADIANS() digunakan untuk mengkonversi nilai derajat ke dalam bentuk *radian*, sedangkan function DEGREES() untuk mengkonversi nilai *radian* ke dalam derajat.

Berikut contoh penggunaan fungsi-fungsi ini:

```
SELECT PI();
-- 3.141593
```

```
SELECT RADIANS(180);
-- 3.141592653589793
```

```
SELECT DEGREES(2.25);
-- 128.91550390443524
```

```
SELECT DEGREES(PI());
-- 180
```

Sebagai pengingat, dalam matematika $1 \text{ pi radian} = 3.14 \text{ radian} = 180 \text{ derajat}$. Oleh karena itulah hasil dari DEGREES(PI()) = 180.

21.4 Function LOG(), LOG2(), LOG10() dan LN()

Semua fungsi ini berkaitan dengan perhitungan logaritma.

Function LOG() digunakan untuk menghitung hasil logaritma. Fungsi ini bisa ditulis dalam 1 atau 2 argumen. Jika hanya diisi dengan 1 argumen, fungsi LOG() akan mengembalikan nilai logaritma natural (e). Berikut contohnya:

```
SELECT LOG(1);
-- 0

SELECT LOG(2);
-- 0.6931471805599453

SELECT LOG(10);
-- 2.302585092994046
```

Jika fungsi LOG() diisi dengan 2 argumen, urutan argumennya akan berubah. Argumen pertama bertujuan untuk mengatur basis logaritma, sedangkan argumen kedua berisi angka logaritma yang akan dihitung. Berikut contohnya:

```
SELECT LOG(2, 128);
-- 7

SELECT LOG(8, 64);
-- 2

SELECT LOG(10, 10000);
-- 4
```

Fungsi LOG(2, 128) artinya kita mencari nilai logaritma basis 2 dari 128, yang hasilnya adalah 7. Dengan kata lain $2^7 = 128$.

Fungsi LOG(8, 64) artinya kita mencari nilai logaritma basis 8 dari 64, yang hasilnya adalah 2. Dimana $8^2 = 64$.

Fungsi LOG(10, 10000) artinya kita mencari nilai logaritma basis 10 dari 10000, yang hasilnya adalah 4. Dimana $10^4 = 10000$.

Function LOG2() dan LOG10() merupakan penulisan singkat untuk algoritma berbasis 2, dan 10. Berikut contohnya:

```
SELECT LOG2(128);
-- 7

SELECT LOG10(10000);
-- 4
```

Function LN() merupakan fungsi khusus untuk mencari logaritma natural (e). Artinya, fungsi ini sama dengan fungsi LOG() dengan 1 argumen:

```
SELECT LN(5);
-- 1.6094379124341003
```

```
SELECT LOG(5);
-- 1.6094379124341003
```

21.5 Function POW(), POWER() dan EXP()

Ketiga fungsi ini digunakan untuk mencari nilai hasil pemangkatan.

Function POW() membutuhkan 2 buah argumen. Argumen pertama adalah angka yang akan dipangkatkan, sedangkan argumen kedua berupa nilai pangkat. POW(2,3) artinya sama dengan 2^3 , dan POW(5,3) sama dengan 5^3 . Function POWER() merupakan *alias* dari POW(), dimana akan berfungsi sama.

Berikut contoh penggunaan fungsi POW() dan POWER():

```
SELECT POW(2,3);
-- 8
```

```
SELECT POW(5,3);
-- 125
```

```
SELECT POWER(10,4);
-- 10000
```

Function EXP() digunakan untuk mencari nilai pangkat dari bilangan natural (e). Fungsi ini butuh 1 argumen, yakni nilai yang akan dicari pangkat e-nya. EXP(2) artinya 2^e , dan EXP(5) sama dengan 5^e . Berikut contoh penggunaannya:

```
SELECT EXP(2);
-- 7.38905609893065
```

```
SELECT EXP(5);
-- 148.4131591025766
```

Fungsi EXP() ini merupakan kebalikan dari logaritma natural LN():

```
SELECT EXP(2);
-- 7.38905609893065
```

```
SELECT LN(7.38905609893065);
-- 2
```

21.6 Function RAND()

Function RAND() berguna untuk menghasilkan angka acak. Fungsi ini bisa dipanggil tanpa argumen dan akan menghasilkan angka acak dengan jangkauan 0 hingga 1, dimana angka 1 itu sendiri tidak termasuk ($0 \leq \text{angka_acak} < 1$).

Berikut contoh penggunaan fungsi RAND():

```
SELECT RAND();
-- 0.2497786202000255
```

```
SELECT RAND();
-- 0.6431679526745975
```

```
SELECT RAND();
-- 0.4665039335065558
```

Sesuai dengan kegunaannya, setiap kali fungsi RAND() dijalankan, akan menghasilkan angka acak dalam rentang 0 - 1.

Fungsi RAND() sebenarnya bisa diisi dengan 1 argumen opsional berupa angka. Angka ini berfungsi sebagai *seed* untuk menghasilkan angka acak.

Dalam teori angka acak, *seed* (babit) adalah nilai input untuk membuat suatu “keacakan” (*randomness*). Seed ini berguna untuk membuat angka yang “lebih acak”, meskipun dalam prakteknya tidak akan terlalu terlihat. Jika anda tertarik untuk mempelajari lebih lanjut, bisa membaca teori seputar [Random seed³](#) dan [Pseudorandom number generator⁴](#).

Berikut contoh pemanggilan fungsi RAND() dengan argumen:

```
SELECT RAND(3);
-- 0.9057697559760601
```

```
SELECT RAND(10);
-- 0.6570515219653505
```

```
SELECT RAND(188);
-- 0.18964495900053993
```

Dapat terlihat angka acak yang dihasilkan tetap antara 0 - 1, sama seperti pemanggilan fungsi RAND() tanpa argumen.

Sekarang, bagaimana caranya untuk menghasilkan angka acak yang lebih dari 1? misalnya untuk membuat angka bulat antara 0 - 9? Kita bisa mengalikan angka hasil RAND() dengan 10 lalu dibulatkan menggunakan fungsi FLOOR().

Berikut contoh penggunaannya:

³https://en.wikipedia.org/wiki/Random_seed

⁴https://en.wikipedia.org/wiki/Pseudorandom_number_generator

```
SELECT FLOOR(RAND() * 10);
-- 9
```

```
SELECT FLOOR(RAND() * 10);
-- 6
```

```
SELECT FLOOR(RAND() * 10);
-- 1
```

```
SELECT FLOOR(RAND() * 10);
-- 3
```

Bagaimana untuk menghasilkan angka acak dalam rentang 100 - 200? Kita tinggal memodifikasi fungsi diatas menjadi berikut:

```
SELECT FLOOR(100 + (RAND() * 100));
-- 106
```

```
SELECT FLOOR(1 + (RAND() * 9));
-- 147
```

```
SELECT FLOOR(1 + (RAND() * 9));
-- 120
```

Disini rumus yang digunakan adalah `FLOOR(i + RAND() * (j - i))`, dimana **i** = range awal, dan **j** sebagai range akhir. Misalnya untuk membuat fungsi angka acak antara 50 - 75, fungsi yang dipakai adalah `FLOOR(50 + RAND() * (75 - 50))`, atau `FLOOR(50 + RAND() * (25))`.

Sebagai contoh kasus lain, saya ingin menginput data baru ke dalam tabel `nilai_mahasiswa`, tapi nilai IP akan di hasilkan secara acak menggunakan fungsi `RAND()`.

Berikut isi dari tabel `nilai_mahasiswa` saat ini:

```
SELECT * FROM nilai_mahasiswa;
+-----+-----+-----+-----+
| nim      | nama           | semester_1 | semester_2 | semester_3 |
+-----+-----+-----+-----+
| 17080305 | Rina Kumala Sari |     3.45 |     2.56 |     3.67 |
| 17090113 | Riana Putria   |     3.12 |     2.98 |     3.45 |
| ...       | ...             |     ...    |     ...    |     ...    |
+-----+-----+-----+-----+
```

Nilai untuk semester terdiri dari angka 0.00 hingga 4.00. Agar terlihat nyata, batas nilai saya naikkan menjadi 2.00 hingga 4.00, karena mahasiswa dengan IP dibawah 2.00 relatif jarang (kecuali sering bolos kuliah).

Bisakah anda merancang fungsi untuk menghasilkan nilai acak ini? Silahkan di coba sebentar. Karena angka acak yang dihasilkan memiliki nilai desimal, akan lebih mudah menggunakan fungsi ROUND() sebagai pengganti FLOOR(). Fungsi ROUND() bisa diinput dengan argumen kedua yang dipakai untuk menentukan jumlah digit di belakang koma.

Baik, berikut perpaduan fungsi ROUND() dan RAND() untuk menghasilkan angka acak dengan 2 tempat desimal:

```
SELECT ROUND((2 + (RAND() * 2)), 2);
-- 2.97
```

```
SELECT ROUND((2 + (RAND() * 2)), 2);
-- 3.12
```

```
SELECT ROUND((2 + (RAND() * 2)), 2);
-- 3.60
```

Selanjutnya kita tinggal menggunakan rumus ini ke dalam query INSERT untuk menginput data baru:

```
INSERT INTO nilai_mahasiswa VALUES (
    '17080225',
    'Husli Khairan',
    ROUND((2 + (RAND() * 2)), 2),
    ROUND((2 + (RAND() * 2)), 2),
    ROUND((2 + (RAND() * 2)), 2)
);
```

```
SELECT * FROM nilai_mahasiswa;
```

nim	nama	semester_1	semester_2	semester_3
17080225	Husli Khairan	2.50	2.05	2.75
17080305	Rina Kumala Sari	3.45	2.56	3.67
17090113	Riana Putria	3.12	2.98	3.45
17090308	Christine Wijaya	3.78	3.23	3.11
17140119	Sandri Fatmala	2.12	2.78	2.56
17140143	Rudi Permana	2.56	3.14	3.22

Terlihat di baris pertama, nilai semester untuk Husli Khairan sudah berisi angka 2.50, 2.05 dan 2.75 yang semuanya berasal dari fungsi RAND().



Apabila anda teliti, mungkin muncul pertanyaan kenapa di awal saya menggunakan fungsi FLOOR() untuk membulatkan angka hasil RAND()? Kenapa tidak ROUND() saja, atau CEIL()?

Ini berkaitan dengan distribusi bilangan acak yang dihasilkan. Jika menggunakan fungsi ROUND() atau CEIL(), distribusi pembulatan menjadi kurang merata. Teori mengenai hal ini pernah saya bahas di buku **JavaScript Uncover**, atau bisa dibaca-baca kesini: [Random numbers and floor vs round function⁵](#).

21.7 Function SQRT()

Fungsi SQRT() digunakan untuk mencari akar kuadrat (*square root*). Fungsi ini membutuhkan 1 argumen berupa angka yang akan dicari nilai akar kuadratnya. Berikut contoh penggunaan fungsi ini:

```
SELECT SQRT(64);
-- 8
```

```
SELECT SQRT(121);
-- 11
```

Mencari nilai akar kuadrat sebenarnya juga bisa dari fungsi POW(). Caranya, input nilai pecahan sebagai argumen kedua untuk fungsi ini:

```
SELECT POW(64,1/2);
-- 8
```

```
SELECT POW(121,1/2);
-- 11
```

```
SELECT POW(625,1/4);
-- 5
```

Pada contoh terakhir fungsi POW(625,1/4) digunakan untuk mencari nilai akar pangkat empat dari angka 625. Hasil ini tidak bisa didapat jika menggunakan fungsi SQRT() yang hanya khusus untuk mencari akar pangkat dua.

21.8 Function ABS()

Function ABS() digunakan untuk mencari nilai absolut. Fungsi ini membutuhkan 1 argumen. Jika argumen tersebut berupa angka positif, angka akan langsung dikembalikan. Jika argumen tersebut berupa angka negatif, akan diubah menjadi positif.

Berikut contoh penggunaannya:

⁵ <https://stackoverflow.com/questions/7377735/random-numbers-and-floor-vs-round-function>

```
SELECT ABS(7);
-- 7

SELECT ABS(-9);
-- 9

SELECT ABS(-1987);
-- 1987

SELECT ABS(0);
-- 0
```

21.9 Function COUNT(), MIN(), MAX(), AVG(), SUM() dan STD()

Keenam fungsi ini termasuk ke dalam **Aggregate Function**, yakni fungsi yang dipakai untuk isi tabel dalam bentuk kelompok. Umumnya fungsi-fungsi ini digunakan dengan query GROUP BY.

Salah satu ciri khas dari *aggregate function* adalah kita tidak bisa menggunakannya untuk data yang diinput langsung. Data tersebut harus diambil dari hasil kolom tabel. Seluruh fungsi ini membutuhkan satu argumen berupa kolom tabel yang akan dicari nilainya.

Fungsi COUNT() dipakai untuk mengetahui jumlah baris yang ada di dalam tabel. Berikut contoh penggunaannya:

```
SELECT COUNT(nama) FROM nilai_mahasiswa;
+-----+
| COUNT(nama) |
+-----+
|          6 |
+-----+
```

Fungsi MIN() dan MAX() dipakai untuk mencari berapa nilai terendah dan tertinggi yang ada di dalam kolom, sedangkan fungsi AVG() dipakai untuk mencari nilai rata-rata (*average*) dari seluruh kolom. Berikut praktek dari ketiga fungsi ini:

```
SELECT MIN(semester_1), MAX(semester_1), AVG(semester_1) FROM nilai_mahasiswa;
+-----+-----+-----+
| MIN(semester_1) | MAX(semester_1) | AVG(semester_1) |
+-----+-----+-----+
|      2.12 |       3.78 |   2.921667 |
+-----+-----+-----+
```

Fungsi SUM() digunakan untuk mencari total penjumlahan dari seluruh nilai kolom. Sedangkan fungsi STD() berguna untuk menghitung total standar deviasi. Fungsi STD() ini baru terpakai jika kita ingin menampilkan hasil perhitungan statistika.

Berikut contoh penggunaan kedua fungsi ini:

```
SELECT SUM(semester_1), STD(semester_1) FROM nilai_mahasiswa;
```

```
+-----+-----+
| SUM(semester_1) | STD(semester_1) |
+-----+-----+
|      17.53 |     0.578285 |
+-----+-----+
```

Dalam bab ini kita sudah membahas beberapa fungsi bawaan MySQL untuk memanipulasi tipe data **numeric**. Berikutnya akan dibahas tentang fungsi bawaan MySQL untuk tipe data **string**.

22. MySQL String Function

String function adalah fungsi MySQL untuk memanipulasi tipe data **string**, termasuk diantaranya CHAR, VARCHAR dan TEXT. Sama seperti materi dalam *numeric function*, saya akan membahas fungsi-fungsi string secara berkelompok.



Jika anda tertarik mempelajari semua **string function** bawaan MySQL, bisa ke [MySQL Documentation: String Functions¹](#) atau [MySQL Functions²](#).

22.1 Function FORMAT()

Fungsi **FORMAT()** digunakan untuk men-format tampilan angka dengan tambahan karakter pemisah ribuan, yakni tanda koma setiap 3 digit. Misalnya angka 1553000 akan diformat menjadi 1,553,000.

Fungsi ini sebenarnya juga termasuk ke dalam *numeric function* karena nilai input fungsi **FORMAT()** berupa tipe data *numeric*. Akan tetapi hasil akhir dari fungsi **FORMAT()** ini adalah tipe data *string*.

Fungsi **FORMAT()** membutuhkan 2 atau 3 argumen dimana argumen ketiga bersifat opsional. Argumen pertama diisi dengan angka yang akan di format, sedangkan argumen kedua berupa jumlah digit desimal. Digit desimal ini akan dibulatkan jika terdapat pemotongan angka.

Berikut contoh penggunaan dari fungsi **FORMAT()**:

```
SELECT FORMAT(1553000.156123, 2);
-- 1,553,000.16
```

```
SELECT FORMAT(1553000.156123, 4);
-- 1,553,000.1561
```

Pada contoh pertama, terjadi pembulatan ke atas karena digit pecahan ketiga berupa angka 6. Sedangkan pada contoh kedua akan dibulatkan ke bawah karena digit kelima berupa angka 2.

Secara default, MySQL menggunakan format penulisan angka di Amerika, yakni tanda koma sebagai pemisah ribuan dan tanda titik sebagai pemisah pecahan. Bagaimana cara untuk menukar tanda ini? Karena di Indonesia kita menggunakan tanda titik sebagai pemisah ribuan dan tanda koma sebagai pemisah pecahan.

¹<https://dev.mysql.com/doc/refman/5.7/en/string-functions.html>

²https://www.w3schools.com/sql/sql_ref_mysql.asp

Fungsi FORMAT() memiliki argumen ketiga yang bersifat opsional. Argumen ketiga ini bisa diisi dengan format kode bahasa ISO ([ISO Language Code³](#)). Secara bawaan, kode bahasa yang dipakai adalah ‘en_US’ yang berarti ‘bahasa inggris_amerika serikat’ atau ‘english_united states’. Kedua query berikut akan menghasilkan nilai yang sama:

```
SELECT FORMAT(1553000.156123, 2);
-- 1,553,000.16

SELECT FORMAT(1553000.156123, 2, 'en_US');
-- 1,553,000.16
```

Kode ISO untuk Bahasa Indonesia adalah ‘id_ID’, mari kita coba:

```
SELECT FORMAT(1553000.156123, 2, 'id_ID');
-- 1.553.000,16
```

Sekarang, angka akan di format sesuai dengan kaedah penulisan Bahasa Indonesia, dimana tanda titik dipakai sebagai pemisah ribuan dan tanda koma sebagai pemisah pecahan.

22.2 Function LOWER(), LCASE(), UPPER() dan UCASE()

Keempat fungsi ini digunakan untuk mengubah jenis huruf (*case*). Fungsi LOWER() dan LCASE() untuk mengubah string ke huruf kecil, serta fungsi UPPER() dan UCASE() untuk mengubah string ke huruf besar. Semua fungsi membutuhkan 1 argumen sebagai string yang akan diubah.

Berikut contoh penggunaan dari keempat fungsi ini:

```
SELECT LOWER('Belajar MySQL');
-- belajar mysql

SELECT LCASE('Belajar MySQL');
-- belajar mysql

SELECT UPPER('Belajar MySQL');
-- BELAJAR MYSQL

SELECT UCASE('Belajar MySQL');
-- BELAJAR MYSQL
```

Sebagai contoh kasus, saya ingin menampilkan nama mahasiswa dari tabel nilai_mahasiswa dengan menggunakan fungsi LOWER() dan UPPER():

³<http://www.lingoes.net/en/translator/langcode.htm>

```
SELECT LOWER(nama), UPPER(nama) FROM nilai_mahasiswa;
```

LOWER(nama)	UPPER(nama)
husli khairan	HUSLI KHAIRAN
rina kumala sari	RINA KUMALA SARI
riana putria	RIANA PUTRIA
christine wijaya	CHRISTINE WIJAYA
sandri fatmala	SANDRI FATMALA
rudi permana	RUDI PERMANA

22.3 Function CHAR_LENGTH(), CHARACTER_LENGTH(), LENGTH() dan BIT_LENGTH()

Fungsi-fungsi ini digunakan untuk menghitung panjang string, namun dengan sedikit perbedaan. Keempatnya membutuhkan 1 argumen berupa string yang akan dihitung.

Fungsi CHAR_LENGTH() dan CHARACTER_LENGTH() sama-sama digunakan untuk menghitung jumlah karakter dari sebuah string. Berikut contoh penggunaan kedua fungsi ini:

```
SELECT CHAR_LENGTH('Belajar MySQL');
--13
```

```
SELECT CHARACTER_LENGTH('Belajar MySQL');
--13
```

Angka 13 berarti string 'Belajar MySQL' terdiri dari 13 karakter. Ingat, spasi juga termasuk sebuah karakter.

Fungsi LENGTH() digunakan untuk menghitung jumlah byte dalam sebuah string. Untuk karakter set ASCII atau Latin, 1 karakter berupa 1 byte (8 bit). Artinya jika character set yang digunakan adalah ASCII atau Latin, fungsi LENGTH() akan menghasilkan nilai yang sama dengan fungsi CHAR_LENGTH() maupun CHARACTER_LENGTH(), seperti contoh berikut:

```
SELECT LENGTH('Belajar MySQL');
--13
```

Namun jika yang digunakan adalah character set UTF-8 atau diatasnya, 1 karakter bisa saja terdiri dari 2 byte atau lebih. Dalam kasus ini fungsi LENGTH() bisa mengembalikan hasil yang berbeda dari fungsi CHAR_LENGTH() maupun CHARACTER_LENGTH().

Fungsi CHAR_LENGTH() dan CHARACTER_LENGTH() akan menghitung berapa banyak jumlah karakter, terlepas dari berapa byte yang dipakai untuk 1 karakter. Karena cmd Windows tidak mendukung penulisan karakter yang lebih dari 1 byte, saya tidak bisa memperlihatkan perbedaan ini.

Sesuai dengan namanya, fungsi BIT_LENGTH() akan mengembalikan jumlah bit dalam sebuah string. Berikut contoh penggunaannya:

```
SELECT BIT_LENGTH('Belajar MySQL');
-- 104
```

Angka 104 didapat dari $13 * 8$. String 'Belajar MySQL' terdiri dari 13 karakter, karena **Latin-1** merupakan character set bawaan MySQL / MariaDB, maka 1 karakter terdiri dari 8 bit, sehingga $13 * 8 = 104$.

Berikut contoh penggunaan fungsi ini ke dalam tabel nilai_mahasiswa:

```
SELECT nama, CHAR_LENGTH(nama), BIT_LENGTH(nama) FROM nilai_mahasiswa;
+-----+-----+-----+
| nama           | CHAR_LENGTH(nama) | BIT_LENGTH(nama) |
+-----+-----+-----+
| Husli Khairan |          13 |        104 |
| Rina Kumala Sari |         16 |        128 |
| Riana Putria |         12 |         96 |
| Christine Wijaya |         16 |        128 |
| Sandri Fatmala |         14 |        112 |
| Rudi Permana |         12 |         96 |
+-----+-----+-----+
```

Disini saya menampilkan jumlah karakter yang dipakai untuk setiap nama mahasiswa serta jumlah bit untuk nama tersebut.

22.4 Function CONCAT() dan CONCAT_WS()

Fungsi CONCAT() dan CONCAT_WS() digunakan untuk operasi penyambungan string. Bedanya, di dalam fungsi CONCAT_WS() kita bisa menginput karakter pembatas antara string yang disambung.

Fungsi CONCAT() bisa diisi dengan banyak argumen. Setiap argumen merupakan string yang akan disambung. Berikut contoh penggunaannya:

```
SELECT CONCAT('Belajar', 'MySQL');
-- BelajarMySQL

SELECT CONCAT('Belajar', 'MySQL', 'di', 'DuniaIlkom');
-- BelajarMySQLdiDuniaIlkom
```

Untuk fungsi CONCAT_WS(), argumen pertama dari fungsi ini diisi dengan string pemisah. Berikut contoh penggunaannya:

```
SELECT CONCAT_WS('-', 'Belajar', 'MySQL', 'di', 'DuniaIlkom');
-- Belajar-MySQL-di-DuniaIlkom

SELECT CONCAT_WS(' ', 'Belajar', 'MySQL', 'di', 'DuniaIlkom');
-- Belajar MySQL di DuniaIlkom
```

Sebagai contoh kasus, saya ingin menampilkan 1 kolom yang nilainya terdiri dari dari nim # nama. Fungsi CONCAT_WS() bisa digunakan untuk keperluan ini:

```
SELECT CONCAT_WS(' # ', nim, nama) FROM nilai_mahasiswa;
+-----+
| CONCAT_WS(' # ', nim, nama) |
+-----+
| 17080225 # Husli Khairan   |
| 17080305 # Rina Kumala Sari |
| 17090113 # Riana Putria    |
| 17090308 # Christine Wijaya |
| 17140119 # Sandri Fatmala  |
| 17140143 # Rudi Permana    |
+-----+
```

22.5 Function LPAD() dan RPAD()

Fungsi LPAD() dan RPAD() dipakai untuk membuat efek *padding*, yakni menambah beberapa karakter di sisi kiri atau kanan sampai panjang string tersebut sesuai dengan yang sudah ditetapkan.

Fungsi LPAD() untuk menambah karakter di sebelah kiri (*left*), sedangkan fungsi RPAD() untuk menambah karakter di sebelah kanan (*right*).

Kedua fungsi ini membutuhkan 3 buah argumen. Argumen pertama diisi dengan sting awal. Argumen kedua berupa panjang string akhir yang diingkan. Serta argumen ketiga berupa karakter penambah.

Berikut contoh penggunaan dari fungsi LPAD() dan RPAD():

```
SELECT LPAD('MySQL',8,'#');
-- ###MySQL
```

```
SELECT LPAD('123',8,'0');
-- 00000123
```

```
SELECT RPAD('MySQL',8,'#');
-- MySQL###
```

```
SELECT RPAD('123',8,'0');
-- 12300000
```

Fungsi LPAD('MySQL',8,'#') artinya, tambahkan beberapa karakter '#' di sisi kiri string 'MySQL' sampai panjang string tersebut mencapai 8 karakter. Karena string 'MySQL' terdiri dari 5 karakter, maka akan ditambah 3 buah karakter '#' supaya panjangnya menjadi 8.

Fungsi LPAD('123',8,'0') artinya, tambahkan beberapa karakter '0' di sisi kiri string '123' sampai panjang string tersebut 8 karakter. Karena string '123' terdiri dari 3 karakter, maka akan ditambah 5 buah karakter '0' supaya panjangnya menjadi 8 .

Hal yang sama juga berlaku untuk fungsi RPAD(), hanya saja posisi penambahan karakter berada di sisi kanan.

Yang cukup unik, fungsi LPAD() dan RPAD() bisa "memakan" string asal jika panjang string hasil lebih pendek daripada string awal. Berikut contoh kasusnya:

```
SELECT LPAD('MySQL',3,'#');
-- MyS
```

```
SELECT RPAD('MySQL',3,'#');
-- MyS
```

Karena hasil akhir yang diminta adalah 3 karakter, maka karakter awal 'MySQL' akan dipotong.

Berikut contoh penggunaan fungsi LPAD() dan RPAD() untuk menampilkan tabel nilai_mahasiswa:

```
SELECT RPAD(nama,20,'-'), LPAD(semester_1,6,'0') FROM nilai_mahasiswa;
+-----+-----+
| RPAD(nama,20,'-') | LPAD(semester_1,6,'0') |
+-----+-----+
| Husli Khairan---- / 002.50 | /
| Rina Kumala Sari---- / 003.45 | /
| Riana Putria----- / 003.12 | /
| Christine Wijaya--- / 003.78 | /
| Sandri Fatmala---- / 002.12 | /
| Rudi Permana----- / 002.56 | /
+-----+-----+
```

Disini saya menampilkan kolom nama dan semester_1 dengan tambahan string.

22.6 Function LTRIM(), RTRIM() dan TRIM()

Fungsi LTRIM(), RTRIM() dan TRIM() digunakan untuk menghapus karakter spasi di sisi kiri, kanan atau kedua sisi. Fungsi-fungsi ini biasanya dipakai untuk membersihkan string dari penambahan spasi yang tidak disengaja.

Berikut contoh penggunaannya:

```
SELECT LTRIM(' MySQL ');
-- MySQL
```

```
SELECT RTRIM(' MySQL ');
-- MySQL
```

```
SELECT TRIM(' MySQL ');
-- MySQL
```

String input dari ketiga contoh diatas adalah ' MySQL ', dimana ada beberapa spasi sebelum dan sesudah string. Fungsi LTRIM() akan menghapus spasi di sisi kiri string (*left*). Fungsi RTRIM() akan menghapus spasi di sisi kiri *kanan* (*right*). Dan fungsi TRIM() akan menghapus spasi di sisi kiri dan kanan string.

Efek dari kode diatas memang tidak terlalu terlihat. Dengan bantuan fungsi LENGTH(), hasilnya bisa lebih jelas:

```
SELECT LENGTH(' MySQL ');
-- 10
```

```
SELECT LENGTH(LTRIM(' MySQL '));
-- 8
```

```
SELECT LENGTH(RTRIM(' MySQL '));
-- 7
```

```
SELECT LENGTH(TRIM(' MySQL '));
-- 5
```

Fungsi LENGTH() akan menampilkan panjang string saat ini, termasuk spasi. Penghapusan spasi akan mengubah panjang total dari sebuah string.

Inputan untuk password biasanya butuh fungsi TRIM(), karena penambahan satu spasi saja bisa menyebabkan masalah karena nilai password bisa dianggap tidak cocok.

Misalkan saya punya tabel `user` yang berisi daftar `nama` dan `password`. Saat proses registrasi, bisa saja ada yang tidak sengaja menginput spasi di akhir nama atau password. Akibatnya, saat proses validasi user tersebut tidak cocok. String 'budi ' (dengan tambahan 1 spasi di belakang), tidak sama dengan 'budi' (tanpa tambahan spasi).

Fungsi TRIM() ini bisa ditulis saat proses input ke database:

```
INSERT INTO user (nama, password) VALUES (TRIM('budi '), TRIM('rahasia '));
```

Dengan fungsi TRIM(), tambahan spasi di awal dan akhir string akan dihapus sebelum diinput ke dalam tabel.

Fungsi TRIM() sebenarnya juga memiliki fitur tambahan, yakni menghapus karakter lain di awal dan akhir string (tidak hanya spasi). Untuk fitur tambahan ini, butuh penulisan khusus seperti contoh berikut:

```
SELECT TRIM(LEADING '+' FROM '++++MySQL++');
-- MySQL++
```

```
SELECT TRIM(TRAILING '+' FROM '++++MySQL++');
-- ++++MySQL
```

```
SELECT TRIM(BOTH '+' FROM '++++MySQL++');
-- MySQL
```

Format penulisannya adalah:

```
TRIM(LEADING atau TRAILING atau BOTH 'karakter' FROM 'string')
```

LEADING akan menghapus karakter di awal string, TRAILING untuk akhir string, dan BOTH untuk awal dan akhir string.

22.7 Function LOCATE()

Fungsi LOCATE() berguna untuk mencari posisi suatu karakter atau string di dalam string lain.

Fungsi ini bisa diisi dengan 2 atau 3 argumen. Argumen pertama berupa karakter atau string yang akan dicari. Argumen kedua diisi dengan string sumber. Argumen ketiga bersifat opsional, berupa posisi dimulainya pencarian. Jika argumen ketiga ini tidak diisi, pencarian akan dilakukan dari awal string.

Hasil akhir dari fungsi LOCATE() adalah posisi karakter di dalam string. Berbeda dengan bahasa pemrograman seperti PHP atau JavaScript, karakter pertama string mulai dari posisi 1, bukan 0.

Berikut contoh penggunaan dari fungsi LOCATE():

```
SELECT LOCATE('a', 'Belajar MySQL di DuniaIlkom');
-- 4

SELECT LOCATE('a', 'Belajar MySQL di DuniaIlkom', 5);
-- 6

SELECT LOCATE('a', 'Belajar MySQL di DuniaIlkom', 7);
-- 22
```

Dalam perintah pertama, hasilnya adalah 4. Artinya huruf a ditemukan di posisi ke-4 dari string 'Belajar MySQL di DuniaIlkom'.

Untuk perintah kedua dan ketiga, saya menginput argumen ketiga. Karena diisi dengan angka 5 dan 7, pencarian akan dilakukan dari posisi ini. Huruf a ditemukan lagi di posisi 6 dan 22.

Fungsi LOCATE() bersifat *case insensitive*, yang artinya tidak membedakan huruf besar dan huruf kecil. Jika karakter yang dicari tidak ditemukan, fungsi ini akan mengembalikan nilai 0:

```
SELECT LOCATE('b', 'Belajar MySQL di DuniaIlkom');
-- 1

SELECT LOCATE('b', 'Belajar MySQL di DuniaIlkom', 2);
-- 0
```

Sebagai contoh praktik, saya ingin mencari posisi huruf a dari seluruh nama mahasiswa di tabel nilai_mahasiswa:

```
SELECT nama, LOCATE('a',nama) FROM nilai_mahasiswa;
+-----+-----+
| nama           | LOCATE('a',nama) |
+-----+-----+
| Husli Khairan |         9 |
| Rina Kumala Sari |        4 |
| Riana Putria |        3 |
| Christine Wijaya |      14 |
| Sandri Fatmala |        2 |
| Rudi Permana |       10 |
+-----+-----+
```

22.8 Function SUBSTR() dan SUBSTRING()

Kedua fungsi ini bertujuan untuk memotong string atau mengambil sebagian string (membuat *sub-string*). Fungsi SUBSTR() dan SUBSTRING() membutuhkan 2 atau 3 argumen.

Argumen pertama berupa string sumber. Argumen kedua adalah posisi awal pengambilan. Argumen ketiga bersifat opsional berupa jumlah karakter yang akan diambil. Jika argumen ketiga tidak diisi, pengambilan string akan dilakukan hingga akhir.

Berikut contoh penggunaannya:

```
SELECT SUBSTR('Belajar MySQL',4);
-- ajar MySQL

SELECT SUBSTRING('Belajar MySQL',4);
-- ajar MySQL

SELECT SUBSTR('Belajar MySQL',4,4);
-- ajar
```

Sama seperti fungsi LOCATE(), posisi karakter string dimulai dari 1 untuk karakter pertama, bukan 0. Fungsi SUBSTR('Belajar MySQL', 4) artinya ambil string 'Belajar MySQL' mulai dari karakter ke-4 hingga akhir, dimana hasilnya adalah ajar MySQL.

Jika ditulis sebagai SUBSTR('Belajar MySQL', 4, 4), artinya ambil string 'Belajar MySQL' mulai dari karakter ke-4, ambil sebanyak 4 karakter setelah itu. hasilnya adalah ajar.

Argumen kedua juga bisa diisi dengan angka negatif. Akibatnya, pengambilan string akan dihitung dari akhir string, seperti contoh berikut:

```
SELECT SUBSTR('Belajar MySQL', -5);
-- MySQL

SELECT SUBSTR('Belajar MySQL', -5, 2);
-- My
```

Sebagai contoh praktek dari tabel nilai_mahasiswa, saya ingin menampilkan 5 karakter awal dari setiap nama mahasiswa:

```
SELECT SUBSTR(nama,1,5) FROM nilai_mahasiswa;
+-----+
| SUBSTR(nama,1,5) |
+-----+
| Husli      |
| Rina       |
| Riana      |
| Chris      |
| Sandr      |
| Rudi       |
+-----+
```

22.9 Function LEFT() dan RIGHT()

Function LEFT() dan RIGHT() bisa dibilang sebagai versi sederhana dari SUBSTR(). Kedua fungsi ini akan mengambil sebagian string dari posisi kiri untuk LEFT() dan posisi kanan untuk RIGHT().

Kedua fungsi ini membutuhkan 2 buah argumen. Argumen pertama berupa string sumber. Argumen kedua adalah jumlah karakter yang akan diambil.

Berikut contoh penggunaannya:

```
SELECT LEFT('Belajar MySQL',5);
-- Belaj

SELECT RIGHT('Belajar MySQL',5);
-- MySQL
```

22.10 Function INSERT()

Function INSERT() digunakan untuk menginput sebuah karakter atau string lain ke dalam string saat ini. Selain itu fungsi INSERT() juga bisa dipakai untuk mengganti beberapa karakter di dalam string asal.

Fungsi ini membutuhkan 4 argumen. Argumen pertama untuk input string asal. Argumen kedua berupa posisi awal penambahan. Argumen ketiga berupa jumlah karakter string asal yang ingin dihapus. Dan argumen keempat diisi dengan string pengganti.

Agar lebih mudah dipahami, berikut contoh penggunaan dari fungsi INSERT():

```
SELECT INSERT('Belajar HTML', 9, 4, 'MySQL');
-- Belajar MySQL

SELECT INSERT('Belajar MySQL', 1, 4, 'Learn');
-- Learnjar MySQL

SELECT INSERT('Belajar MySQL', 1, 7, 'Learn');
-- Learn MySQL
```

Fungsi INSERT('Belajar HTML', 9, 4, 'MySQL') bisa dibaca: ganti 4 karakter yang ada di dalam string 'Belajar HTML', mulai dari karakter ke 9 dan hapus 4 karakter, lalu ganti karakter yang dihapus tersebut dengan string 'MySQL'. Hasilnya adalah Belajar MySQL.

Fungsi INSERT('Belajar MySQL', 1, 4, 'Learn') bisa dibaca: ganti 4 karakter yang ada di dalam string 'Belajar MySQL', mulai dari karakter ke 1 dan hapus 4 karakter, lalu ganti karakter yang dihapus tersebut dengan string 'Learn'. Hasilnya adalah Learnjar MySQL.

Fungsi INSERT('Belajar MySQL', 1, 7, 'Learn') bisa dibaca: ganti 7 karakter yang ada di dalam string 'Belajar MySQL', mulai dari karakter ke 1 dan hapus 7 karakter, lalu ganti karakter yang dihapus tersebut dengan string 'Learn'. Hasilnya adalah Learn MySQL.

22.11 Function REVERSE()

Function REVERSE() mungkin lebih cocok untuk ‘permainan’ atau untuk iseng. Fungsi ini dipakai untuk membalik urutan karakter. Berikut contoh penggunaannya:

```
SELECT REVERSE ('Belajar MySQL');
--LQSyM rajaleB
```

Dalam bab ini kita sudah membahas beberapa fungsi bawaan MySQL untuk memanipulasi tipe data **string**. Berikutnya akan dibahas tentang fungsi bawaan MySQL untuk tipe data **date**.

23. MySQL Date and Time Function

Date and time function adalah fungsi MySQL untuk memanipulasi tipe data **date** (tanggal), yang didalamnya juga mencakup **time** (waktu). Terdapat cukup banyak fungsi-fungsi mysql untuk tipe data ini. Kita akan bahas beberapa di antaranya.



Jika anda tertarik mempelajari semua **date function** bawaan MySQL, bisa ke [MySQL Documentation: Date and Time Functions¹](#) atau [MySQL Functions²](#).

23.1 Function CURDATE() dan CURRENT_DATE()

Kedua fungsi ini digunakan untuk menghasilkan tanggal saat ini yang diambil dari settingan server. Karena kita menjalankan MySQL Server dari komputer lokal, maka tanggal yang diambil adalah tanggal yang ada di komputer saat ini.

Fungsi CURDATE() dan CURRENT_DATE() tidak membutuhkan argumen apapun. Berikut contoh penggunaannya:

```
SELECT CURDATE();
-- 2017-11-25
```

```
SELECT CURRENT_DATE();
-- 2017-11-25
```

Sebagai pengingat, tipe data **date** (tanggal) di dalam MySQL disimpan dengan format yyyy-mm-dd, yakni *tahun-bulan-hari*.

Fungsi CURDATE() dan CURRENT_DATE() akan menghasilkan tanggal dalam format yyymmdd jika *terkondisi* sebagai **numeric**. Situasi ini bisa terjadi saat sebuah fungsi disambung dengan operator matematis, seperti contoh berikut:

```
SELECT CURDATE() + 0;
-- 20171125
```

```
SELECT CURRENT_DATE() + 0;
-- 20171125
```

Selain berbentuk function, terdapat juga konstanta CURRENT_DATE yang juga berfungsi untuk menghasilkan tanggal saat ini. Pemanggilan konstanta MySQL tidak perlu tanda kurung:

¹<https://dev.mysql.com/doc/refman/5.7/en/date-and-time-functions.html>

²https://www.w3schools.com/sql/sql_ref_mysql.asp

```
SELECT CURRENT_DATE;
-- 2017-11-25
```

23.2 Function CURTIME() dan CURRENT_TIME()

Fungsi CURTIME() dan CURRENT_TIME() digunakan untuk menampilkan waktu di server saat ini. Format waktu tersebut berupa hh:ii:ss (*hh: jam, ii:menit, ss: detik*). Pemanggilan fungsi CURTIME() dan CURRENT_TIME() juga tidak memerlukan argumen.

Berikut contoh penggunaan fungsi CURTIME() dan CURRENT_TIME():

```
SELECT CURTIME();
-- 17:55:49

SELECT CURRENT_TIME();
-- 17:55:49

SELECT CURTIME() + 0;
-- 175549

SELECT CURRENT_TIME() + 0;
-- 175549
```

Ketika fungsi CURTIME() dan CURRENT_TIME() diproses sebagai angka, format yang dihasilkan menjadi hhmmss. Hal ini bisa terlihat dari 2 baris terakhir dalam contoh diatas.

Terdapat pula konstanta CURTIME yang bisa dijalankan tanpa tambahan tanda kurung:

```
SELECT CURRENT_TIME;
-- 17:55:49
```

23.3 Function CURRENT_TIMESTAMP(), LOCALTIME(), LOCALTIMESTAMP() dan NOW()

Keempat fungsi ini bertujuan untuk menampilkan tanggal dan waktu server saat ini. Format yang dihasilkan berbentuk yyyy-mm-dd hh:ii:ss. Namun jika di proses sebagai angka, format yang dihasilkan berbentuk yyymmddhhiiss (tanpa tanda pemisah).

Berikut contoh penggunaan fungsi-fungsi ini:

```
SELECT CURRENT_TIMESTAMP();
-- 2017-11-25 17:55:59

SELECT LOCALTIME();
-- 2017-11-25 17:55:59

SELECT LOCALTIMESTAMP();
-- 2017-11-25 17:55:59

SELECT NOW();
-- 2017-11-25 17:55:59

SELECT CURRENT_TIMESTAMP() + 0;
-- 20171125175559

SELECT LOCALTIME() + 0;
-- 20171125175559

SELECT LOCALTIMESTAMP() + 0;
-- 20171125175559

SELECT NOW() + 0;
-- 20171125175559
```

Dapat dilihat bahwa semua fungsi diatas akan menghasilkan nilai yang sama.

Terdapat pula konstanta CURRENT_TIMESTAMP, LOCALTIME dan LOCALTIMESTAMP yang bisa dipanggil tanpa tanda kurung:

```
SELECT CURRENT_TIMESTAMP;
-- 2017-11-25 17:55:59

SELECT LOCALTIME;
-- 2017-11-25 17:55:59

SELECT LOCALTIMESTAMP;
-- 2017-11-25 17:55:59
```

23.4 Function DATE() dan TIME()

Kedua fungsi ini digunakan untuk mengambil bagian tanggal dan waktu dari sebuah input. Berikut contoh penggunaannya:

```
SELECT DATE('2017-11-25 17:55:59');
-- 2017-11-25

SELECT DATE('20171125175559');
-- 2017-11-25

SELECT TIME('2017-11-25 17:55:59');
-- 17:55:59

SELECT TIME('20171125175559');
-- 17:55:59

SELECT DATE('2017-11-25 17:55:590');
-- NULL
```

Sekilas fungsi ini tampak tidak terlalu berguna. Dalam contoh diatas saya bisa saja langsung menginput tanggal dan waktu. Namun dalam prakteknya, nilai tersebut biasanya berasal dari kolom dari sebuah tabel.

Jika fungsi DATE() dan TIME() tidak diinput dengan format yang pas, hasilnya adalah NULL seperti contoh di baris terakhir.

23.5 Function ADDDATE() dan DATE_ADD()

Kedua fungsi ini berguna untuk mencari hasil penambahan tanggal dan waktu.

Fungsi ADDDATE() dan DATE_ADD() membutuhkan 2 buah argumen. Argumen pertama berupa tanggal dan waktu awal. Argumen kedua berupa nilai tanggal dan waktu yang ingin ditambahkan. Hasil akhirnya berupa tanggal dan waktu total setelah penambahan.

Berikut format dasar dari fungsi ini:

```
ADDDATE('yyyy-mm-dd hh:ii:ss', INTERVAL jumlah_penambahan)
```

Fungsi DATE_ADD() merupakan *alias* dari ADDDATE() sehingga format penulisannya tetap sama. Berikut contoh penggunaan fungsi ini:

```
SELECT ADDDATE("2018-02-01", INTERVAL 10 DAY);
-- 2018-02-11
```

Disini saya menulis tanggal awal sebagai 2018-02-01, kemudian ditambah 10 hari (INTERVAL 10 DAY). Hasilnya adalah tanggal 2018-02-11.

Bagaimana dengan penambahan bulan dan tahun? Kita tinggal mengubah argumen kedua menjadi INTERVAL 10 MONTH atau INTERVAL 10 YEAR:

```
SELECT ADDDATE("2018-02-01", INTERVAL 10 MONTH);
-- 2018-12-01
```

```
SELECT ADDDATE("2018-02-01", INTERVAL 10 YEAR);
-- 2028-02-01
```

Untuk lebih lengkapnya, berikut daftar input argumen kedua dari fungsi ADDDATE() ini:

- INTERVAL ... DAY
- INTERVAL ... WEEK
- INTERVAL ... MONTH
- INTERVAL ... QUARTER
- INTERVAL ... YEAR
- INTERVAL ... HOUR
- INTERVAL ... MINUTE
- INTERVAL ... SECOND
- INTERVAL ... MICROSECOND

Berikut contoh yang melibatkan tanggal dan waktu:

```
SELECT ADDDATE("2018-02-01 10:30:45", INTERVAL 10 SECOND);
-- 2018-02-01 10:30:55
```

```
SELECT ADDDATE("2018-02-01 10:30:45", INTERVAL 80 MINUTE);
-- 2018-02-01 11:50:45
```

```
SELECT ADDDATE("2018-02-01 10:30:45.027432", INTERVAL 1673 HOUR);
-- 2018-04-12 03:30:45.027432
```

Fungsi ADDDATE() secara pintar akan mengkonversi waktu yang berlebih, seperti pada contoh kedua dimana saya menambahkan 80 menit ke 10:30:45, yang hasilnya adalah 11:50:45. Berikut contoh lain:

```
SELECT ADDDATE("2018-02-01", INTERVAL 30 DAY);
-- 2018-03-03
```

Bulan februari 2018 hanya ada 28 hari, sehingga tanggal 2018-02-01 ditambah 30 hari menjadi 2018-03-03.

Argumen kedua dari fungsi ADDDATE() juga bisa diisi dengan tanda negatif, yang efeknya akan menjadi pengurangan tanggal dan waktu:

```
SELECT ADDDATE("2018-02-01", INTERVAL -10 DAY);
-- 2018-01-22

SELECT ADDDATE("2018-02-01 10:30:45", INTERVAL - 128 MINUTE);
-- 2018-02-01 08:22:45
```

Selanjutnya, bagaimana jika kita ingin menambah kombinasi tanggal dan waktu sekaligus? Fungsi ADDDATE() butuh penulisan argumen kedua yang sedikit lebih rumit.

Jika kita mengkombinasikan 2 atau 3 jenis waktu, harus ditulis kode waktu yang ditambahkan tersebut. Contohnya sebagai berikut:

```
SELECT ADDDATE("2018-02-01 10:30:45", INTERVAL '2-3' YEAR_MONTH);
-- 2020-05-01 10:30:45
```

Disini saya ingin menambahkan tanggal 2018-02-01 10:30:45 dengan 2 tahun 3 bulan. Perhatikan cara penulisan argumen kedua: INTERVAL '2-3' YEAR_MONTH. Argumen ini bisa dibagi menjadi 2 bagian: INTERVAL 'NILAI' dan YEAR_MONTH. Kita harus memberitahu MySQL bahwa angka '2-3' ini adalah YEAR_MONTH.

Berikut contoh lain:

```
SELECT ADDDATE("2018-02-01 10:30:45", INTERVAL '2:30:10' HOUR_SECOND);
-- 2018-02-01 13:00:55
```

Kali ini saya menambah '2:30:10' ke tanggal "2018-02-01 10:30:45". Tapi kenapa penulisan argumen kedua seperti itu? Ini berasal dari aturan penulisan format dari MySQL.

Tabel berikut merangkum aturan penulisan argumen kedua dari fungsi ADDDATE() jika ditulis dengan lebih dari 1 nilai waktu:

Penulisan Nilai	Kode Format
'SECONDS.MICROSECONDS'	SECOND_MICROSECOND
'MINUTES:SECONDS.MICROSECONDS'	MINUTE_MICROSECOND
'MINUTES:SECONDS'	MINUTE_SECOND
'HOURS:MINUTES:SECONDS.MICROSECONDS'	HOUR_MICROSECOND
'HOURS:MINUTES:SECONDS'	HOUR_SECOND
'HOURS:MINUTES'	HOUR_MINUTE
'DAYS HOURS:MINUTES:SECONDS.MICROSECONDS'	DAY_MICROSECOND
'DAYS HOURS:MINUTES:SECONDS'	DAY_SECOND
'DAYS HOURS:MINUTES'	DAY_MINUTE
'DAYS HOURS'	DAY_HOUR
'YEARS-MONTHS'	YEAR_MONTH

Dalam contoh sebelumnya, nilai yang saya input adalah '2:30:10', karena itu kode formatnya adalah HOUR_SECOND.

Mari kita coba beberapa contoh lain:

```
SELECT ADDDATE("2018-02-01 10:30:45", INTERVAL '10 2:30:10' DAY_SECOND);
-- 2018-02-11 13:00:55
```

```
SELECT ADDDATE("2018-02-01 10:30:45.123456", INTERVAL '10 2:30:10.111111'
DAY_MICROSECOND);
-- 2018-02-11 13:00:55.234567
```

Agar bisa memahami penulisan argumen kedua dari fungsi ADDDATE() ini, kita harus menyamakan **penulisan nilai** dengan **kode format**.

Dalam contoh-contoh sebelum ini saya baru menggunakan fungsi ADDDATE(). Cara penulisan yang sama juga berlaku untuk fungsi DATE_ADD():

```
SELECT DATE_ADD("2018-02-01", INTERVAL 10 DAY);
-- 2018-02-11
```

```
SELECT DATE_ADD("2018-02-01 10:30:45.027432", INTERVAL 1673 HOUR);
-- 2018-04-12 03:30:45.027432
```

```
SELECT DATE_ADD("2018-02-01 10:30:45", INTERVAL '10 2:30:10' DAY_SECOND);
-- 2018-02-11 13:00:55
```

23.6 Function SUBDATE() dan DATE_SUB()

Sesuai dengan namanya, kedua fungsi ini adalah kebalikan dari fungsi ADDDATE() dan DATE_ADD. Fungsi SUBDATE() dan DATE_SUB() dipakai untuk proses pengurangan tanggal dan waktu.

Fungsi SUBDATE() dan DATE_SUB() membutuhkan 2 buah argumen yang sama persis seperti pada fungsi ADDDATE() dan DATE_ADD sehingga tidak akan saya bahas lagi. Berikut contoh penggunaan kedua fungsi ini:

```
SELECT SUBDATE("2018-02-01", INTERVAL 10 DAY);
-- 2018-01-22
```

```
SELECT SUBDATE("2018-02-01 10:30:45", INTERVAL '10 2:30:10' DAY_SECOND);
-- 2018-01-22 08:00:35
```

```
SELECT DATE_SUB("2018-02-01 10:30:45.027432", INTERVAL 1673 HOUR);
-- 2017-11-23 17:30:45.027432
```

```
SELECT DATE_SUB("2018-02-01 10:30:45", INTERVAL '10 2:30:10' DAY_SECOND);
-- 2018-01-22 08:00:35
```

23.7 Function ADDTIME()

Fungsi ADDTIME() berguna untuk mencari hasil penambahan tanggal dan waktu. Fungsi ini mirip seperti fungsi ADDDATE() dan DATE_ADD(), namun lebih sederhana terutama untuk penulisan argumen kedua. Selain itu fungsi ADDTIME() hanya bisa menambahkan sampai ke nilai hari. Fungsi ini tidak bisa dipakai untuk penambahan bulan dan tahun.

Fungsi ADDTIME() butuh 2 buah argumen. Argumen pertama diisi dengan tanggal dan waktu awal, serta argumen kedua yang diisi dengan tanggal dan waktu yang akan ditambahkan.

Penulisan argumen kedua untuk fungsi ADDTIME() mengikuti format standar penulisan tanggal dan waktu: dd hh:ii:ss dengan pembacaan dimulai dari sebelah kanan (dari detik).

Sebagai contoh, saya ingin menambah 2 detik dan 3600 detik menggunakan fungsi ADDTIME():

```
SELECT ADDTIME("2018-02-01 10:30:45", "2");
-- 2018-02-01 10:30:47
```

```
SELECT ADDTIME("2018-02-01 10:30:45", "3600");
-- 2018-02-01 11:06:45
```

Karena argumen kedua hanya ditulis 1 nilai saja, nilai tersebut akan diambil sebagai detik. Bagaimana untuk nilai jam, menit dan detik? Kita bisa tambahkan formatnya sebagai berikut:

```
SELECT ADDTIME("2018-02-01 10:30:45", "2:30:10");
-- 2018-02-01 13:00:55
SELECT ADDTIME("2018-02-01 10:30:45.123456", "2:30:10.222222");
-- 2018-02-01 13:00:55.345678
```

Pada contoh kedua, saya juga menambahkan hingga microdetik.

Fungsi ADDTIME() juga bisa menambahkan nilai hari, dengan contoh sebagai berikut:

```
SELECT ADDTIME("2018-02-01 10:30:45", "20 2:30:10");
-- 2018-02-21 13:00:55
```

Disini saya ingin menambahkan 20 hari 2 jam 30 menit dan 10 detik dari tanggal 2018-02-01 10:30:45. Hasilnya 2018-02-21 13:00:55.

23.8 Function DATEDIFF()

Fungsi DATEDIFF() dipakai untuk menghitung selisih hari dari 2 buah tanggal. Fungsi ini butuh 2 buah argumen, yakni tanggal akhir sebagai argumen pertama, dan tanggal awal sebagai argumen kedua.

Berikut contoh penggunaannya:

```
SELECT DATEDIFF('2018-01-15', '2017-11-25');
-- 51
```

Artinya, antara '2018-01-15' dengan '2017-11-25' ada selisih sebanyak 51 hari. Jika penulisan argumennya terbalik, hasil akhirnya berupa nilai negatif:

```
SELECT DATEDIFF('2017-11-25', '2018-04-29');
-- -155
```

23.9 Function DATE_FORMAT()

Fungsi DATE_FORMAT() bisa jadi merupakan salah satu fungsi MySQL yang akan banyak kita pakai. Fungsi ini bertujuan untuk men-format tampilan tanggal sesuai keinginan. Fungsi DATE_FORMAT() butuh 2 buah argumen. Argumen pertama berupa tanggal dan waktu . Argumen kedua berupa string yang berisi format penulisan tampilan tanggal dan waktu tersebut.

Berikut contoh penggunaannya:

```
SELECT DATE_FORMAT('2018-06-29', '%d %m %y');
-- 29 06 18
```

Perhatikan kode format yang dipakai sebagai argumen kedua. Nilai '%d %m %y' Artinya saya ingin tampilan nilai tanggal dalam bentuk: tanggal (00..31) bulan (00..12) tahun (dua digit).

Terdapat format lain yang bisa dipakai, termasuk dalam bentuk nama hari serta nama bulan seperti contoh berikut:

```
SELECT DATE_FORMAT('2018-06-29', '%d-%M-%Y');
-- 29-June-2018
```

Format yang bisa dipakai sangat beragam, yang dirangkum dalam tabel berikut:

Kode Format	Penjelasan
%a	Singkatan nama hari (Sun..Sat)
%b	Singkatan nama bulan (Jan..Dec)
%c	Bulan dalam bentuk angka (0..12)
%D	Hari dalam bentuk angka + tambahan awalan (1st, 2nd ...)
%d	Hari dalam bentuk angka (00..31)
%e	Hari dalam bentuk angka (0..31)
%f	Microdetik (000000..999999)
%H	Jam (00..23)
%h	Jam (01..12)
%I	Jam (01..12)
%i	Menit dalam bentuk angka (00..59)
%j	Hari dalam 1 tahun (001..366)

Kode Format	Penjelasan
%k	Jam (0..23)
%l	Jam (1..12)
%M	Nama bulan (January..December)
%m	Bulan dalam bentuk angka (00..12)
%p	AM atau PM
%r	Waktu dengan format 12 jam (hh:mm:ss diikuti AM atau PM)
%S	Detik (00..59)
%s	Detik (00..59)
%T	Waktu, dalam format 24 jam (hh:mm:ss)
%W	Nama hari (Sunday..Saturday)
%w	Hari dalam bentuk angka (0=Sunday..6=Saturday)
%Y	Tahun dalam bentuk 4 digit angka
%y	Tahun dalam bentuk 2 digit angka
%%	Karakter khusus %

Kode diatas digunakan sebagai nilai argumen kedua dari fungsi DATE_FORMAT(). Berikut contoh lain dari penggunaan format tersebut:

```
SELECT DATE_FORMAT('2018-06-29 20:30:15', '%d#%m#%Y, %l:%i:%s %p');
-- 29#06#2018, 8:30:15 PM
```

```
SELECT DATE_FORMAT('2018-06-29 20:30:15', '%W %M %Y');
-- Friday June 2018
```

Dengan menggunakan fungsi DATE_FORMAT(), kita tidak perlu terpaku dengan format tipe data date MySQL yang penulisannya ‘terbalik’, yakni *tahun-bulan-tanggal*.

Khusus untuk nama hari dan bulan, bawaan MySQL berupa nama hari dan bulan dalam bahasa inggris, seperti *Friday* dan *June*. Bisakah diubah menjadi bahasa indonesia?

Di dalam MySQL terdapat sistem variabel yang bernama `lc_time_names`. Variabel ini berfungsi untuk mengatur format bahasa tanggal. Kita bisa mengubah isi variabel ini agar mendukung nama hari dan bulan dalam bahasa indonesia.

Secara default, variabel `lc_time_names` berisi kode `en_US`. Untuk memeriksanya, jalankan perintah berikut:

```
SELECT @@lc_time_names;
-- en_US
```

Kode ‘`en_US`’ berarti ‘*bahasa inggris_amerika serikat*’ atau ‘*english_united states*’. Ini merupakan kode bahasa ISO ([ISO Language Code³](#)).

Kita akan mengubah nilai variabel `lc_time_names` menjadi ‘`id_ID`’, yakni kode untuk Bahasa Indonesia. Caranya, jalankan perintah berikut:

³<http://www.lingoes.net/en/translator/langcode.htm>

```
SET lc_time_names = 'id_ID';

SELECT @@lc_time_names;
-- id_ID
```

Sekarang, format tanggal MySQL sudah menggunakan Bahasa Indonesia. Mari kita coba:

```
SELECT DATE_FORMAT('2018-06-29 20:30:15', '%W %M %Y');
-- Jumat Juni 2018

SELECT DATE_FORMAT('2018-06-29 20:30:15', '%d %M %Y %l:%i:%s %p');
-- 29 Juni 2018 8:30:15 PM

SELECT DATE_FORMAT(NOW(), '%d %M %Y %l:%i:%s %p');
-- 26 November 2017 5:04:03 PM
```

Terlihat, nama hari dan bulan sudah dalam bentuk bahasa indonesia.

Sebagai catatan, perintah `SET lc_time_names = 'id_ID'` berlaku untuk satu session. Artinya ketika kita memutus hubungan dengan MySQL Server, pengaturannya akan kembali ke default, yakni '`en_US`'.

23.10 Function STR_TO_DATE()

Fungsi `STR_TO_DATE()` adalah kebalikan dari fungsi `DATE_FORMAT()`. Fungsi `STR_TO_DATE()` berguna untuk mengambil nilai tanggal dan waktu dari sebuah string.

Fungsi `STR_TO_DATE()` membutuhkan 2 buah argumen. Argumen pertama berupa string asal yang di dalamnya terdapat tanggal atau waktu. Argumen kedua berupa kode format. Hasil akhir fungsi ini adalah tipe data `date` MySQL dalam format `yyyy-mm-dd`. Berikut contoh penggunaannya:

```
SELECT STR_TO_DATE('29-06-18', '%d-%m-%Y');
-- 2018-06-29

SELECT STR_TO_DATE('06#29#2018', '%m#%d#%Y');
-- 2018-06-29
```

Dalam contoh pertama, string asal adalah '`29-06-18`'. Meskipun berbentuk tanggal, nilai ini tidak bisa langsung diinput ke dalam tabel MySQL karena formatnya salah. Dengan menggunakan fungsi `STR_TO_DATE()`, kita bisa menyesuaikan format string tersebut.

Kode format untuk argumen kedua dari fungsi `STR_TO_DATE()` sama seperti tabel pada pembahasan fungsi `DATE_FORMAT()`.

Fungsi `STR_TO_DATE()` ini akan berguna jika kita memiliki tabel yang kolomnya ingin dikonversi menjadi tipe data `date` namun format di dalam kolom tersebut belum berbentuk `yyyy-mm-dd`.

Berikut contoh lain dari fungsi `STR_TO_DATE()`

```
SELECT STR_TO_DATE('29 June 2018 8:30:15 PM', '%d %M %Y %l:%i:%s %p');
-- 2018-06-29 20:30:15
```

```
SELECT STR_TO_DATE('29 Juni 2018 8:30:15 PM', '%d %M %Y %l:%i:%s %p');
-- NULL
```

Bisakah anda mencari tahu kenapa contoh di baris terakhir menghasilkan nilai NULL?

Penyebabnya ada di penulisan nama bulan. Fungsi STR_TO_DATE() hanya bisa membaca nama bulan dalam bahasa inggris, yakni June. Di baris terakhir saya menggunakan nama bulan dalam bahasa indonesia, yakni Juni.

23.11 Function SEC_TO_TIME()

Fungsi SEC_TO_TIME() berguna untuk konversi nilai detik ke dalam bentuk jam dan menit. Fungsi ini butuh 1 argumen yakni jumlah detik yang akan dikonversi. Berikut contoh penggunaannya:

```
SELECT SEC_TO_TIME(30);
-- 00:00:30
```

```
SELECT SEC_TO_TIME(3600);
-- 01:00:00
```

```
SELECT SEC_TO_TIME(12654);
-- 03:30:54
```

```
SELECT SEC_TO_TIME(-1237);
-- -00:20:37
```

23.12 MySQL Function vs PHP Function

Sebelum menutup bab ini, saya ingin membahas sedikit tentang MySQL Function vs PHP Function. Jika sebelumnya anda sudah mempelajari PHP, hampir semua fungsi-fungsi yang kita bahas dalam bab ini juga tersedia di dalam PHP.

Pertanyaannya, data dari database sebaiknya di olah dari MySQL langsung atau di PHP saja? Contoh situasinya seperti ini. Secara bawaan, tipe data date MySQL disimpan dalam format **yyyy-mm-dd**. Bentuk format seperti ini tidak umum dipakai. Sehari-hari kita terbiasa melihat tanggal dalam format **dd-mm-yyyy**.

Jadi apakah sebaiknya menggunakan fungsi DATE_FORMAT() dari query MySQL lalu mengirimnya ke PHP, atau mengirimkan “data mentah” dari MySQL lalu menggunakan fungsi date() PHP untuk men-format tanggal tersebut?

Tidak ada jawaban yang pasti untuk hal ini. Keduanya bisa dipakai tergantung kebutuhan. Jika anda merasa lebih paham PHP, boleh ambil langsung data dari MySQL, lalu diolah di sisi PHP. Jika anda ingin menerapkan skill MySQL, data tanggal tersebut bisa di-format dulu di sisi MySQL, sehingga sesampainya di PHP tinggal di tampilkan saja.

Hanya saja, menurut saya kebanyakan orang akan lebih suka data tersebut diolah di PHP, karena PHP memang ditujukan untuk programming, sedangkan MySQL lebih ke tempat penyimpanan data yang kebetulan juga memiliki fitur programming (*function*) sebagai nilai tambah.

Dalam 3 bab ini kita telah membahas puluhan fungsi bawaan MySQL untuk tipe data **numeric**, **string** dan **date**. Karena keterbatasan tempat, masih ada fungsi-fungsi bawaan lain yang belum saya bahas. Jika anda tertarik, bisa mempelajarinya ke [MySQL Manual: Functions and Operators⁴](#).

Di dalam MySQL manual tersebut juga tersedia fungsi lanjutan untuk Encryption, BIT, XML, dll yang bisa di eksplorasi lebih lanjut.

Berikutnya kita akan bahas tentang **MySQL View**.

⁴<https://dev.mysql.com/doc/refman/5.7/en/functions.html>

24. MySQL VIEW

View, Stored Procedure dan **Trigger** termasuk materi advanced di dalam MySQL. Ketiganya mungkin tidak sering dipakai, namun dengan memahami cara penggunaan fitur ini, akan mempersiapkan kita untuk project dan tugas yang lebih kompleks.

Dalam bab ini saya akan bahas tentang **view** terlebih dahulu.

24.1 Pengertian VIEW

View adalah tabel *virtual* atau tabel logis yang berasal dari hasil query **SELECT**. Dengan menggunakan view, kita bisa membuat ‘tabel’ baru yang isinya diambil dari tabel yang sudah ada saat ini.

View bersifat dinamis, dalam artian ketika tabel asal diupdate, seluruh view yang dibuat dari tabel tersebut juga akan berubah.

Berikut beberapa fitur dan keunggulan dari **view**:

Menyederhanakan query yang kompleks

Salah satu bentuk query kompleks adalah yang melibatkan *subquery* (query di dalam query) bahkan bisa bertingkat-tingkat. Subquery ini bisa dikonversi menjadi view sehingga penulisan querynya jadi lebih sederhana.

Menyembuyikan struktur asli tabel

Kita bisa membuat view yang berasal dari sebagian kolom tabel asli. User yang tidak berhak cukup mengakses view saja.

Membuat kolom komputasi

Design database yang ideal sebaiknya tidak menyimpan hasil perhitungan ke dalam tabel. Sebagai contoh, kita tidak perlu membuat kolom IPK ke dalam tabel `nilai_mahasiswa`. Kolom IPK ini bisa digenerate secara otomatis dari perhitungan nilai tiap semester. Kolom hasil perhitungan IPK seperti ini bisa ditempatkan ke dalam view.

Menjaga *backward compatibility*

Misalkan kita mengembangkan suatu aplikasi dengan database terpusat yang akan diakses oleh ratusan client. Konsep aplikasi seperti ini memiliki resiko ketika ingin mengubah struktur database, karena belum tentu setiap client sudah mengupdate aplikasinya ke versi terbaru. Untuk mengatasi hal ini, kita bisa membuat view seperti database lama yang diambil dari struktur database baru.

Pengertian view ini akan lebih jelas ketika kita masuk ke dalam contoh praktek sesaat lagi.

Mempersiapkan tabel mahasiswa dan universitas

Sebagai tabel sample, saya kembali menggunakan tabel `mahasiswa` dan `universitas`. Berikut query yang dipakai untuk membuat kedua tabel ini:

```

DROP TABLE IF EXISTS mahasiswa;
DROP TABLE IF EXISTS universitas;

CREATE TABLE mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR (20),
    nilai_uan DEC(5,2)
) ENGINE = InnoDB;

INSERT INTO mahasiswa VALUES
('17090113', 'Riana Putria', 'Padang', 'Kimia', 339.20),
('17140143', 'Rudi Permana', 'Bandung', 'Ilmu Komputer', 290.44),
('17080305', 'Rina Kumala Sari', 'Jakarta', 'Akuntansi', 337.99),
('17140119', 'Sandri Fatmala', 'Bandung', 'Ilmu Komputer', 322.91),
('17080225', 'Husli Khairan', 'Jakarta', 'Akuntansi', 288.55),
('17140133', 'Ikhsan Prayoga', 'Jakarta', 'Ilmu Komputer', 300.16),
('17140120', 'Bobby Permana', 'Medan', 'Ilmu Komputer', 280.82);

CREATE TABLE universitas (
    jurusan VARCHAR(20) PRIMARY KEY,
    tgl_berdiri DATE,
    nama_dekan VARCHAR(50),
    jum_mhs SMALLINT UNSIGNED,
    akr ENUM('A', 'B', 'C', 'N/A')
) ENGINE = InnoDB;

INSERT INTO universitas VALUES
('Kimia', '1987-07-12', 'Prof. Mulyono, M.Sc.', 662, 'B'),
('Ilmu Komputer', '2003-02-23', 'Dr. Syahrial, M.Kom.', 412, 'A'),
('Akuntansi', '1985-03-19', 'Maya Fitrianti, M.M.', 895, 'B'),
('Farmasi', '1997-05-30', 'Prof. Silvia Nst, M.Farm.', 312, 'C'),
('Fisika', '1989-12-10', 'Dr. Umar Agustinus, M.Sc.', 275, 'A'),
('Hukum', '1983-08-08', 'Prof. Gunarto, M.H.', 754, 'B');

SELECT * FROM mahasiswa;
+-----+-----+-----+-----+-----+
| nim      | nama           | asal       | jurusan        | nilai_uan |
+-----+-----+-----+-----+-----+
| 17080225 | Husli Khairan | Jakarta   | Akuntansi     | 288.55    |

```

17080305 Rina Kumala Sari Jakarta Akuntansi 337.99
17090113 Riana Putria Padang Kimia 339.20
17140119 Sandri Fatmala Bandung Ilmu Komputer 322.91
17140120 Bobby Permana Medan Ilmu Komputer 280.82
17140133 Ikhsan Prayoga Jakarta Ilmu Komputer 300.16
17140143 Rudi Permana Bandung Ilmu Komputer 290.44

```
SELECT * FROM universitas;
```

jurusan	tgl_berdiri	nama_dekan	jumlah_mhs	akreditasi
Akuntansi	1985-03-19	Maya Fitrianti, M.M.	895	B
Farmasi	1997-05-30	Prof. Silvia Nst, M.Farm.	312	C
Fisika	1989-12-10	Dr. Umar Agustinus, M.Sc.	275	A
Hukum	1983-08-08	Prof. Gunarto, M.H.	754	B
Ilmu Komputer	2003-02-23	Dr. Syahrial, M.Kom.	412	A
Kimia	1987-07-12	Prof. Mulyono, M.Sc.	662	B

Seperti biasa, tabel mahasiswa dan universitas ini ditempatkan ke dalam database belajar.

24.2 Membuat dan Menghapus VIEW

Pada dasarnya view merupakan query SELECT yang “tersimpan”. Oleh karena itu isi view akan diambil dari hasil query SELECT. Berikut format dasar yang dipakai untuk membuat view:

```
CREATE VIEW [database_name].[view_name] AS [SELECT statement]
```

Bagian [database_name]. [view_name] adalah penulisan lengkap, biasanya kita hanya perlu menulis view_name saja sebagai nama dari view. Nama database sering kali sudah terpilih secara default yang berasal dari query seperti USE belajar.

Bagian [SELECT statement] adalah query SELECT yang akan menjadi isi dari view.



Format dasar view diatas adalah versi sederhana karena saya tidak akan membahas semua aspek tentang view. Apabila anda berminat untuk melihat format lengkap pembuatan view, bisa ke [MySQL Reference Manual: CREATE VIEW Syntax¹](#).

Sebagai contoh pertama, saya akan membuat view detail_mhs yang isinya berasal dari tabel mahasiswa:

¹<https://dev.mysql.com/doc/refman/5.7/en/create-view.html>

```
CREATE VIEW detail_mhs AS
SELECT nama, nim, jurusan FROM mahasiswa ORDER BY nama ASC;
-- Query OK, 0 rows affected (0.09 sec)
```

Hasil dari perintah diatas, view dengan nama detail_mhs berhasil dibuat. Untuk mengakses view ini, caranya sama seperti mengakses tabel biasa:

```
SELECT * FROM detail_mhs;
+-----+-----+-----+
| nama      | nim       | jurusan    |
+-----+-----+-----+
| Bobby Permana | 17140120 | Ilmu Komputer |
| Husli Khairan | 17080225 | Akuntansi   |
| Ikhsan Prayoga | 17140133 | Ilmu Komputer |
| Riana Putria  | 17090113 | Kimia        |
| Rina Kumala Sari | 17080305 | Akuntansi   |
| Rudi Permana  | 17140143 | Ilmu Komputer |
| Sandri Fatmala | 17140119 | Ilmu Komputer |
+-----+-----+-----+
```

Inilah isi dari view detail_mhs. Terlihat tidak ada bedanya dengan tabel biasa. View diatas berisi kolom `nama`, `nim` dan `jurusan` yang diambil dari tabel `mahasiswa`.

Jika menggunakan perintah `SHOW TABLES`, view detail_mhs juga akan terlihat:

```
SHOW TABLES;
+-----+
| Tables_in_belajar |
+-----+
| detail_mhs        |
| mahasiswa         |
| ...               |
+-----+
```

Bagaimana cara membedakan antara view dengan tabel asli? Kita bisa menjalankan query `SHOW FULL TABLES` agar hasilnya lebih detail:

```
SHOW FULL TABLES;
+-----+-----+
| Tables_in_belajar | Table_type |
+-----+-----+
| detail_mhs        | VIEW       |
| mahasiswa         | BASE TABLE |
| ...               | ...        |
+-----+-----+
```

Query SHOW FULL TABLES akan menampilkan semua tabel di dalam database belajar beserta tipe dari tabel tersebut. Jika itu adalah view, maka kolom Table_type akan bernilai VIEW. Jika itu adalah tabel ‘asli’, Table_type-nya akan berisi BASE TABLE.

Lebih jauh lagi, jika anda membuka file tempat database disimpan, yakni C:\xampp\mysql\data\belajar, akan terlihat sebuah file **detail_mhs.frm** yang menyimpan isi dari view.

Untuk melihat perintah pembuatan dari sebuah view, bisa menjalankan query SHOW CREATE VIEW seperti contoh berikut:

```
SHOW CREATE VIEW detail_mhs \G
*****
View: detail_mhs
Create View:
CREATE
    ALGORITHM=UNDEFINED
    DEFINER=`root`@`localhost`
    SQL SECURITY DEFINER
        VIEW `detail_mhs` AS select `mahasiswa`.`nama` AS `nama`, `mahasiswa`.`nim` AS `nim`, `mahasiswa`.`jurusan` AS `jurusan` from `mahasiswa` order by `mahasiswa`.`nama`
character_set_client: cp850
collation_connection: cp850_general_ci
-- 1 row in set (0.00 sec)
```

Inilah perintah detail yang dijalankan MySQL pada saat pembuatan view **detail_mhs**.

Sebagian besar dari hasil diatas merupakan pengaturan opsional yang terlalu teknis sehingga tidak akan saya bahas, misalnya ALGORITHM yang mengatur cara kerja penyimpanan view. Jika pengaturan ini tidak ditulis, akan menggunakan pengaturan default MySQL.

Untuk mengubah isi sebuah view, kita bisa menggunakan query ALTER VIEW. Misalkan saya ingin mengubah view **detail_mhs** agar urutan kolomnya menjadi **nim, nama dan jurusan**:

```
ALTER VIEW detail_mhs AS
SELECT nim, nama, jurusan FROM mahasiswa ORDER BY nim DESC;
-- Query OK, 0 rows affected (0.11 sec)

SELECT * FROM detail_mhs;
+-----+-----+-----+
| nim | nama | jurusan |
+-----+-----+-----+
| 17140143 | Rudi Permana | Ilmu Komputer |
| 17140133 | Ikhsan Prayoga | Ilmu Komputer |
| 17140120 | Bobby Permana | Ilmu Komputer |
| 17140119 | Sandri Fatmala | Ilmu Komputer |
| 17090113 | Riana Putria | Kimia |
| 17080305 | Rina Kumala Sari | Akuntansi |
| 17080225 | Husli Khairan | Akuntansi |
+-----+-----+-----+
```

Cara lain untuk mengubah view adalah dengan menggunakan query CREATE OR REPLACE VIEW, seperti contoh berikut:

```
CREATE OR REPLACE VIEW detail_mhs AS
SELECT jurusan, nama FROM mahasiswa;
-- Query OK, 0 rows affected (0.03 sec)
```

```
SELECT * FROM detail_mhs;
+-----+-----+
| jurusan | nama |
+-----+-----+
| Akuntansi | Husli Khairan |
| Akuntansi | Rina Kumala Sari |
| Kimia | Riana Putria |
| Ilmu Komputer | Sandri Fatmala |
| Ilmu Komputer | Bobby Permana |
| Ilmu Komputer | Ikhsan Prayoga |
| Ilmu Komputer | Rudi Permana |
+-----+-----+
```

Query CREATE OR REPLACE VIEW akan membuat sebuah view baru jika view tersebut belum ada, atau akan menimpa view yang sudah ada. Hasilnya, isi view detail_mhs akan berubah.

Untuk menghapus view, tersedia query DROP VIEW dengan format dasar sebagai berikut:

```
DROP VIEW [IF EXISTS] [database_name].[view_name]
```

Untuk menghapus view detail_mhs, saya bisa menjalankan query dibawah ini:

```
DROP VIEW detail_mhs;
-- Query OK, 0 rows affected (0.00 sec)
```

Sekarang, view detail_mhs sudah terhapus.

Sebagai informasi tambahan, pemilihan nama view mengikuti aturan penulisan *identifier*, yakni sama seperti aturan penulisan untuk nama tabel dan nama database.

Karena view merupakan bagian dari anggota database, nama view juga tidak boleh sama dengan nama tabel yang sudah ada. Sebagai contoh, saya tidak bisa membuat view bernama universitas karena nama tersebut sudah diambil oleh tabel lain di dalam database yang sama (database belajar):

```
CREATE VIEW universitas AS
SELECT nama, nim, jurusan FROM mahasiswa ORDER BY nama ASC;
-- ERROR 1050 (42S01): Table 'universitas' already exists
```

24.3 VIEW untuk JOIN dan Penambahan Data

Contoh view detail_mhs yang saya buat sebelum ini sangat sederhana karena hanya mengambil beberapa kolom dari 1 tabel saja. View biasanya dipakai untuk query yang lebih kompleks seperti hasil SELECT...JOIN.

Sebagai contoh selanjutnya, saya ingin membuat ulang view detail_mhs yang berasal dari 2 buah tabel: mahasiswa dan universitas. Query JOIN akan dipakai untuk menggabungkan kedua tabel ini.

Untuk membuat view yang cukup kompleks, sebaiknya kita uji dengan menjalankan query SELECT terlebih dahulu. Setelah dirasa pas, baru dibuat menjadi view:

```
SELECT nama, nim, mahasiswa.jurusan, nama_dekan
FROM mahasiswa INNER JOIN universitas
WHERE mahasiswa.jurusan = universitas.jurusan
ORDER BY nama ASC;
```

nama	nim	jurusan	nama_dekan
Bobby Permana	17140120	Ilmu Komputer	Dr. Syahrial, M.Kom.
Husli Khairan	17080225	Akuntansi	Maya Fitrianti, M.M.
Ikhsan Prayoga	17140133	Ilmu Komputer	Dr. Syahrial, M.Kom.
...

Query SELECT...JOIN diatas menggabungkan tabel mahasiswa dengan universitas. Mari kita buat menjadi view:

```
CREATE VIEW detail_mhs AS
  SELECT nama, nim, mahasiswa.jurusan, nama_dekan
  FROM mahasiswa INNER JOIN universitas
  WHERE mahasiswa.jurusan = universitas.jurusan
  ORDER BY nama ASC;
```

```
SELECT * FROM detail_mhs;
```

nama	nim	jurusan	nama_dekan
Bobby Permana	17140120	Ilmu Komputer	Dr. Syahrial, M.Kom.
Husli Khairan	17080225	Akuntansi	Maya Fitrianti, M.M.
Ikhsan Prayoga	17140133	Ilmu Komputer	Dr. Syahrial, M.Kom.
...

Dalam query yang cukup kompleks seperti inilah view lebih bermanfaat daripada menjalankan query SELECT secara manual.

Lebih lanjut, view detail_mhs ini bisa diolah lebih jauh lagi. Misalnya saya ingin menampilkan kolom nama mahasiswa dan kolom nama_dekan saja:

```
SELECT nama, nama_dekan FROM detail_mhs;
```

nama	nama_dekan
Bobby Permana	Dr. Syahrial, M.Kom.
Husli Khairan	Maya Fitrianti, M.M.
...	...

Atau bagaimana dengan membuat view baru dari hasil view detail_mhs diatas? Tidak ada masalah:

```
CREATE VIEW detail_mhs_format AS
  SELECT
    UCASE(nama) AS 'Nama Mahasiswa',
    CONCAT(jurusan, ' (' , nama_dekan, ')' ) AS 'Nama Jurusan dan Dekan'
  FROM detail_mhs;
```

Disini saya membuat view detail_mhs_format yang kolomnya diambil dari view detail_mhs. Selain itu kolom nama, jurusan dan nama_dekan di format menggunakan fungsi string bawaan MySQL: UCASE() dan CONCAT().

Berikut hasilnya:

```
SELECT * FROM detail_mhs_format;
+-----+-----+
| Nama Mahasiswa | Nama Jurusan dan Dekan |
+-----+-----+
| HUSLI KHAIRAN | Akuntansi (Maya Fitrianti, M.M.) |
| RINA KUMALA SARI | Akuntansi (Maya Fitrianti, M.M.) |
| SANDRI FATMALA | Ilmu Komputer (Dr. Syahrial, M.Kom.) |
| ... | ...
+-----+-----+
```

Selanjutnya, mari kita uji dengan menginput data baru ke dalam tabel `mahasiswa`. Apakah ada perubahan di sisi view?

```
UPDATE universitas SET nama_dekan = 'Prof.Rika Susilawati M.Kom'
WHERE jurusan = 'Ilmu Komputer';
-- Query OK, 1 row affected (0.12 sec)
-- Rows matched: 1 Changed: 1 Warnings: 0
```

Query diatas dipakai untuk mengubah nama dekan jurusan Ilmu Komputer. Sebelumnya, nama dekan adalah Dr. Syahrial, M.Kom. yang diubah menjadi Prof.Rika Susilawati M.Kom.

Mari kita cek hasil dari view `detail_mhs` dan `detail_mhs_format`:

```
SELECT * FROM detail_mhs;
+-----+-----+-----+-----+
| nama | nim | jurusan | nama_dekan |
+-----+-----+-----+-----+
| Bobby Permana | 17140120 | Ilmu Komputer | Prof.Rika Susilawati M.Kom |
| Husli Khairan | 17080225 | Akuntansi | Maya Fitrianti, M.M. |
| ... | ... | ... | ...
+-----+-----+-----+-----+
```

```
SELECT * FROM detail_mhs_format;
+-----+-----+
| Nama Mahasiswa | Nama Jurusan dan Dekan |
+-----+-----+
| HUSLI KHAIRAN | Akuntansi (Maya Fitrianti, M.M.) |
| RINA KUMALA SARI | Akuntansi (Maya Fitrianti, M.M.) |
| SANDRI FATMALA | Ilmu Komputer (Prof.Rika Susilawati M.Kom) |
| BOBBY PERMANA | Ilmu Komputer (Prof.Rika Susilawati M.Kom) |
| ... | ...
+-----+-----+
```

Seperti yang bisa terlihat, nama dekan jurusan Ilmu Komputer juga ikut berubah ketika kita mengakses view `detail_mhs` dan `detail_mhs_format`.

24.4 Update data dari VIEW

Fitur lain dari view adalah kita bisa mengupdate data ke dalam view. Efeknya, tabel yang menjadi bagian dari view juga ikut diupdate. Update ini mencakup penambahan data baru, menghapus data, hingga mengubah data yang ada.

Akan tetapi, tidak semua view bisa diupdate. Jika view tersebut dibuat dari query dibawah ini, maka tidak bisa diupdate:

- Aggregate function seperti MIN(), MAX(), SUM(), AVG(), COUNT(), dll.
- DISTINCT
- GROUP BY
- HAVING
- UNION atau UNION ALL
- LEFT JOIN atau OUTER JOIN
- Subquery

Secara umum, kita bisa mengupdate data ke dalam view yang dibuat dari query SELECT sederhana.

Untuk contoh praktik, saya akan membuat view `mahasiswa_ilkom` yang isinya berasal dari tabel `mahasiswa`. Sebagai syarat tambahan, view ini hanya akan menampilkan data mahasiswa yang memilih jurusan Ilmu Komputer saja. Berikut query yang dipakai:

```
CREATE VIEW mahasiswa_ilkom AS
SELECT nim, nama, jurusan FROM mahasiswa WHERE jurusan = 'Ilmu Komputer';

SELECT * FROM mahasiswa_ilkom;
+-----+-----+-----+
| nim      | nama           | jurusan        |
+-----+-----+-----+
| 17140119 | Sandri Fatmala | Ilmu Komputer |
| 17140120 | Bobby Permana  | Ilmu Komputer |
| 17140133 | Ikhsan Prayoga  | Ilmu Komputer |
| 17140143 | Rudi Permana   | Ilmu Komputer |
+-----+-----+-----+
```

Selanjutnya, saya akan menginput mahasiswa baru ke dalam view `mahasiswa_ilkom`:

```
INSERT INTO mahasiswa_ilkom VALUES
('17140155', 'Tika Herina', 'Ilmu Komputer');
```

```
SELECT * FROM mahasiswa_ilkom;
```

nim	nama	jurusan
17140119	Sandri Fatmala	Ilmu Komputer
17140120	Bobby Permana	Ilmu Komputer
17140133	Ikhsan Prayoga	Ilmu Komputer
17140143	Rudi Permana	Ilmu Komputer
17140155	Tika Herina	Ilmu Komputer

Terlihat, nama Tika Herina sudah masuk ke dalam view.

Sebagaimana yang sudah kita pelajari, view hanyalah *tbl virtual*. Data asli dari view itu sendiri tetap diambil dari tabel asal. Dengan demikian, perintah **INSERT** ke dalam view **mahasiswa_ilkom** secara tidak langsung akan mengupdate tabel **mahasiswa**. Mari kita periksa:

```
SELECT * FROM mahasiswa;
```

nim	nama	asal	jurusan	nilai_uan
17080225	Husli Khairan	Jakarta	Akuntansi	288.55
...	
17140143	Rudi Permana	Bandung	Ilmu Komputer	290.44
17140155	Tika Herina	NULL	Ilmu Komputer	NULL

Terlihat mahasiswa Tika Herina juga ada di dalam tabel **mahasiswa**. Hanya saja, karena view **mahasiswa_ilkom** hanya terdiri dari kolom **nim**, **nama** dan **jurusan** saja, maka nilai untuk kolom **asal** dan **nilai_uan** berisi nilai default. Dalam contoh ini, **NULL** adalah nilai default untuk kolom **asal** dan **nilai_uan**.

Efek yang sama juga terjadi jika baris di dalam view dihapus:

```
DELETE FROM mahasiswa_ilkom WHERE nama = 'Bobby Permana';
```

```
SELECT * FROM mahasiswa_ilkom;
```

nim	nama	jurusan
17140119	Sandri Fatmala	Ilmu Komputer
17140133	Ikhsan Prayoga	Ilmu Komputer
17140143	Rudi Permana	Ilmu Komputer

```
| 17140155 | Tika Herina      | Ilmu Komputer |
+-----+-----+-----+
SELECT * FROM mahasiswa;
+-----+-----+-----+-----+-----+
| nim      | nama           | asal     | jurusan       | nilai_uan |
+-----+-----+-----+-----+
| 17080225 | Husli Khairan   | Jakarta  | Akuntansi    | 288.55   |
| 17080305 | Rina Kumala Sari | Jakarta  | Akuntansi    | 337.99   |
| 17090113 | Riana Putria     | Padang    | Kimia        | 339.20   |
| 17140119 | Sandri Fatmala   | Bandung   | Ilmu Komputer| 322.91   |
| 17140133 | Ikhsan Prayoga    | Jakarta  | Ilmu Komputer| 300.16   |
| 17140143 | Rudi Permana     | Bandung   | Ilmu Komputer| 290.44   |
| 17140155 | Tika Herina      | NULL      | Ilmu Komputer| NULL     |
+-----+-----+-----+-----+
```

Ketika data mahasiswa Bobby Permana dihapus dari view `mahasiswa_ilkom`, baris tersebut juga akan terhapus di tabel `mahasiswa`.

24.5 Konsistensi Update VIEW dengan WITH CHECK OPTION

Fitur update ke dalam view yang kita coba sebelum ini memiliki sebuah masalah konsistensi. View `mahasiswa_ilkom` hanya menampilkan mahasiswa yang memiliki jurusan Ilmu Komputer saja, ini karena terdapat kondisi `WHERE jurusan = 'Ilmu Komputer'` pada saat pembuatan view.

Agar lebih jelas, saya akan membuat ulang view `mahasiswa_ilkom` sebelum ini:

```
DROP VIEW mahasiswa_ilkom;

CREATE VIEW mahasiswa_ilkom AS
SELECT nim, nama, jurusan FROM mahasiswa WHERE jurusan = 'Ilmu Komputer';

SELECT * FROM mahasiswa_ilkom;
+-----+-----+-----+
| nim      | nama           | jurusan       |
+-----+-----+-----+
| 17140119 | Sandri Fatmala | Ilmu Komputer |
| 17140133 | Ikhsan Prayoga | Ilmu Komputer |
| 17140143 | Rudi Permana   | Ilmu Komputer |
| 17140155 | Tika Herina    | Ilmu Komputer |
+-----+-----+-----+
```

Sekarang, apa yang terjadi jika saya jalankan query berikut?

```
INSERT INTO mahasiswa_ilkom VALUES
('17090127', 'Herman Eka Putra', 'Kimia');
Query OK, 1 row affected (0.07 sec)
```

Query diatas sukses berjalan. Isinya saya menambahkan data baru ke dalam view mahasiswa_ilkom.

Masalahnya, jurusan mahasiswa yang ditambahkan adalah **Kimia**, bukan **Ilmu Komputer**. Penambahan data diatas tidak akan terlihat dari view mahasiswa_ilkom karena view ini hanya akan menampilkan mahasiswa yang memilih jurusan Ilmu Komputer saja.

Namun query diatas tetap bisa diakses dari tabel “induk” view mahasiswa_ilkom, yakni tabel mahasiswa:

```
SELECT * FROM mahasiswa;
+-----+-----+-----+-----+-----+
| nim | nama | asal | jurusan | nilai_uan |
+-----+-----+-----+-----+-----+
| ... | ... | ... | ... | ... |
| 17090127 | Herman Eka Putra | NULL | Kimia | NULL |
| ... | ... | ... | ... | ... |
+-----+-----+-----+-----+-----+
```

Untuk mengatasi masalah ketidakkonsistenan ini, tersedia perintah tambahan **WITH CHECK OPTION** yang bisa ditulis pada saat pendefenisian view atau kita bisa mengubah view dengan perintah **ALTER**.

Saya akan coba buat ulang view mahasiswa_ilkom dengan tambahan perintah **WITH CHECK OPTION** ini:

```
DROP VIEW mahasiswa_ilkom;

CREATE VIEW mahasiswa_ilkom AS
SELECT nim, nama, jurusan FROM mahasiswa WHERE jurusan = 'Ilmu Komputer'
WITH CHECK OPTION;
```

Sekarang, saya akan coba input kembali mahasiswa baru dengan jurusan yang bukan Ilmu Komputer:

```
INSERT INTO mahasiswa_ilkom VALUES
('17080276', 'Rano Tobing', 'Akuntansi');

-- ERROR 1369 (HY000): CHECK OPTION failed 'belajar.mahasiswa_ilkom'
```

Akan tampil pesan error: *CHECK OPTION failed* karena data yang akan ditambahkan tidak sesuai dengan syarat kondisi dari view mahasiswa_ilkom.

Perintah **WITH CHECK OPTION** akan menjaga konsistensi view dengan membatasi input hanya yang sesuai dengan syarat yang bisa diterima oleh view.

24.6 Latihan VIEW

Menutup bab tentang view, saya ingin mengajak anda latihan sejenak untuk menguji pemahaman seputar view serta materi-materi sebelumnya. Sebagai tabel sample, akan menggunakan tabel nilai_mahasiswa:

```
DROP TABLE IF EXISTS nilai_mahasiswa;

CREATE TABLE nilai_mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    semester_1 DECIMAL(4,2),
    semester_2 DECIMAL(4,2),
    semester_3 DECIMAL(4,2)
) ENGINE = InnoDB;
-- Query OK, 0 rows affected (0.24 sec)

INSERT INTO nilai_mahasiswa VALUES
    ('17090113', 'Riana Putria', 3.12, 2.98, 3.45),
    ('17140143', 'Rudi Permana', 2.56, 3.14, 3.22),
    ('17080305', 'Rina Kumala Sari', 3.45, 2.56, 3.67),
    ('17140119', 'Sandri Fatmala', 2.12, 2.78, 2.56),
    ('17090308', 'Christine Wijaya', 3.78, 3.23, 3.11);
-- Query OK, 5 rows affected (0.03 sec)
-- Records: 5 Duplicates: 0 Warnings: 0

SELECT * FROM nilai_mahasiswa;
+-----+-----+-----+-----+
| nim      | nama            | semester_1 | semester_2 | semester_3 |
+-----+-----+-----+-----+
| 17080305 | Rina Kumala Sari |      3.45 |      2.56 |      3.67 |
| 17090113 | Riana Putria   |      3.12 |      2.98 |      3.45 |
| 17090308 | Christine Wijaya |      3.78 |      3.23 |      3.11 |
| 17140119 | Sandri Fatmala  |      2.12 |      2.78 |      2.56 |
| 17140143 | Rudi Permana    |      2.56 |      3.14 |      3.22 |
+-----+-----+-----+-----+
-- 5 rows in set (0.00 sec)
```

Buatlah query untuk soal berikut:

Soal 1

Buat view nilai_ipk yang diambil dari tabel nilai_mahasiswa. View ini terdiri dari kolom nim, nama, dan IPK. Tampilan akhir view adalah sebagai berikut:

```
SELECT * FROM nilai_ipk;
+-----+-----+-----+
| nim      | nama            | IPK   |
+-----+-----+-----+
| 17080305 | Rina Kumala Sari | 3.23 |
| 17090113 | Riana Putria    | 3.18 |
| 17090308 | Christine Wijaya | 3.37 |
| 17140119 | Sandri Fatmala  | 2.49 |
| 17140143 | Rudi Permana    | 2.97 |
+-----+-----+-----+
```

Kolom IPK dalam kasus ini merupakan *kolom komputasi*, dimana nilainya akan dihitung dari kolom-kolom yang sudah ada. Kolom seperti ini sebaiknya tidak ikut disimpan ke dalam database, tapi cukup dihitung pada saat akan ditampilkan, atau disimpan ke dalam view.

Soal 2

Buat view kedua dengan nama `nilai_ipk_format`. Isi view ini diambil dari view `nilai_ipk`. Tampilan akhir view adalah sebagai berikut:

```
SELECT * FROM nilai_ipk_format;
+-----+
| Nama dan IPK          |
+-----+
| Christine Wijaya (3.37) |
| Rina Kumala Sari (3.23) |
| Riana Putria (3.18)     |
| Rudi Permana (2.97)    |
| Sandri Fatmala (2.49)  |
+-----+
```

Hasil diatas harus terurut dari IPK tertinggi ke terendah. Tips: gunakan fungsi `CONCAT()` untuk menggabungkan string, serta query `ORDER BY` untuk pengurutan.

Soal 3

Input 1 baris baru ke dalam tabel `nilai_mahasiswa`, kemudian periksa isi view `nilai_ipk` dan `nilai_ipk_format`. Data baru yang diinput boleh sembarang. Pastikan isi view juga langsung terupdate dengan penambahan data ini.

Soal 4

Tampilkan mahasiswa yang namanya diawali dengan huruf **R**. Data ini diambil dari view `nilai_ipk_format`. Berikut tampilan akhir yang diiginkan:

```
+-----+
| Nama dan IPK          |
+-----+
| Rina Kumala Sari (3.23) |
| Riana Putria (3.18)    |
| Rudi Permana (2.97)   |
+-----+
```

Tips: Gunakan query SELECT...LIKE untuk proses seleksi.

Soal 5

Hapus view nilai_ipk dan nilai_ipk_format.

24.7 Jawaban Latihan VIEW

Jawaban Soal 1

```
CREATE VIEW nilai_ipk AS
  SELECT
    nim,
    nama,
    ROUND(((semester_1 + semester_2 + semester_3)/3),2) AS 'IPK'
  FROM nilai_mahasiswa;
```

Jawaban Soal 2

```
CREATE VIEW nilai_ipk_format AS
  SELECT CONCAT(nama, ' (' , IPK, ')') AS 'Nama dan IPK'
  FROM nilai_ipk
  ORDER BY IPK DESC;
```

Jawaban Soal 3

```
INSERT INTO nilai_mahasiswa VALUES
('17140155', 'Tika Herina', 2.56, 2.77, 3.12);
```

```
SELECT * FROM nilai_ipk;
+-----+-----+-----+
| nim      | nama           | IPK   |
+-----+-----+-----+
| 17080305 | Rina Kumala Sari | 3.23 |
| 17090113 | Riana Putria    | 3.18 |
| 17090308 | Christine Wijaya | 3.37 |
| 17140119 | Sandri Fatmala   | 2.49 |
```

```
| 17140143 | Rudi Permana      | 2.97 |
| 17140155 | Tika Herina       | 2.82 |
+-----+-----+-----+
```

```
SELECT * FROM nilai_ipk_format;
+-----+
| Nama dan IPK          |
+-----+
| Christine Wijaya (3.37) |
| Rina Kumala Sari (3.23) |
| Riana Putria (3.18)     |
| Rudi Permana (2.97)    |
| Tika Herina (2.82)     |
| Sandri Fatmala (2.49)  |
+-----+
```

Jawaban Soal 4

```
SELECT * FROM nilai_ipk_format WHERE `Nama dan IPK` LIKE 'R%';
```

Jawaban Soal 5

```
DROP VIEW nilai_ipk;
DROP VIEW nilai_ipk_format;
```

Dalam bab ini kita telah mempelajari cara membuat **view** di dalam MySQL. **View** merupakan *tbl virtual* yang berisi hasil query **SELECT**.

Umumnya view dipakai untuk menyederhanakan query yang kompleks, menyembunyikan kolom tabel asli, membuat kolom komputasi (seperti kolom IPK dalam soal latihan), serta menjaga *backward compatibility* jika perlu menyediakan struktur database lama.

Berikutnya, kita akan masuk ke materi advanced selanjutnya, yakni **Stored Procedure**.

25. MySQL Stored Procedure dan Stored Function

Dalam bab kali ini akan dibahas tentang “programming” di dalam MySQL, yakni cara membuat **stored procedure** dan **stored function**. SQL sendiri pada dasarnya bukanlah sebuah bahasa pemrograman, tetapi bahasa kode yang mirip seperti HTML atau CSS.

Stored procedure dan **stored function** memungkinkan kita menulis kode program menggunakan bahasa SQL, termasuk diantaranya membuat kondisi **if else**, **case**, **perulangan while**, serta **perulangan repeat**. Materi seperti ini umum ditemui dalam bahasa pemrograman “asli” seperti PHP atau JavaScript.

Karena keterbatasan tempat, saya tidak bisa membahas semua hal tentang stored procedure dan stored function (yang sebenarnya cukup luas). Namun dengan materi yang ada, saya rasa sudah cukup untuk memahami cara kerja kedua fitur ini.

Untuk pembahasan yang lebih dalam, anda bisa kunjungi MySQL Reference Manual: [CREATE PROCEDURE](#) and [CREATE FUNCTION Syntax¹](#) atau [MySQL Stored Procedure²](#).



Karena materi tentang stored procedure dan stored function ini sangat berkaitan dengan programming, saya berasumsi anda sudah pernah belajar bahasa pemrograman seperti Pascal, C, C++, PHP atau JavaScript (cukup salah satu saja).

Apabila belum punya dasar programming, materi ini kali ini mungkin terasa sedikit sulit karena kita akan membahas seputar konsep variabel, kondisi if else, case, perulangan while serta perulangan repeat.

25.1 Pengertian Stored Procedure dan Stored Function

Stored procedure dan **stored function** adalah kode program bahasa SQL yang disimpan ke dalam database. Keduanya berfungsi sebagaimana *prosedur* dan *function* pada bahasa pemrograman umum.

Di dalam teori bahasa pemrograman komputer, **procedure** dan **function** adalah kumpulan kode program yang terpisah dari program utama. Procedure dan function disebut juga sebagai **subroutine**, yang dalam bahasa indonesia di *translate* sebagai *subrutin*.

Setiap *subroutine* akan menjalankan sebuah tugas tertentu. Berbagai subroutine nantinya digabung untuk menjadi program akhir. Inilah konsep dari apa yang dikenal sebagai **pemrograman prosedural**. Jenis lain adalah **pemrograman berbasis object**, dimana berbagai object saling bergabung menjadi program utama.

¹<https://dev.mysql.com/doc/refman/5.7/en/create-procedure.html>

²<http://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx>

Apa bedanya antara procedure dan function?

Dari beberapa bahasa pemrograman yang pernah saya gunakan, Pascal merupakan salah satu bahasa yang secara tegas memisahkan antara **procedure** dengan **function**. Di dalam PHP dan JavaScript yang ada hanya function saja, tidak ada procedure.

Dalam bahasa C dan C++ juga tidak ada procedure, yang tersedia juga hanya function. Walaupun, function yang tidak mengembalikan nilai (*void*) sering dianggap sebagai procedure.

Berikut perbedaan antara procedure dan function di dalam bahasa SQL:

- Procedure tidak harus mengembalikan sebuah nilai, sedangkan function harus mengembalikan nilai (melalui perintah RETURN).
- Procedure harus dipanggil menggunakan perintah CALL, sedangkan function bisa dipakai di dalam query seperti SELECT atau dalam kondisi WHERE.
- Procedure memiliki *input / output parameter*, sedangkan function hanya memiliki *input parameter* saja.

Seluruh perbedaan diatas akan kita praktekkan sepanjang bab ini.

25.2 Perlukah menggunakan Stored Procedure dan Stored Function?

Sama seperti **view**, **stored procedure** dan **stored function** tidak akan terpakai dalam project-project sederhana. Fitur ini baru cocok untuk aplikasi besar yang kompleks dan butuh pemrograman di sisi database. Selain itu, membuat program di sisi database punya pro dan kontra tersendiri.

Berikut kekurangan jika menggunakan **stored procedure** dan **stored function**:

Pengelolaan aplikasi menjadi lebih rumit

Banyak programmer berpendapat bahwa seharusnya programming itu cukup di bahasa pemrograman tempat database diakses saja, misalnya dari PHP. Database sebaiknya dipakai sebagai tempat menyimpan “data mentah”. Jika di dalam database juga ada pemrograman, pengelolaan aplikasi menjadi lebih rumit. Setidaknya harus ada yang ahli di PHP serta ahli di bahasa SQL.

Programming di SQL sangat terbatas

Programming di SQL juga tidak selengkap di PHP. Proses *debugging* (pencarian kesalahan) relatif lebih susah karena kode error di SQL tidak spesifik ke masalah yang terjadi. Umumnya error yang tampil adalah “ada yang kurang di baris sekian”, tapi tidak dijelaskan apa yang salah.

Butuh resources yang besar

Jatah *resources* (CPU dan RAM) untuk database server bisanya tidak sebesar web server. Penggunaan **stored procedure** dan **stored function** akan menambah beban pemrosesan di database server.

Dilain pihak, berikut keunggulan jika menggunakan **stored procedure** dan **stored function**:

Untuk jenis aplikasi tertentu, bisa lebih efisien

Jika database diakses dari berbagai bahasa pemrograman dan/atau dari banyak aplikasi, membuat programming di SQL akan lebih efisien. Setiap bahasa pemrograman tinggal mengakses data yang sudah jadi dari database (bukan data mentah lagi). Keuntungannya, kita tidak perlu membuat pemrosesan di masing-masing bahasa pemrograman.

Misalkan saya punya aplikasi desktop dan android untuk situs duniaIlkom. Jika data dari database hanya data “mentah”, seperti tanggal dalam format yyyy-mm-dd, maka di setiap aplikasi saya harus membuat kode program untuk mengubahnya menjadi dd-mm-yyyy. Namun jika dari database datanya sudah di format, setiap bahasa pemrograman tinggal menampilkan saja.

Mengurangi beban kerja dari web server

Jika anda memiliki *resources* yang besar untuk database server, maka programming di sisi SQL akan memindahkan beban web server ke database server.

Akan berguna jika menggunakan trigger

Stored procedure dan stored function akan dibutuhkan jika anda berencana menggunakan **trigger** (yang akan kita bahas dalam bab selanjutnya). Trigger merupakan fitur untuk menjalankan kode program secara otomatis ketika sebuah event terjadi.

Mempersiapkan tabel nilai_mahasiswa

Sebagai tabel sample untuk bab ini, saya kembali menggunakan tabel `nilai_mahasiswa`. Berikut query yang dipakai untuk membuat tabel ini:

```

DROP TABLE IF EXISTS nilai_mahasiswa;

CREATE TABLE nilai_mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    semester_1 DECIMAL(4,2),
    semester_2 DECIMAL(4,2),
    semester_3 DECIMAL(4,2)
) ENGINE = InnoDB;
-- Query OK, 0 rows affected (0.24 sec)

INSERT INTO nilai_mahasiswa VALUES
    ('17090113', 'Riana Putria', 3.12, 2.98, 3.45),
    ('17140143', 'Rudi Permana', 2.56, 3.14, 3.22),
    ('17080305', 'Rina Kumala Sari', 3.45, 2.56, 3.67),
    ('17140119', 'Sandri Fatmala', 2.12, 2.78, 2.56),
    ('17090308', 'Christine Wijaya', 3.78, 3.23, 3.11);
-- Query OK, 5 rows affected (0.03 sec)
-- Records: 5  Duplicates: 0  Warnings: 0

```

```
SELECT * FROM nilai_mahasiswa;
```

nim	nama	semester_1	semester_2	semester_3
17080305	Rina Kumala Sari	3.45	2.56	3.67
17090113	Riana Putria	3.12	2.98	3.45
17090308	Christine Wijaya	3.78	3.23	3.11
17140119	Sandri Fatmala	2.12	2.78	2.56
17140143	Rudi Permana	2.56	3.14	3.22

Seperti biasa, tabel `nilai_mahasiswa` ditempatkan ke dalam database `belajar`.

25.3 MySQL Variable

Sebelum kita masuk ke pembuatan *stored procedure* dan *stored function*, saya ingin membahas materi pendahuluan, yakni cara membuat variabel di dalam MySQL.

Sama seperti dalam bahasa pemrograman lain, **variabel** adalah tempat untuk menyimpan sebuah nilai. Di dalam MySQL, variabel ditandai dengan karakter ‘at’ (@). Variabel ini disebut sebagai **user-defined variables**, yakni variabel yang dibuat sendiri oleh user (kita), bukan bawaan dari MySQL.

User-defined variables berlaku per-session, artinya ketika koneksi dengan MySQL server diputus, isi variabel juga ikut terhapus.

Selain *user-defined variables*, terdapat juga variabel sistem bawaan MySQL yang ditandai dengan dua buah karakter at (@@). Contohnya seperti `@@lc_time_names` yang pernah kita gunakan untuk mengubah settingan waktu MySQL saat pembahasan fungsi `DATE_FORMAT()`.

Untuk mengisi nilai ke dalam variabel, gunakan perintah `SET` seperti contoh berikut:

```
SET @nama = 'andi';
-- Query OK, 0 rows affected (0.00 sec)
```

Disini saya membuat variabel `@nama` yang isinya berupa string ‘andi’. Untuk menampilkan isi dari variabel, bisa menggunakan perintah `SELECT`:

```
SELECT @nama;
+-----+
| @nama |
+-----+
| andi |
+-----+
```

Variabel di dalam MySQL bisa diisi dengan berbagai jenis data, termasuk number dan date:

```
SET @nilai = 3.14;
SET @tanggal := '18-08-17';

SELECT @nilai, @tanggal;
+-----+-----+
| @nilai | @tanggal |
+-----+-----+
| 3.14 | 18-08-17 |
+-----+-----+
```

Variabel juga bisa diisi dengan hasil perhitungan berbagai nilai, termasuk hasil dari function. Selain itu kita juga bisa menggunakan nilai yang tersimpan dari variabel lain:

```
SET @hasil = CEIL(@nilai) + 10;
SET @tanggal_format = DATE_FORMAT(@tanggal, '%W %M %Y');

SELECT @hasil, @tanggal_format;
+-----+-----+
| @hasil | @tanggal_format |
+-----+-----+
| 14 | Friday August 2018 |
+-----+-----+
```

Dalam contoh diatas, variabel @hasil akan menyimpan hasil perhitungan operasi `CEIL(@nilai) + 10.`

Variabel @nilai pada awalnya berisi angka 3.14 yang berasal dari perintah sebelumnya. Fungsi `CEIL(@nilai)` akan membulatkan angka 3.14 ke atas menjadi 4. Lalu angka 4 + 10 hasilnya 14. Angka 14 inilah yang disimpan ke dalam variabel @hasil.

Hal yang sama juga terjadi untuk variabel @tanggal_format, dimana saya menformat isi dari variabel @tanggal menggunakan fungsi `DATE_FORMAT()`. Variabel @tanggal sendiri sebelumnya berisi '18-08-17'.

Selain menggunakan perintah `SET`, variabel juga bisa di set dari perintah `SELECT`:

```
SELECT @nilai:= 1500;
+-----+
| @nilai:= 1500 |
+-----+
|      1500 |
+-----+
```

Jika variabel di set menggunakan perintah SELECT, operator yang digunakan adalah titik dua sama dengan (:=).

Isi dari variabel juga bisa dipakai ke dalam query, seperti contoh berikut:

```
SET @nilai_ip = 3.20;

SELECT * FROM nilai_mahasiswa WHERE semester_1 <= @nilai_ip;
+-----+-----+-----+-----+
| nim      | nama           | semester_1 | semester_2 | semester_3 |
+-----+-----+-----+-----+
| 17090113 | Riana Putria   |      3.12  |      2.98  |      3.45  |
| 17140119 | Sandri Fatmala |      2.12  |      2.78  |      2.56  |
| 17140143 | Rudi Permana   |      2.56  |      3.14  |      3.22  |
+-----+-----+-----+-----+
```

Disini saya mengisi variabel @nilai_ip dengan angka 3.20. Variabel ini selanjutnya dipakai untuk membuat kondisi WHERE. Hasilnya, akan tampil mahasiswa dengan nilai IP semester_1 yang kurang dari 3.20.

Kebalikan dari fungsi diatas, hasil dari perintah SELECT juga bisa disimpan ke dalam variabel. Tapi nilai yang bisa disimpan hanya nilai yang sederhana. Sebagai contoh, perhatikan query berikut ini:

```
SELECT COUNT(nama) FROM nilai_mahasiswa;
+-----+
| COUNT(nama) |
+-----+
|      5      |
+-----+
```

Saya ingin menyimpan hasil fungsi COUNT() diatas ke dalam sebuah variabel. Caranya, gunakan perintah SELECT... INTO:

```
SELECT COUNT(nama) INTO @jml_mhs FROM nilai_mahasiswa;
-- Query OK, 1 row affected (0.00 sec)

SELECT @jml_mhs;
+-----+
| @jml_mhs |
+-----+
|      5   |
+-----+
```

Perhatikan cara penulisan query SELECT diatas. Perintah `SELECT COUNT(nama) INTO @jml_mhs` artinya saya ingin menyimpan hasil dari fungsi `COUNT(nama)` ke dalam variabel `@jml_mhs`. Cara penyimpanan ke dalam variabel seperti ini akan sering dipakai ketika merancang *stored procedure* maupun *stored function*.

25.4 Format Dasar Stored Procedure dan Stored Function

Struktur penulisan *stored procedure* dan *stored function* di bahasa SQL mirip seperti bahasa pemrograman lain, terutama seperti bahasa PASCAL. Kita akan menggunakan perintah dalam bentuk ‘keyword’ seperti BEGIN, END, IF, atau END IF. Bukan tanda kurung kurawal { } yang sering ditemui dalam bahasa turunan C seperti PHP dan JavaScript.

Berikut format dasar penulisan *stored procedure* di dalam MySQL:

```
CREATE PROCEDURE procedure_name(parameter_1, parameter_2, . . .)
BEGIN
    .
    .
END
```

Setelah perintah `CREATE PROCEDURE`, diikuti dengan `procedure_name`, yakni nama dari procedure yang akan dibuat. Setiap procedure bisa diisi dengan beberapa **parameter** atau tidak sama sekali (opsional). Parameter ini ditulis setelah nama procedure. Isi dari procedure itu sendiri harus ditulis diantara perintah `BEGIN` dan `END`.

Untuk menjalankan stored procedure, harus melalui perintah `CALL`:

```
CALL procedure_name(argument_1, argument_2, . . .);
```

Jika pada saat pendefinisian procedure terdapat parameter, kita harus menyamakan jumlah parameter tersebut dengan jumlah argumen ketika memanggil procedure.

Sedangkan untuk **stored function**, berikut penulisan format dasarnya:

```
CREATE FUNCTION function_name(parameter_1, ...) RETURNS return_type
BEGIN
    ...
RETURN return_value
END//
```

Format penulisan stored function mirip seperti stored procedure. Hanya saja kali ini ditambahkan keyword RETURNS dan tipe data setelah penulisan parameter. Di dalam badan function, juga harus ditulis perintah RETURN yang berfungsi sebagai tempat mengembalikan nilai function.

Berbeda dengan stored procedure, stored function bisa dijalankan dari query SQL seperti SELECT atau di dalam kondisi WHERE:

```
SELECT function_name(argument_1, argument_2, ...);
```

Penulisan struktur dasar ini akan lebih jelas ketika sudah masuk ke contoh praktek yang akan kita lakukan sesaat lagi.

Juga karena isi dari stored procedure dan stored function mirip satu sama lain, saya akan membahas tentang stored procedure terlebih dahulu. Setelah itu baru masuk ke stored function di akhir bab ini.

Parameter vs Argumen

Dalam penjelasan diatas, terdapat istilah **argumen** dan **parameter**. Keduanya hampir sama, dengan sedikit perbedaan.

Parameter adalah sebutan untuk inputan subrutin (prosedur atau function) pada saat pendefinisian subrutin tersebut. Sedangkan **argumen** adalah sebutan untuk inputan subrutin pada saat pemanggilan.

Dalam penggunaan sehari-hari, kedua istilah ini sering dipertukarkan. Misalnya di dalam dokumentasi resmi PHP, istilah yang sering dipakai adalah argumen. Pada dasarnya, baik argumen maupun parameter merujuk ke data yang sama.

25.5 Perintah DELIMETER

Perintah DELIMETER sebenarnya bukan bagian dari stored procedure, akan tetapi query ini hampir selalu ditulis saat pembuatan stored procedure maupun stored function. Query DELIMETER digunakan untuk mengubah karakter penanda akhir query.

Secara default bawaan MySQL, tanda titik koma (;) dipakai untuk mengakhiri penulisan sebuah query sebagaimana yang kita gunakan selama ini. Akan tetapi tanda titik koma juga dipakai sebagai penanda akhir *statement* / baris perintah di dalam stored procedure. Padahal stored procedure itu sendiri seharusnya baru berakhir dengan perintah END.

Agar tidak bentrok, kita harus menukar fungsi penanda akhir query dari titik koma (;) menjadi karakter lain. Inilah fungsi dari perintah **DELIMITER**.

Berikut contoh penggunaannya:

```
MariaDB [belajar]> DELIMITER //
MariaDB [belajar]> SELECT 5+1;
-> ;
-> //
+----+
| 5+1 |
+----+
|   6 |
+----+

MariaDB [belajar]> SELECT 5+1 //
+----+
| 5+1 |
+----+
|   6 |
+----+

MariaDB [belajar]> SELECT * FROM nilai_mahasiswa //
+-----+-----+-----+-----+
| nim      | nama              | semester_1 | semester_2 | semester_3 |
+-----+-----+-----+-----+
| 17080305 | Rina Kumala Sari |      3.45 |      2.56 |      3.67 |
| ...      | ...                |      ... |      ... |      ... |
+-----+-----+-----+-----+

MariaDB [belajar]> DELIMITER ;
MariaDB [belajar]> SELECT 5+1;
+----+
| 5+1 |
+----+
|   6 |
+----+
```

Di baris pertama, saya menulis **DELIMITER //**. Artinya, karakter penanda akhir query sudah diubah menjadi karakter double slash (//), tidak lagi tanda titik koma (;).

Pada saat perintah **SELECT 5+1;** ditulis, tidak terjadi apa-apa. MySQL masih menunggu agar kita menginput perintah lain. Barulah ketika tanda // ditambahkan, query tersebut bisa diproses.

Begitu juga dengan perintah **SELECT * FROM nilai_mahasiswa //** yang langsung diproses karena query tersebut sudah diakhiri dengan karakter //.

Setiap baris setelah perintah `DELIMITER //` dijalankan, harus menggunakan tanda `//` untuk menutup baris tersebut (agar perintah itu bisa di proses oleh MySQL). Ini berlaku sepanjang session, selama tidak ada perintah `DELIMETER` lain.

Di dua baris terakhir, saya menulis `DELIMITER ;`. Perintah ini akan mengembalikan fungsi karakter titik koma `(;)` sebagai penanda akhir query, yakni kembali ke pengaturan normal.

Tanda akhir query juga tidak harus berupa double slash `(//)`, tapi bisa diganti dengan karakter lain. Karakter double dollar `($$)` juga sering dipakai sebagai penanda akhir query. Untuk yang seperti ini, perintahnya menjadi `DELIMITER $$`.

Dalam setiap pembuatan stored procedure dan stored function, saya akan selalu menulis perintah `DELIMETER //` di awal, dan `DELIMITER ;` di akhir untuk mengembalikan fungsi tanda titik koma sebagai penanda akhir perintah query.

25.6 Membuat Stored Procedure

Baik, mari masuk ke contoh pembuatan `stored procedure` di dalam MySQL:

```
DELIMITER //
CREATE PROCEDURE salam()
BEGIN
    SELECT 'Selamat pagi Indonesia';
END//
DELIMITER ;
```

Di baris pertama terdapat perintah `DELIMITER //`. Perintah ini akan menukar karakter penutup query dari titik koma `(;)` menjadi double slash `(//)`. Di baris terakhir, terdapat perintah `DELIMITER ;` yang akan mengembalikan fungsi karakter titik koma sebagai penanda akhir query.

Pada kode diatas, saya membuat procedure `salam()` yang isinya hanya satu baris, yakni `SELECT 'Selamat pagi Indonesia'`. Isi procedure diawali dengan perintah `BEGIN`, dan ditutup dengan perintah `END//`.

Perintah diatas baru dipakai untuk membuat atau mendefinisikan procedure `salam()`. Untuk menjalankannya, panggil dengan query `CALL`:

```
CALL salam();
+-----+
| Selamat pagi Indonesia |
+-----+
| Selamat pagi Indonesia |
+-----+
-- 1 row in set (0.03 sec)
```

Inilah contoh sederhana dari sebuah stored procedure .

Stored procedure disimpan secara permanen, artinya procedure `salam()` tetap bisa dipanggil meskipun MySQL client dan MySQL server di restart.

Stored procedure juga “melekat” ke database. Karena saya menggunakan database `belajar`, procedure `salam()` akan tersimpan ke dalam database ini. Aturan penamaan dari stored procedure sama seperti *identifier* lainnya, yakni sama seperti aturan penamaan nama tabel, nama database, dst.

Untuk melihat apa saja stored procedure yang ada di database `belajar`, bisa menggunakan query `SHOW PROCEDURE STATUS`.

```
SHOW PROCEDURE STATUS WHERE db = 'belajar' \G
*****
1. row *****
    Db: belajar
    Name: salam
    Type: PROCEDURE
    Definer: root@localhost
    Modified: 2017-12-01 17:10:24
    Created: 2017-12-01 17:10:24
    Security_type: DEFINER
    Comment:
character_set_client: cp850
collation_connection: cp850_general_ci
Database Collation: latin1_swedish_ci
-- 1 row in set (0.01 sec)
```

Disini akan terlihat nama database, nama procedure, pembuat procedure (*definer*), tanggal dibuat, serta *charset* dan *collation* yang digunakan. Jika kondisi `WHERE db = 'belajar'` tidak ditulis, maka akan tampil seluruh stored procedure dari semua database di dalam MySQL.

Untuk melihat dengan detail query yang dipakai dalam pembuatan procedure, bisa menggunakan perintah `SHOW CREATE PROCEDURE`:

```
SHOW CREATE PROCEDURE salam \G
*****
1. row *****
    Procedure: salam
    sql_mode: STRICT_ALL_TABLES,NO_AUTO_CREATE_USER, ...
    Create Procedure: CREATE DEFINER=`root`@`localhost` PROCEDURE `salam`() BEGIN
        SELECT 'Selamat pagi Indonesia';
    END
character_set_client: cp850
collation_connection: cp850_general_ci
Database Collation: latin1_swedish_ci
1 row in set (0.00 sec)
```

Sebagai contoh kedua, saya ingin membuat procedure `tampil_mhs()`. Procedure ini akan menampilkan seluruh data yang ada di dalam tabel `nilai_mahasiswa`:

```

DELIMITER $$

CREATE PROCEDURE tampil_mhs()
BEGIN
    SELECT * FROM nilai_mahasiswa;
END$$
DELIMITER ;

CALL tampil_mhs();
+-----+-----+-----+-----+
| nim | nama | semester_1 | semester_2 | semester_3 |
+-----+-----+-----+-----+
| 17080305 | Rina Kumala Sari | 3.45 | 2.56 | 3.67 |
| 17090113 | Riana Putria | 3.12 | 2.98 | 3.45 |
| 17090308 | Christine Wijaya | 3.78 | 3.23 | 3.11 |
| 17140119 | Sandri Fatmala | 2.12 | 2.78 | 2.56 |
| 17140143 | Rudi Permana | 2.56 | 3.14 | 3.22 |
+-----+-----+-----+-----+

```

Kali ini saya menggunakan tanda double dollar \$\$ sebagai DELIMETER. Isi procedure tampil_mhs() juga masih sederhana, yakni hanya berisi query `SELECT * FROM nilai_mahasiswa;`.

Untuk contoh yang sedikit rumit, saya akan membuat procedure `hitung_ipk()` yang berfungsi untuk menampilkan nama mahasiswa dari tabel `nilai_mahasiswa`, plus perhitungan nilai IPK dari setiap mahasiswa:

```

DELIMITER $$

CREATE PROCEDURE hitung_ipk()
BEGIN
    SELECT
        nim,
        nama,
        ROUND(((semester_1 + semester_2 + semester_3)/3),2) AS 'IPK'
    FROM nilai_mahasiswa;
END$$
DELIMITER ;

CALL hitung_ipk();
+-----+-----+-----+
| nim | nama | IPK |
+-----+-----+-----+
| 17080305 | Rina Kumala Sari | 3.23 |
| 17090113 | Riana Putria | 3.18 |
| 17090308 | Christine Wijaya | 3.37 |
| 17140119 | Sandri Fatmala | 2.49 |
| 17140143 | Rudi Permana | 2.97 |
+-----+-----+-----+

```

Cara pencarian rumus IPK ini sudah kita bahas pada bab sebelumnya, yakni dengan mencari rata-rata nilai semester_1 + semester_2 + semester_3.

25.7 Menghapus Stored Procedure

Di dalam MySQL, fungsi ALTER PROCEDURE tidak bisa dipakai untuk mengubah isi dari stored procedure. ALTER PROCEDURE hanya digunakan untuk mengubah COMMENT, LANGUAGE dan SQL SECURITY dari sebuah procedure.

Jika kita ingin mengubah isi procedure, harus dihapus dulu kemudian dibuat ulang. Untuk menghapus stored procedure, tersedia query DROP PROCEDURE dengan format sebagai berikut:

```
DROP PROCEDURE [IF EXISTS] procedure_name;
```

Sebagai contoh, saya ingin menghapus ketiga procedure yang sudah kita buat sebelumnya:

```
DROP PROCEDURE salam;
DROP PROCEDURE tampil_mhs;
DROP PROCEDURE hitung_ipk;
-- Query OK, 0 rows affected (0.06 sec)
```

Sekarang, ketiga procedure sudah terhapus.



Stored procedure butuh penulisan yang cukup panjang. Oleh karena itu sebaiknya tulis terlebih dahulu di teks editor seperti Notepad++, lalu di copy paste ke dalam cmd Windows (MySQL client). Klik kanan di cmd windows untuk men-paste kode SQL.

25.8 Stored Procedure Variable

Di awal bab ini, kita telah pelajari cara membuat variabel di dalam MySQL. Khusus untuk stored procedure, terdapat cara lain. Untuk membuat **stored procedure variable**, gunakan perintah DECLARE dengan format dasar sebagai berikut:

```
DECLARE variable_name datatype(size) [DEFAULT default_value];
```

Setelah perintah DECLARE, diikuti dengan nama variabel serta tipe data dari variabel tersebut. Tipe data ini bersesuaian dengan tipe data kolom yang ada di dalam MySQL, seperti INT, VARCHAR, atau DATE. Variabel yang di deklarasikan di dalam stored procedure, hanya bisa diakses di dalam stored procedure tersebut.

Secara bawaan, variabel bernilai NULL. Untuk mengganti nilai default ini, bisa ditambahkan perintah DEFAULT dan diikuti dengan nilai awal untuk variabel.

Berikut contoh pembuatan dan penggunaan variabel di dalam stored procedure:

```

DROP PROCEDURE IF EXISTS salam;

DELIMITER //
CREATE PROCEDURE salam()
BEGIN
    DECLARE selamat_pagi VARCHAR(30) DEFAULT 'Selamat Pagi Indonesia';
    SELECT selamat_pagi;
END//
DELIMITER ;

CALL salam();
+-----+
| selamat_pagi      |
+-----+
| Selamat Pagi Indonesia |
+-----+

```

Di awal kode program, saya menulis query `DROP PROCEDURE IF EXISTS salam`. Perintah ini dipakai untuk menghapus procedure `salam()` jika sudah didefinisikan sebelumnya.

Dalam procedure `salam()`, saya membuat variabel `selamat_pagi` dengan tipe data `VARCHAR(30)`. Variabel ini langsung diisi dengan nilai default berupa string 'Selamat Pagi Indonesia'.

Di baris berikutnya, isi variabel `selamat_pagi` ditampilkan menggunakan perintah `SELECT selamat_pagi`. Hasilnya, akan tampil string 'Selamat Pagi Indonesia' ketika procedure `salam()` dipanggil.

Sebagaimana layaknya variabel, nilai di dalam variabel bisa diubah sepanjang kode program. Untuk keperluan ini tersedia perintah `SET`, dengan format sebagai berikut:

```
SET variable_name = value;
```

Mari kita lihat contoh prakteknya:

```

DROP PROCEDURE IF EXISTS salam;

DELIMITER //
CREATE PROCEDURE salam()
BEGIN
    DECLARE selamat_pagi VARCHAR(30) DEFAULT 'Selamat Pagi Indonesia';
    SET selamat_pagi = 'Selamat Pagi Dunia';
    SELECT selamat_pagi;
END//
DELIMITER ;

CALL salam();
+-----+

```

```
| selamat_pagi      |
+-----+
| Selamat Pagi Dunia |
+-----+
```

Procedure `salam()` diatas mirip seperti sebelumnya. Hanya saja kali ini terdapat perintah `SET` untuk mengubah isi variabel `selamat_pagi` menjadi 'Selamat Pagi Dunia'. Dengan perintah ini, string default dari variabel `selamat_pagi` akan tertimpas dengan nilai yang baru.

Sebagai contoh selanjutnya, saya ingin membuat procedure `cari_ip()`. Procedure ini berfungsi untuk menampilkan daftar mahasiswa dengan nilai IP tertentu. Nilai IP ini akan disimpan ke dalam variabel terlebih dahulu:

```
DELIMITER $$  
CREATE PROCEDURE cari_ip()  
BEGIN  
    DECLARE minimum_ip DECIMAL(4,2) DEFAULT 0.00;  
    SET minimum_ip = 3.00;  
  
    SELECT nama, semester_1  
    FROM nilai_mahasiswa  
    WHERE semester_1 >= minimum_ip;  
END$$  
DELIMITER ;  
  
CALL cari_ip();  
+-----+-----+  
| nama          | semester_1 |  
+-----+-----+  
| Rina Kumala Sari |      3.45 |  
| Riana Putria   |      3.12 |  
| Christine Wijaya |      3.78 |  
+-----+-----+
```

Disini saya membuat variabel `minimum_ip` dengan tipe data `DECIMAL(4,2)`. Variabel ini kemudian diisi dengan angka 3.00 dan digunakan di dalam kondisi `WHERE` dari query `SELECT`. Hasilnya, akan tampil semua mahasiswa yang memiliki IP `semester_1` yang lebih besar atau sama dengan 3.00.

25.9 Stored Procedure Parameter

Parameter adalah sebutan untuk nilai input ke dalam sebuah procedure. Parameter ini ditulis di dalam tanda kurung setelah nama procedure. Sebuah procedure bisa memiliki beberapa parameter, atau tidak sama sekali. Contoh procedure yang kita tulis sebelum ini tidak ada yang menggunakan procedure.

Sebagai pengingat, berikut format dasar penempatan sebuah parameter:

```
CREATE PROCEDURE procedure_name(parameter_1, parameter_2, . . . )
BEGIN
    . . .
    . . .
END//
```

Parameter itu sendiri harus ditulis dengan tipe datanya, kurang lebih sama seperti pendefenisian variabel:

```
[IN|OUT|INOUT] parameter_name parameter_type(parameter_size)
```

Di awal penulisan parameter terdapat pilihan opsional **mode parameter**, yakni IN, OUT atau INOUT. Perbedaannya akan kita bahas sesaat lagi. Mode parameter ini tidak harus ditulis. Apabila mode parameter tidak ditemukan, maka akan dianggap sebagai IN.

Berikutnya adalah penulisan nama parameter yang diikuti dengan tipe data parameter seperti INT, VARCHAR atau DATE.

Sebagai contoh praktik, saya akan membuat ulang procedure `salam()` dengan penambahan parameter:

```
DROP PROCEDURE IF EXISTS salam;

DELIMITER //
CREATE PROCEDURE salam(siapa VARCHAR(30))
BEGIN
    SELECT CONCAT('Selamat Pagi ', siapa);
END//
DELIMITER ;

CALL salam('Bandung');
-- Selamat Pagi Bandung

CALL salam('Medan');
-- Selamat Pagi Medan

CALL salam('Menado');
-- Selamat Pagi Menado
```

Sekarang procedure `salam()` memiliki sebuah parameter `siapa`. Parameter ini bertipe string yang nantinya akan menjadi inputan untuk fungsi `CONCAT('Selamat Pagi ', siapa)`.

Karena procedure `salam()` sekarang memiliki 1 parameter, pemanggilan fungsi ini juga harus diinput dengan 1 argumen.

Pada saat perintah `CALL salam('Bandung')` dijalankan, string 'Bandung' akan dikirim ke dalam parameter `siapa`. Hasilnya tampil string `Selamat Pagi Bandung`, hasil ini berasal dari perintah `SELECT CONCAT('Selamat Pagi ', siapa)` yang terdapat di dalam procedure `salam()`.

Dengan mengubah isi argumen pada saat pemanggilan procedure `salam()`, hasil yang ditampilkan juga akan berbeda-beda.

3 Mode Parameter: IN, OUT dan INOUT

Berikutnya, mari kita bahas 3 mode parameter untuk stored procedure, yakni IN, OUT dan INOUT:

- Jika parameter di definisikan sebagai IN, sebuah parameter akan berfungsi untuk menampung nilai masukan (**input**). IN adalah mode parameter default. Jika mode parameter tidak ditulis, mode inilah yang akan dipakai. Perubahan nilai parameter IN tidak akan berdampak dan tidak bisa diakses di luar procedure.
- Jika parameter di definisikan sebagai OUT, sebuah parameter akan berfungsi untuk penampung nilai hasil (**output**). Jika nilai parameter ini diubah, perubahan itu akan bisa diakses dari luar procedure.
- Jika parameter di definisikan sebagai INOUT, sebuah parameter akan berfungsi untuk penampung nilai **input** dan **output** sekaligus. Kita bisa mengirim nilai ke dalam parameter ini, dan hasilnya juga bisa diakses dari luar procedure.

Berikut contoh dari penggunaan mode parameter IN dan OUT:

```
DROP PROCEDURE IF EXISTS salam;

DELIMITER //
CREATE PROCEDURE salam(IN siapa VARCHAR(30), OUT hasil VARCHAR(30))
BEGIN
    SELECT CONCAT('Selamat Pagi ',siapa) INTO hasil;
END//
DELIMITER ;

CALL salam('Jakarta',@hasil_salam);
-- Query OK, 1 row affected (0.00 sec)

SELECT @hasil_salam;
-- Selamat Pagi Jakarta
```

Kembali, saya mendefinisikan ulang procedure `salam()`. Kali ini procedure `salam()` memiliki 2 buah parameter. Satu sebagai parameter IN, dan satu lagi sebagai parameter OUT.

Parameter IN di definisikan sama seperti sebelumnya, yakni `siapa` dengan tipe `VARCHAR(30)`. Sedangkan parameter OUT di definisikan sebagai `hasil` `VARCHAR(30)`. Konsepnya, parameter `hasil` ini akan diisi dengan sebuah string dari dalam procedure `salam()`.

Proses pengisian parameter `hasil` dilakukan dengan perintah `SELECT CONCAT('Selamat Pagi ',siapa) INTO hasil`. Teknik ini sama seperti pengisian **user defined variable** yang kita bahas di awal bab.

Cara pemanggilan procedure ini juga berbeda, yakni `CALL salam('Jakarta',@hasil_salam)`. Argumen pertama, yakni '`Jakarta`' akan dikirim sebagai nilai input untuk parameter `siapa`.

Argumen kedua, berupa sebuah variabel `@hasil_salam` yang nantinya akan menampung hasil dari parameter `hasil`. Setelah procedure `salam()` dipanggil, nilainya akan dikirim ke dalam

variabel @hasil_salam, yakni berupa string Selamat Pagi Jakarta. Inilah fungsi dari mode parameter OUT.

Untuk contoh mode parameter INOUT, silahkan pelajari sejenak kode program berikut ini:

```
DELIMITER //
CREATE PROCEDURE tambah_satu(INOUT hasil INT)
BEGIN
    SET hasil = hasil + 1;
END//
DELIMITER ;

SET @angka= 1;

CALL tambah_satu(@angka);
CALL tambah_satu(@angka);
CALL tambah_satu(@angka);

SELECT @angka;
-- 4
```

Kode program diatas akan terasa cukup rumit terutama jika anda belum pernah mempelajari cara pembuatan function di dalam bahasa pemrograman lain.

Disini saya membuat procedure tambah_satu() dengan sebuah parameter hasil INT. Parameter ini diset sebagai INOUT, yang artinya bisa sebagai nilai input dan nilai output sekaligus.

Isi dari procedure tambah_satu() hanya 1 baris, yakni SET hasil = hasil + 1. Konsep dari procedure ini adalah, setiap kali procedure tambah_satu() dijalankan, isi parameter hasil bertambah 1 angka.

Pemanggilan procedure tambah_satu() juga perlu cara khusus. Saya membuat sebuah variabel @angka yang diisi dengan angka 1. Variabel @angka ini dipakai sebagai argumen pada saat pemanggilan procedure tambah_satu().

Ketika perintah CALL tambah_satu(@angka) dijalankan pertama kali, nilai 1 ini akan dikirim ke dalam parameter hasil. Di dalam procedure, akan dijalankan perintah SET hasil = hasil + 1. Hasilnya, parameter set akan bernilai 2.

Nilai 2 ini dikembalikan lagi ke dalam variabel @angka karena parameter hasil diset sebagai INOUT. Artinya, variabel @angka sekarang bernilai 2.

Pada saat perintah CALL tambah_satu(@angka) dijalankan lagi, nilai 2 akan dikirim ke dalam parameter hasil, dan proses yang sama akan berulang.

Karena terdapat 3 kali pemanggilan procedure tambah_satu(@angka), setiap pemanggilan akan menaikkan nilai variabel @angka. Hasilnya, variabel @angka akan berisi angka 4.

Konsep parameter INOUT ini mirip seperti konsep static variable di dalam bahasa pemrograman PHP.

Menutup pembahasan tentang procedure parameter, saya ingin membuat ulang procedure `cari_ip()`. Kali ini dengan penambahan sebuah parameter untuk menampung nilai IP yang akan dicari. Berikut modifikasinya:

```
DROP PROCEDURE IF EXISTS cari_ip;

DELIMITER $$
CREATE PROCEDURE cari_ip(minimum_ip DECIMAL(4,2))
BEGIN
    SELECT nama, semester_1
    FROM nilai_mahasiswa
    WHERE semester_1 >= minimum_ip;
END$$
DELIMITER ;

CALL cari_ip(3.5);
+-----+
| nama          | semester_1 |
+-----+
| Christine Wijaya |      3.78 |
+-----+

CALL cari_ip(2.5);
+-----+
| nama          | semester_1 |
+-----+
| Rina Kumala Sari |      3.45 |
| Riana Putria   |      3.12 |
| Christine Wijaya |      3.78 |
| Rudi Permana   |      2.56 |
+-----+
```

Sekarang procedure `cari_ip()` jadi lebih fleksibel. Jika dipanggil `CALL cari_ip(2.5)`, hasilnya akan menampilkan seluruh data mahasiswa dengan nilai IP minimal 2.5.

25.10 Stored Procedure IF Statement

Sebagaimana layaknya sebuah bahasa pemrograman, SQL menyediakan fitur kondisi **IF**, **IF ELSE** dan **IF ELSEIF**. Semua perintah ini digunakan untuk membuat percabangan kode program.

Kita akan bahas tentang **IF** terlebih dahulu. Berikut format dasar penulisannya:

```
IF expression THEN
    statements;
END IF;
```

Format penulisan kondisi IF ini mirip seperti yang digunakan oleh bahasa PASCAL. Berikut contoh prakteknya:

```
DROP PROCEDURE IF EXISTS salam;

DELIMITER //
CREATE PROCEDURE salam(waktu TIME)
BEGIN
    IF waktu < '10:00:00' THEN
        SELECT 'Selamat Pagi';
    END IF;
END//
DELIMITER ;

CALL salam('08:30:00');
-- Selamat Pagi

CALL salam('00:00:01');
-- Selamat Pagi

CALL salam('10:30:00');
-- Query OK, 0 rows affected (0.00 sec)
```

Saya membuat ulang procedure `salam()`. Kali ini procedure `salam()` memiliki 1 parameter `waktu` bertipe DATE.

Di dalam procedure, isi dari parameter `waktu` akan diperiksa menggunakan kondisi IF. Jika waktu tersebut berisi nilai kurang dari '10:00:00', jalankan perintah `SELECT 'Selamat Pagi'`.

Hasilnya, ketika procedure `salam()` dipanggil dengan `CALL salam('08:30:00')` dan `CALL salam('00:00:01')`, akan tampil string 'Selamat Pagi'.

Namun jika yang dijalankan `CALL salam('10:30:00')`, tidak ada tampilan apa-apa karena nilai ini tidak memenuhi syarat kondisi IF `waktu < '10:00:00'`.

25.11 Stored Procedure IF ELSE Statement

Percabangan kedua dari kondisi IF adalah IF ELSE. Disini kita bisa membuat kondisi lain seandainya kondisi IF tidak terpenuhi. Berikut format dasar penulisannya:

```
IF expression THEN
    statements_1;
ELSE
    statements_2;
END IF;
```

Berikut modifikasi dari procedure `salam()` sebelumnya dengan penambahan kondisi IF ELSE:

```
DROP PROCEDURE IF EXISTS salam;

DELIMITER //
CREATE PROCEDURE salam(waktu TIME)
BEGIN
    IF waktu < '10:00:00' THEN
        SELECT 'Selamat Pagi';
    ELSE
        SELECT 'Selamat Beraktifitas';
    END IF;
END//
DELIMITER ;

CALL salam('08:30:00');
-- Selamat Pagi

CALL salam('10:30:00');
-- Selamat Beraktifitas

CALL salam('23:59:59');
-- Selamat Beraktifitas
```

Kali ini ketika waktu yang diinput tidak sesuai dengan syarat `IF waktu < '10:00:00'`, yang dijalankan adalah bagian ELSE. Dalam contoh diatas, pemanggilan `CALL salam('10:30:00')` dan `CALL salam('23:59:59')` akan menampilkan string `Selamat Beraktifitas` karena sudah lebih dari '10:00:00'.

25.12 Stored Procedure IF ELSEIF ELSE Statement

Bentuk ketiga dari kondisi IF adalah IF ELSEIF ELSE. Struktur percabangan ini adalah rangkaian dari berbagai kondisi IF ELSE. Berikut format dasarnya:

```

IF expression THEN
    statements_1;
ELSEIF
    statements_2;
ELSE
    statements_3;
END IF;

```

Saya akan merancang ulang procedure `salam()` agar bisa menangani berbagai waktu yang diinput:

```

DROP PROCEDURE IF EXISTS salam;

DELIMITER //
CREATE PROCEDURE salam(waktu TIME)
BEGIN
    IF waktu < '10:00:00' THEN
        SELECT 'Selamat Pagi';
    ELSEIF waktu < '15:00:00' THEN
        SELECT 'Selamat Siang';
    ELSEIF waktu < '18:00:00' THEN
        SELECT 'Selamat Sore';
    ELSEIF waktu < '24:00:00' THEN
        SELECT 'Selamat Malam';
    ELSE
        SELECT 'Selamat Beraktifitas';
    END IF;
END//
DELIMITER ;

CALL salam('08:30:00');
-- Selamat Pagi

CALL salam('10:30:00');
-- Selamat Siang

CALL salam('20:49:30');
-- Selamat Malam

CALL salam('25:00:30');
-- Selamat Beraktifitas

```

Rangkaian kondisi IF ELSEIF ELSE diatas akan menampilkan sapaan salam untuk berbagai jenis waktu.

Jika procedure `salam()` dijalankan dengan nilai input waktu kurang dari ‘10:00:00’, akan tampil string `Selamat Pagi`, jika kondisi ini tidak terpenuhi maka akan lanjut ke pemeriksaan kedua.

Jika procedure salam dijalankan dengan argumen waktu kurang dari ‘15:00:00’, akan tampil string Selamat Siang, jika kondisi ini tidak terpenuhi, akan lanjut ke pemeriksaan ketiga dan seterusnya.

Jika waktu yang diinput tidak valid, hal itu akan ditangani oleh bagian ELSE terakhir. Misalnya diinput waktu ‘25:00:30’, maka hasilnya adalah Selamat Beraktifitas.

25.13 Stored Procedure Simple CASE Statement

CASE statement adalah bentuk lain dari percabangan kondisi IF ELSEIF ELSE. Biasanya, struktur percabangan CASE hanya cocok untuk kondisi yang sederhana. Akan tetapi bahasa SQL juga mendukung kondisi CASE kompleks dengan format yang sedikit berbeda.

Kali ini akan kita bahas Simple CASE Statement, yakni struktur CASE yang sederhana. Berikut format dasarnya:

```
CASE case_expression
    WHEN when_expression_1 THEN statements_1
    WHEN when_expression_2 THEN statements_2
    ...
    ELSE statements_3
END CASE;
```

Berikut contoh praktek dari simple CASE statement:

```
DROP PROCEDURE IF EXISTS penilaian;

DELIMITER //
CREATE PROCEDURE penilaian(nilai_ip INT)
BEGIN
    CASE nilai_ip
        WHEN 1 THEN
            SELECT 'Serius kuliah g sih?';
        WHEN 2 THEN
            SELECT 'Kebanyakan main';
        WHEN 3 THEN
            SELECT 'Berusaha lagi';
        WHEN 4 THEN
            SELECT 'Mantap, pertahankan!';
        ELSE
            SELECT 'Nilai tidak sesuai';
    END CASE;
END//
DELIMITER ;
```

```
CALL penilaian (1);
-- Serius kuliah g sih?

CALL penilaian (2);
-- Kebanyakan main

CALL penilaian (4);
-- Mantap, pertahankan!

CALL penilaian (5);
-- Nilai tidak sesuai
```

Saya membuat procedure `penilaian()`. Procedure ini akan menampilkan string berdasarkan nilai angka. Setiap kondisi CASE akan memeriksa apakah nilai yang diinput 1, 2, 3, 4 atau yang lain, lalu menampilkan string yang sesuai dengan kondisi tersebut.

Simple case statement hanya mendukung pengecekan kondisi yang sederhana, yakni apakah isi nilai tersebut sama atau tidak. Kondisi yang diperiksa harus sama persis dengan nilai input.

25.14 Stored Procedure Complex CASE Statement

Struktur berikutnya masih bernama CASE statement. Namun berbeda dengan simple CASE statement, kita bisa membuat kondisi yang lebih kompleks, misalnya menggunakan operator perbandingan atau perpaduan kondisi menggunakan AND dan OR.

Dalam bahasa pemrograman umum, kondisi yang kompleks seperti ini hanya bisa ditangani dengan kondisi IF ELSE.

Berikut format dasar dari struktur Complex CASE statement:

```
CASE
    WHEN condition_1 THEN statements_1
    WHEN condition_2 THEN statements_2
    ...
    ELSE statements_3
END CASE;
```

Terdapat sedikit perbedaan cara penulisan dari simple CASE. Disini pengecekan kondisi dilakukan di setiap baris WHEN.

Berikut contoh praktek modifikasi dari procedure `penilaian()` sebelumnya:

```
DROP PROCEDURE IF EXISTS penilaian;

DELIMITER //
CREATE PROCEDURE penilaian(nilai_ip DECIMAL(4,2))
BEGIN
    CASE
        WHEN (nilai_ip <= 1.00) THEN
            SELECT 'Serius kuliah g sih?';
        WHEN (nilai_ip <= 2.00) THEN
            SELECT 'Kebanyakan main';
        WHEN (nilai_ip <= 3.00) THEN
            SELECT 'Berusaha lagi';
        WHEN (nilai_ip <= 4.00) THEN
            SELECT 'Mantap, pertahankan!';
        ELSE
            SELECT 'Nilai tidak sesuai';
    END CASE;
END//
DELIMITER ;

CALL penilaian (1.01);
-- Kebanyakan main

CALL penilaian (3.56);
-- Mantap, pertahankan!

CALL penilaian (4.00);
-- Mantap, pertahankan!

CALL penilaian (4.01);
-- Nilai tidak sesuai

CALL penilaian (-1.00);
-- Serius kuliah g sih?
```

Kali ini procedure penilaian() bisa memproses jangkauan angka karena pemeriksaan kondisi bisa memakai operator perbandingan.

Sebagai bahan latihan, bisakah anda memodifikasi procedure penilaian() diatas agar menampilkan pesan Nilai tidak sesuai jika angka yang diinput kurang dari 0? Untuk kondisi seperti ini, kita harus menggabungkan beberapa kondisi dalam 1 pemeriksaan CASE.

Berikut salah satu solusi untuk hal ini:

```

DROP PROCEDURE IF EXISTS penilaian;

DELIMITER //
CREATE PROCEDURE penilaian(nilai_ip DECIMAL(4,2))
BEGIN
    CASE
        WHEN ((nilai_ip >= 0.00) AND (nilai_ip <= 1.00)) THEN
            SELECT 'Serius kuliah g sih?';
        WHEN ((nilai_ip > 1.00) AND (nilai_ip <= 2.00)) THEN
            SELECT 'Kebanyakan main';
        WHEN ((nilai_ip > 2.00) AND (nilai_ip <= 3.00)) THEN
            SELECT 'Berusaha lagi';
        WHEN ((nilai_ip > 3.00) AND (nilai_ip <= 4.00)) THEN
            SELECT 'Mantap, pertahankan!';
        ELSE
            SELECT 'Nilai tidak sesuai';
    END CASE;
END//
DELIMITER ;

CALL penilaian (-1.00);
-- Nilai tidak sesuai

CALL penilaian (7.00);
-- Nilai tidak sesuai

CALL penilaian (0.00);
-- Serius kuliah g sih?

CALL penilaian (4.00);
-- Mantap, pertahankan!

```

Dalam contoh diatas, saya menggunakan 2 buah kondisi dalam setiap CASE. Dua buah kondisi ini digabung dengan operator AND.

Kondisi WHEN ((nilai_ip >= 0.00) AND (nilai_ip <= 1.00)) hanya bisa terpenuhi jika nilai_ip berisi angka lebih atau sama dengan 0 hingga kurang dari atau sama dengan 1. Artinya jangkauan angka yang sesuai adalah 0 - 1.

Logika penggabungan kondisi seperti ini akan lebih mudah dipahami jika anda sudah berpengalaman dengan teknik-teknik algoritma dalam bahasa pemrograman.

25.15 Stored Procedure WHILE Loop

Perulangan atau loop merupakan salah satu fitur yang selalu hadir di setiap bahasa pemrograman. SQL juga menyediakan WHILE dan REPEAT loop. Keduanya dipakai untuk menjalankan perintah secara berulang.

Berikut format dasar dari perulangan **WHILE** di dalam MySQL:

```
WHILE expression DO
    statements
    [counter]
END WHILE
```

Kurang lebih mirip seperti perulangan WHILE dalam bahasa pemrograman lain. Berikut contoh penggunaannya:

```
DROP PROCEDURE IF EXISTS faktorial;

DELIMITER //
CREATE PROCEDURE faktorial(nilai SMALLINT)
BEGIN
    DECLARE i SMALLINT DEFAULT 1;
    DECLARE hasil INT DEFAULT 1;
    WHILE i <= nilai DO
        SET hasil = hasil * i;
        SET i = i + 1;
    END WHILE;
    SELECT hasil;
END//
DELIMITER ;

CALL faktorial(3);
-- 6

CALL faktorial(5);
-- 120

CALL faktorial(8);
-- 40320
```

Saya merancang procedure `faktorial()` yang berfungsi untuk menghitung hasil faktorial dari sebuah angka. Dalam matematika, faktorial adalah perkalian dari setiap angka mulai dari $1 * 2 * 3 * \dots$, dst. Angka faktorial biasa ditulis dengan tanda seru (!).

Sebagai contoh, $5!$ dihitung sebagai $1 * 2 * 3 * 4 * 5 = 120$, dan $8!$ sebagai $1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 = 40320$.

Ketika procedure `CALL faktorial(5)` dijalankan, argumen berupa angka 5 akan diisi ke dalam parameter `nilai` dari procedure `faktorial()`. Di dalam procedure, saya membuat dua buah variabel: `i` dan `hasil`. Variabel `i` dipakai sebagai **variabel counter**, yang akan menjadi penanda akhir perulangan. Sedangkan variabel `hasil` berfungsi untuk menampung hasil perhitungan.

Perintah `WHILE i <= nilai DO` akan selalu memeriksa apakah isi variabel `i` saat ini kurang atau sama dengan `nilai`. Jika tidak, perintah `SET hasil = hasil * i` akan diproses, kemudian nilai

variabel i dinaikkan 1 angka dengan perintah `SET i = i + 1`. Proses ini akan terus berlangsung hingga perintah `WHILE i <= nilai DO` tidak sesuai lagi.

Secara bertahap, nilai variabel `hasil` akan menampung perhitungan $1 * 2 * 3$, dst.

25.16 Stored Procedure REPEAT Loop

Bentuk looping atau perulangan kedua adalah **REPEAT**. Konsep kerjanya sedikit berbeda dengan perulangan **WHILE**. Di dalam perulangan **REPEAT**, pemeriksaan kondisi dilakukan di akhir perulangan, bukan di awal seperti di dalam **WHILE**.

Berikut format dasar perulangan **REPEAT** di dalam MySQL:

```
REPEAT
  statements
  [counter]
  UNTIL expression
END REPEAT
```

Berikut modifikasi dari procedure `faktorial()` sebelumnya, tapi kali ini menggunakan **REPEAT**:

```
DROP PROCEDURE IF EXISTS faktorial;

DELIMITER //
CREATE PROCEDURE faktorial(nilai SMALLINT)
BEGIN
  DECLARE i SMALLINT DEFAULT 1;
  DECLARE hasil INT DEFAULT 1;
  REPEAT
    SET hasil = hasil * i;
    SET i = i + 1;
    UNTIL i > nilai
  END REPEAT;
  SELECT hasil;
END//
DELIMITER ;

CALL faktorial(3);
-- 6

CALL faktorial(5);
-- 120

CALL faktorial(8);
-- 40320
```

Silahkan anda bandingkan cara penulisan procedure() ini dengan versi sebelumnya, yakni antara perulangan REPEAT dengan WHILE.

25.17 Stored Function

Setelah membahas konsep programming menggunakan procedure, saatnya kita pelajari cara membuat **stored function**.

Perbedaan mendasar antara stored procedure dengan stored function adalah, stored function harus mengembalikan sebuah nilai. Di dalam stored function, harus ada perintah RETURN. Sedangkan di dalam stored procedure, tidak perlu mengembalikan sebuah nilai.

Berikut format dasar penulisan stored function di dalam MySQL:

```
CREATE FUNCTION function_name(parameter_1, . . .) RETURNS return_type
BEGIN
    .
    .
    .
    RETURN return_value
END//
```

Untuk menjalankan stored function, kita tidak lagi menggunakan perintah CALL, tapi bisa menggunakan query biasa seperti SELECT atau di dalam kondisi WHERE. Cara pemanggilan stored function ini sama seperti cara menjalankan function bawaan MySQL yang sudah kita bahas dalam beberapa bab sebelum ini.

Berikut contoh praktek pembuatan stored function:

```
DROP FUNCTION IF EXISTS tambah;

DELIMITER //
CREATE FUNCTION tambah(nilai_1 INT, nilai_2 INT) RETURNS INT
BEGIN
    RETURN nilai_1 + nilai_2;
END//
DELIMITER ;

SELECT tambah(10,5);
-- 15

SELECT tambah(20,100);
-- 120
```

Kali ini saya membuat sebuah function tambah(). Function ini sangat sederhana, isinya akan menambahkan dua buah angka yang diinput ke dalam dua parameter, lalu mengembalikan hasil dari penambahan tersebut.

Perhatikan setelah penulisan parameter, terdapat perintah RETURNS INT. Artinya, fungsi tambah() akan mengembalikan sebuah nilai bertipe data INT.

Isi dari function tambah() hanya satu baris, yakni RETURN nilai_1 + nilai_2. Perintah ini akan mengembalikan nilai hasil penambahan parameter nilai_1 + nilai_2.

Salah satu keunggulan dari stored function adalah, kita bisa memanggilnya di dalam query apa saja, seperti contoh berikut:

```
SELECT * FROM nilai_mahasiswa WHERE semester_2 < tambah(1,2);
+-----+-----+-----+-----+
| nim      | nama            | semester_1 | semester_2 | semester_3 |
+-----+-----+-----+-----+
| 17080305 | Rina Kumala Sari |      3.45 |      2.56 |      3.67 |
| 17090113 | Riana Putria    |      3.12 |      2.98 |      3.45 |
| 17140119 | Sandri Fatmala   |      2.12 |      2.78 |      2.56 |
+-----+-----+-----+-----+
```

Disini saya menggunakan function tambah() di dalam kondisi WHERE.

Secara teknis, function bisa digunakan untuk semua perintah yang menghendaki suatu nilai. Termasuk untuk diinput ke dalam variabel:

```
SET @hasil_tambah = tambah(1234, 456);
Query OK, 0 rows affected (0.05 sec)
```

```
SELECT @hasil_tambah;
+-----+
| @hasil_tambah |
+-----+
|      1690 |
+-----+
```

Variabel @hasil_tambah akan menyimpan hasil dari fungsi tambah(1234,456).

25.18 Latihan Stored Procedure

Menutup bab tentang **stored procedure** dan **stored function**, saya sudah menyiapkan sebuah latihan. Latihan ini berdasarkan data dari tabel nilai_mahasiswa.

Tujuan latihan ini, membuat ulang procedure penilaian(). Stored procedure penilaian() nantinya menerima 1 buah parameter berupa kode nim mahasiswa. Berdasarkan kode nim ini, cari nilai IP semester_1 dari mahasiswa tersebut.

Hasil akhir dari procedure penilaian() berupa string, tergantung nilai IP semester_1 dari mahasiswa tersebut. Aturannya adalah sebagai berikut:

- IPK 0.00 - 1.00: ‘Serius kuliah g sih?’
- IPK 1.01 - 2.00: ‘Kebanyakan main’
- IPK 2.01 - 3.00: ‘Berusaha lagi’
- IPK 3.01 - 4.00: ‘Mantap, pertahankan!’
- Selain itu: ‘Nilai tidak sesuai’

Berikut hasil akhir yang diinginkan dari pemanggilan stored procedure penilaian():

```
CALL penilaian('17080305');
-- Mantap, pertahankan!
```

```
CALL penilaian('17140143');
-- Berusaha lagi
```

Silahkan anda coba rancang kode programnya!

Tips:

Membuat procedure ini butuh beberapa langkah:

1. Ambil nilai argumen yang berupa nim lalu input ke dalam SELECT . . . WHERE untuk mencari nilai semester_1 dari mahasiswa tersebut.
2. Siapkan sebuah variabel untuk menampung nilai semester_1 hasil dari langkah nomor 1.
3. Buat kondisi IF ELSEIF ELSE atau CASE puncuk mengkonversi nilai semester_1 menjadi string dengan aturan di dalam soal.

25.19 Jawaban Latihan Stored Procedure

Jika anda butuh membuat kode program yang agak kompleks seperti ini, pecah masalah tersebut menjadi beberapa langkah.

Pertama, kita butuh kode program untuk mencari nilai semester_1 berdasarkan nilai nim. Rancang kode ini menggunakan variabel dan query SELECT terlebih dahulu. Tidak perlu menggunakan stored procedure karena itu hanya akan mempersulit perancangan kode program.

Berikut contoh percobaannya:

```
SELECT * FROM nilai_mahasiswa;

+-----+-----+-----+-----+
| nim | nama | semester_1 | semester_2 | semester_3 |
+-----+-----+-----+-----+
| 17080305 | Rina Kumala Sari | 3.45 | 2.56 | 3.67 |
| 17090113 | Riana Putria | 3.12 | 2.98 | 3.45 |
| 17090308 | Christine Wijaya | 3.78 | 3.23 | 3.11 |
| 17140119 | Sandri Fatmala | 2.12 | 2.78 | 2.56 |
| 17140143 | Rudi Permana | 2.56 | 3.14 | 3.22 |
+-----+-----+-----+-----+
```



```
SET @nim_mhs = '17080305';
SET @nilai_ip = 0.00;
SELECT semester_1 INTO @nilai_ip FROM nilai_mahasiswa WHERE nim = @nim_mhs;
-- ERROR 1267 (HY000): Illegal mix of collations (latin1_swedish_ci,IMPLICIT)
-- and (cp850_general_ci,IMPLICIT) for operation '='
```

Diawal saya menampilkan seluruh isi tabel `nilai_mahasiswa` sebagai referensi.

Selanjutnya saya membuat variabel `@nim_mhs` yang diisi dengan nim '17080305'. Variabel ini menjadi sample untuk proses pencarian nilai `semester_1`. Dalam procedure nanti, `nim` ini akan didapat dari parameter procedure `penilaian()`.

Lalu saya membuat variabel kedua `@nilai_ip = 0.00`. Variabel ini berguna untuk menampung nilai `semester_1` dari mahasiswa yang akan dicari.

Menggunakan query `SELECT ... INTO`, saya ingin menyimpan nilai kolom `semester_1` ke dalam variabel `@nilai_ip`, dimana kondisi yang digunakan adalah `WHERE nim = @nim_mhs`.

Namun tampil pesan error `Illegal mix of collations`, apa yang terjadi? Pada saat merancang query diatas, saya yakin tidak ada yang salah. Tapi kenapa tampil error?

Dalam kasus seperti ini, saya juga ingin mengajak anda dalam proses *debugging* (pencarian kesalahan). Jika mengalami situasi yang sama, yang harus diperhatikan adalah pesan error. Error `Illegal mix of collations` tidak umum terjadi.

Biasanya, pesan error MySQL yang sering di dapat adalah '*ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near '...' at line 1*'. Error ini kebanyakan karena kita salah ketik perintah atau ada salah penempatan query.

Hal pertama yang saya lakukan adalah *googling* dengan keyword sesuai pesan error, yakni "`Illegal mix of collations mysql`". Salah satu hasil yang membahas hal ini adalah dari stackoverflow.com: [Illegal mix of collations MySQL Error³](https://stackoverflow.com/questions/1008287/illegal-mix-of-collations-mysql-error).

Disana dijelaskan bahwa error tersebut tampil akibat perbedaan collation yang dipakai oleh MySQL. Di dalam MySQL, ternyata ada 3 buah default collation:

³<https://stackoverflow.com/questions/1008287/illegal-mix-of-collations-mysql-error>

```
SHOW VARIABLES LIKE 'collation%';
+-----+-----+
| Variable_name      | Value       |
+-----+-----+
| collation_connection | cp850_general_ci |
| collation_database   | latin1_swedish_ci |
| collation_server     | latin1_swedish_ci |
+-----+-----+
```

`collation_connection` adalah collation yang dipakai ketika membuat nilai diluar tabel, seperti ketika membuat variabel, sedangkan `collation_database` adalah collation default dari tabel dan database.

Dalam kasus kita, variabel `@nilai_ip` memiliki collation `cp850_general_ci`, sedangkan kolom `nim` dalam tabel `nilai_mahasiswa` memiliki collation `latin1_swedish_ci`. MySQL protes ketika dua buah collation yang berbeda saling dibandingkan.

Error seperti ini memang tidak umum. Siapa yang menyangka bahwa perbedaan collation bisa menyebabkan error di operasi perbandingan string MySQL. Google dan situs forum seperti stackoverflow adalah solusi terbaik untuk masalah seperti ini.

Baik, kembali ke kode program kita. Saya akan mengubah pengaturan `collation_connection` menggunakan perintah berikut (didapat dari jawaban forum stackoverflow.com sebelumnya):

```
SET collation_connection = 'latin1_swedish_ci';
-- Query OK, 0 rows affected (0.03 sec)
```

Perubahan collation ini berlaku per-session. Artinya ketika kita logout dari MySQL client, pengaturan `collation_connection` akan kembali ke `cp850_general_ci` (collation awal). Jika ingin pengaturan `collation_connection` bersifat permanen, kita harus mengubahnya dari file `my.ini`.

Setelah perubahan `collation_connection`, mari uji kembali query `SELECT...INTO` sebelumnya.

```
SET @nim_mhs = '17080305';
SET @nilai_ip = 0.00;
SELECT semester_1 INTO @nilai_ip FROM nilai_mahasiswa WHERE nim = @nim_mhs;
-- Query OK, 1 row affected (0.23 sec)
```

Kali ini tidak error, saya akan periksa isi dari variabel `@nilai_ip`:

```
SELECT @nilai_ip;
+-----+
| @nilai_ip |
+-----+
|      3.45 |
+-----+
```

Yup, sesuai dengan yang diharapkan. Mahasiswa dengan nim '17080305' memiliki IP semester 1 dengan nilai 3.45.

Saya akan test untuk nim lain:

```
SET @nim_mhs = '17140119';
SET @nilai_ip = 0.00;
SELECT semester_1 INTO @nilai_ip FROM nilai_mahasiswa WHERE nim = @nim_mhs;

SELECT @nilai_ip;
+-----+
| @nilai_ip |
+-----+
|      2.12 |
+-----+
```

Hasilnya juga sesuai.

Langkah berikutnya adalah mengkonversi query diatas menjadi procedure penilaian():

```
DROP PROCEDURE IF EXISTS penilaian;

DELIMITER //
CREATE PROCEDURE penilaian(nim_mhs CHAR(8))
BEGIN
    DECLARE nilai_ip DECIMAL(4,2) DEFAULT 0.00;
    SELECT semester_1 INTO nilai_ip FROM nilai_mahasiswa WHERE nim = nim_mhs;
    SELECT nilai_ip;
END//
DELIMITER ;

CALL penilaian('17080305');
-- 3.45

CALL penilaian('17140143');
-- 2.56
```

Kembali, untuk membuat procedure yang kompleks, kita harus buat langkah demi langkah lalu uji, tambah sedikit lagi, lalu kembali uji. Ini dilakukan supaya error bisa dideteksi sedini

mungkin. Mencari error di 6 baris kode program tentu lebih mudah dibandingkan mencari error dari 20 baris.

Procedure penilaian() diatas sudah menampilkan nilai IP semester_1 untuk setiap mahasiswa yang nim-nya dijadikan sebagai nilai input.

Sesudah memastikan tidak ada kendala, barulah saya membuat kondisi CASE:

```
DROP PROCEDURE IF EXISTS penilaian;

DELIMITER //
CREATE PROCEDURE penilaian(nim_mhs CHAR(8))
BEGIN
    DECLARE nilai_ip DECIMAL(4,2) DEFAULT 0.00;
    SELECT semester_1 INTO nilai_ip FROM nilai_mahasiswa WHERE nim = nim_mhs;

    CASE
        WHEN ((nilai_ip >= 0.00) AND (nilai_ip <= 1.00)) THEN
            SELECT 'Serius kuliah g sih!';
        WHEN ((nilai_ip > 1.00) AND (nilai_ip <= 2.00)) THEN
            SELECT 'Kebanyakan main';
        WHEN ((nilai_ip > 2.00) AND (nilai_ip <= 3.00)) THEN
            SELECT 'Berusaha lagi';
        WHEN ((nilai_ip > 3.00) AND (nilai_ip <= 4.00)) THEN
            SELECT 'Mantap, pertahankan!';
        ELSE
            SELECT 'Nilai tidak sesuai';
    END CASE;

END//
DELIMITER ;

CALL penilaian('17080305');
-- Mantap, pertahankan!

CALL penilaian('17140143');
-- Berusaha lagi
```

Inilah kode akhir dari procedure penilaian(). Hasil nilai semester 1 yang didapat dari query SELECT akan diproses ke dalam 4 kondisi. Jika nilai tersebut memenuhi syarat kondisi, tampilkan string yang sesuai.

Sebagai latihan tambahan, bisakah anda memodifikasi kode diatas agar proses seleksi diganti jadi IF ELSE? atau bagaimana jika perhitungannya melibatkan nilai IPK (nilai rata-rata dari 3 semester)? silahkan berkreasi.

Dalam bab ini kita telah membahas tentang salah satu fitur lanjutan di dalam MySQL, yakni **stored procedure** dan **stored function**. Berikutnya, kita akan masuk ke **Trigger**.

26. MySQL Trigger

Materi yang akan kita bahas dalam bab kali ini melengkapi fitur advanced di MySQL, yakni tentang **Trigger**. *Trigger* sendiri sangat mirip seperti **stored procedure**, hanya saja trigger akan berjalan secara otomatis saat sebuah event terjadi.



Agar bisa mengikuti pembahasan yang ada, perlu dasar pemrograman SQL yang sudah kita bahas dalam bab tentang **stored procedure**.

26.1 Pengertian Trigger

Trigger adalah kumpulan kode program SQL yang dijalankan secara otomatis ketika sebuah event terjadi. Event yang dimaksud yakni saat perintah **INSERT**, **UPDATE** atau **DELETE** diproses ke dalam sebuah tabel.

Trigger sebenarnya merupakan tipe khusus dari stored procedure. Isi dari trigger itu sendiri tidak berbeda dengan stored procedure yang sudah kita bahas dalam bab sebelumnya.

Di dalam trigger, kita juga bisa membuat variabel, kondisi if else, perulangan, hingga parameter. Namun berbeda dengan stored procedure yang harus dipanggil secara eksplisit menggunakan perintah **CALL**, trigger berjalan secara otomatis.

Keunggulan dan kekurangan trigger juga sama seperti stored procedure. Ada yang berpendapat bahwa programming seperti trigger seharusnya tidak dibuat di database, tapi di bahasa pemrograman tempat data nantinya akan diolah (misalnya menggunakan PHP).

Terlepas dari itu semua, trigger merupakan salah satu fitur yang bisa sangat bermanfaat untuk kasus-kasus tertentu.

26.2 Format Dasar Penulisan Trigger

Trigger adalah tipe khusus dari stored procedure, sehingga cara penulisannya juga hampir sama. Berikut format dasar penulisan trigger di dalam MySQL:

```
CREATE TRIGGER trigger_name trigger_time trigger_event
ON table_name
FOR EACH ROW
BEGIN
...
END;
```

Setelah penulisan `CREATE TRIGGER`, diikuti dengan `trigger_name` yang berfungsi sebagai nama trigger. Aturan penulisan nama trigger ini sama seperti *identifier* lainnya di dalam MySQL, yakni sama seperti aturan penulisan nama tabel dan nama database.

Berikutnya adalah `trigger_time`, yakni kapan waktu trigger ini akan dijalankan. Pilihan untuk `trigger_time` ada 2 buah: BEFORE atau AFTER. Kedua pilihan ini akan berkaitan dengan perintah berikutnya, yakni `trigger_event`.

`Trigger_event` adalah event atau query yang menjadi patokan kapan sebuah trigger akan dijalankan. Terdapat 3 event penentu trigger : `INSERT`, `UPDATE` dan `DELETE`.

Dengan mengkombinasikan `trigger_time` dan `trigger_event`, terdapat 6 kondisi event dimana sebuah trigger akan diproses:

- `BEFORE INSERT`
- `AFTER INSERT`
- `BEFORE UPDATE`
- `AFTER UPDATE`
- `BEFORE DELETE`
- `AFTER DELETE`

Sebagai contoh, jika saya menggunakan `BEFORE INSERT`, artinya trigger akan diproses tepat sebelum proses input data ke dalam tabel berlangsung. Jika saya menggunakan `AFTER DELETE`, artinya trigger akan diproses setelah sebuah data dihapus dari dalam tabel.

Setelah menentukan `trigger_name`, `trigger_time` dan `trigger_event`. Perintah selanjutnya berupa nama tabel tempat dimana trigger akan berjalan. Setiap trigger nantinya akan “melekat” ke dalam tabel ini. Jika tabel dihapus, trigger juga akan terhapus.

Isi dari trigger itu sendiri berada diantara perintah `BEGIN` dan `END`. Disini kita bisa menggunakan semua konsep programming SQL yang sudah dibahas di dalam bab tentang stored procedure.

26.3 Membuat Trigger

Baik, mari kita masuk ke contoh praktik pembuatan trigger di dalam MySQL. Sebagai tabel sample, saya akan membuat ulang tabel `mahasiswa`:

```
DROP TABLE IF EXISTS mahasiswa;

CREATE TABLE mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    asal VARCHAR(50),
    jurusan VARCHAR (20),
    nilai_uan DEC(5,2)
);
```

Tidak perlu input data ke dalam tabel ini, karena kita akan proses sambil menjalankan trigger. Ideanya adalah, ketika sebuah event terjadi ke dalam tabel `mahasiswa`, seperti penambahan data baru, hal itu akan dicatat ke tabel lain. Disini saya ingin membuat sebuah `log` atau catatan aktifitas dari tabel `mahasiswa`.

Catatan aktifitas atau log ini akan disimpan ke dalam tabel `mahasiswa_log`, dengan query pembuatan tabel sebagai berikut:

```
CREATE TABLE mahasiswa_log (
    no INT AUTO_INCREMENT PRIMARY KEY,
    nama VARCHAR(50),
    waktu TIMESTAMP,
    keterangan TEXT
);
```

Tabel `mahasiswa_log` terdiri dari 4 kolom:

- Kolom `no` berisi nomor urut yang di set sebagai `AUTO_INCREMENT PRIMARY KEY`.
- Kolom `nama` nantinya berisi nama mahasiswa yang diambil dari tabel `mahasiswa`.
- Kolom `waktu` akan menyimpan tanggal dan waktu perubahan data terjadi.
- Kolom `keterangan` berisi penjelasan tentang apa yang terjadi di dalam tabel mahasiswa.

Sebagai contoh trigger pertama, saya ingin tabel `mahasiswa_log` berisi otomatis pada saat proses `INSERT` berlangsung ke dalam tabel `mahasiswa`. Maksudnya, ketika sebuah data diinput ke dalam tabel `mahasiswa`, catat aktifitas tersebut ke tabel `mahasiswa_log`.

Untuk keperluan ini, trigger akan berjalan **sebelum** query `INSERT`, atau `BEFORE INSERT`. Berikut kode program untuk membuat trigger ini:

```
DELIMITER $$

CREATE TRIGGER before_mahasiswa_insert
BEFORE INSERT ON mahasiswa
FOR EACH ROW
BEGIN
    INSERT INTO mahasiswa_log VALUES
    (0, NEW.nama, NOW(), 'Ada penambahan mahasiswa baru');
END$$
DELIMITER ;
```

Di baris pertama, kita tetap butuh perintah `DELIMITER` untuk mengganti karakter penutup query sebelum membuat trigger.

Disini saya membuat sebuah trigger dengan nama `before_mahasiswa_insert`. Penamaan seperti ini semata-mata agar mudah dikelola. Berikut format penulisan nama trigger yang saya gunakan:

```
(before | after)_namabel_(insert| update | delete)
```

Dari nama trigger `before_mahasiswa_insert`, kita bisa langsung paham bahwa trigger ini akan berjalan sebelum proses insert ke dalam tabel `mahasiswa`. Aturan penamaan seperti ini bukanlah sebuah keharusan, tapi lebih ke saran.

Setelah menulis nama trigger, berikutnya terdapat perintah `BEFORE INSERT ON mahasiswa FOR EACH ROW`, inilah yang menegaskan kalau trigger akan berjalan sebelum perintah `INSERT` diproses ke tabel `mahasiswa`.

Isi dari trigger itu sendiri hanya 1 perintah query `INSERT`:

```
INSERT INTO mahasiswa_log VALUES  
(0, NEW.nama, NOW(), 'Ada penambahan mahasiswa baru');
```

Artinya, ketika terdapat data baru yang diinput ke dalam tabel `mahasiswa`, jalankan perintah diatas.

Untuk kolom pertama, saya menulis angka 0. Angka ini akan dikirim ke kolom `no` di tabel `mahasiswa_log`. Karena kolom `no` sudah di set sebagai `AUTO_INCREMENT`, maka nilai 0 akan dikonversi sesuai nomor urut berikutnya.

Kolom kedua diisi dengan `NEW.nama`. `NEW` adalah objek khusus yang menyimpan referensi ke tabel induk dari trigger, yang dalam contoh kita adalah tabel `mahasiswa`. Objek `NEW.nama` artinya ambil nilai kolom `nama` yang baru diinput ke dalam tabel `mahasiswa`.

Untuk kolom ketiga diisi dengan fungsi `NOW()`. Fungsi ini akan mengembalikan nilai tanggal dan waktu dalam format `yyyy-mm-dd hh:ii:ss` untuk diinput ke kolom `waktu` dari tabel `mahasiswa_log`. Fungsi `NOW()` sudah kita bahas dalam bab tentang *date function*.

Terakhir, untuk kolom keterangan saya isi dengan string '`Ada penambahan mahasiswa baru`'.

Mari kita coba apakah trigger `before_mahasiswa_insert` bisa berjalan sebagaimana mestinya. Silahkan input satu data ke dalam tabel `mahasiswa`:

```
INSERT INTO mahasiswa VALUES ('17090113', 'Riana Putria', 'Padang', 'Kimia', \  
339.20);
```

Kemudian periksa isi tabel `mahasiswa` dan `mahasiswa_log`:

```
SELECT * FROM mahasiswa;
+-----+-----+-----+-----+
| nim      | nama          | asal      | jurusan | nilai_uan |
+-----+-----+-----+-----+
| 17090113 | Riana Putria | Padang    | Kimia    |     339.20 |
+-----+-----+-----+-----+

SELECT * FROM mahasiswa_log;
+-----+-----+-----+-----+
| no   | nama        | waktu       | keterangan |
+-----+-----+-----+-----+
| 1 | Riana Putria | 2017-12-07 21:19:28 | Ada penambahan mahasiswa baru |
+-----+-----+-----+-----+
```

Trigger sukses berjalan! Tabel `mahasiswa_log` otomatis terisi pada saat sebuah data baru diinput ke dalam tabel `mahasiswa`.

Sebagai contoh kedua, saya ingin membuat trigger `after_mahasiswa_delete`. Dari nama ini tentunya anda sudah bisa menebak kalau trigger tersebut akan dijalankan setelah data dihapus dari tabel `mahasiswa`, yakni event AFTER DELETE.

Berikut kode program yang digunakan:

```
DELIMITER $$
CREATE TRIGGER after_mahasiswa_delete
AFTER DELETE ON mahasiswa
FOR EACH ROW
BEGIN
  INSERT INTO mahasiswa_log VALUES (0, OLD.nama, NOW(), 'Ada penghapusan maha\
siswa');
END$$
DELIMITER ;
```

Isi dari trigger `after_mahasiswa_delete` hampir sama dengan trigger `before_mahasiswa_insert` yang sudah kita buat sebelumnya. Bedanya ada di penulisan event menjadi AFTER DELETE, serta isi kolom kedua dan keempat untuk tabel `mahasiswa_log`.

Untuk kolom kedua, sekarang saya menggunakan `OLD.nama`. Fungsi `OLD` disini sama seperti `NEW`, yakni sebagai referensi kepada tabel asal dari trigger. `OLD.nama` akan mengambil nilai kolom `nama` dari tabel `mahasiswa`.

Berikut aturan penggunaan referensi `NEW` dan `OLD` untuk merujuk ke tabel induk:

- Untuk event `INSERT`, referensi yang digunakan adalah `NEW`.
- Untuk event `DELETE`, referensi yang digunakan adalah `OLD`.
- Untuk event `UPDATE`, referensi yang digunakan adalah `NEW` dan `OLD`.

Referensi NEW dan OLD ini bergantung kepada apakah itu data baru atau data lama.

Untuk query INSERT, kita menambahkan data baru ke dalam tabel, karena itulah referensi yang dipakai berupa NEW. Untuk query DELETE, kita menghapus data yang tidak dibutuhkan dari tabel, sehingga digunakan referensi OLD.

Khusus untuk query UPDATE, referensi NEW dan OLD bisa dipakai, karena disini terdapat 2 data, yakni data baru yang akan diinput, serta data lama yang akan dihapus. Kita akan lihat penggunaannya untuk event trigger untuk query UPDATE sesaat lagi.

Kembali ke trigger after_mahasiswa_delete, kolom keempat untuk tabel mahasiswa_log akan diisi dengan string 'Ada penambahan mahasiswa baru'.

Setelah trigger dibuat, saya akan coba hapus data mahasiswa Riana Putria dari tabel mahasiswa, lalu periksa isi dari tabel mahasiswa_log:

```
DELETE FROM mahasiswa WHERE nama = 'Riana Putria';
```

```
SELECT * FROM mahasiswa;
```

```
Empty set (0.00 sec)
```

```
SELECT * FROM mahasiswa_log;
```

no	nama	waktu	keterangan
1	Riana Putria	2017-12-07 21:19:28	Ada penambahan mahasiswa baru
2	Riana Putria	2017-12-07 21:28:44	Ada penghapusan mahasiswa

Terlihat, dari hasil tampilan tabel mahasiswa_log, trigger after_mahasiswa_delete sudah berhasil berjalan.

Terakhir, saya akan buat trigger untuk event UPDATE, yakni after_mahasiswa_update. Berikut kode programnya:

```
DELIMITER $$  
CREATE TRIGGER after_mahasiswa_update  
AFTER UPDATE ON mahasiswa  
FOR EACH ROW  
BEGIN  
    DECLARE keterangan TEXT;  
    SET keterangan = CONCAT(OLD.nama, ' -> ', NEW.nama);  
    INSERT INTO mahasiswa_log VALUES (0, NEW.nama, NOW(), keterangan);  
END$$  
DELIMITER ;
```

Isi trigger ini juga mirip dengan contoh trigger sebelumnya, hanya saja sekarang menggunakan event AFTER UPDATE ON mahasiswa. Artinya, trigger akan berjalan jika ada query UPDATE yang dilakukan ke dalam tabel mahasiswa.

Di dalam trigger `after_mahasiswa_update`, saya membuat sebuah variabel keterangan. Variabel ini digunakan untuk menampung hasil dari fungsi `CONCAT(OLD.nama, ' -> ', NEW.nama)`. Variabel keterangan ini akan menjadi nilai teks untuk kolom keterangan di tabel `mahasiswa_log`.

Saya ingin ketika query `UPDATE` diproses ke dalam tabel `mahasiswa`, kolom keterangan di tabel `mahasiswa_log` akan mencatat perubahan ini berupa nama mahasiswa sebelum dan sesudah proses update.

Nama mahasiswa sebelum update, yakni nama mahasiswa yang akan diganti, bisa diakses dari referensi `OLD.nama`. Sedangkan nama mahasiswa setelah update, yakni nama mahasiswa baru, bisa diakses menggunakan `NEW.nama`. Kedua perintah ini sama-sama mengambil nilai kolom `nama` dari tabel `mahasiswa`. Fungsi `CONCAT()` kemudian menggabungkan nama lama dan nama baru ini menjadi sebuah string.

Untuk mencoba trigger ini, saya harus menambah data baru ke dalam tabel `mahasiswa`, karena tabel `mahasiswa` masih kosong akibat query `DELETE` sebelumnya:

```
INSERT INTO mahasiswa
VALUES ('17080225', 'Husli Khairan', 'Jakarta', 'Akuntansi', 288.55);
```

```
SELECT * FROM mahasiswa;
```

nim	nama	asal	jurusan	nilai_uan
17080225	Husli Khairan	Jakarta	Akuntansi	288.55

1 row in set (0.00 sec)

```
SELECT * FROM mahasiswa_log;
```

no	nama	waktu	keterangan
1	Riana Putria	2017-12-07 21:19:28	Ada penambahan mahasiswa baru
2	Riana Putria	2017-12-07 21:28:44	Ada penghapusan mahasiswa
3	Husli Khairan	2017-12-07 22:32:09	Ada penambahan mahasiswa baru

Penambahan data ini akan kembali menjalankan trigger `before_mahasiswa_insert`, sehingga muncul data baru ke dalam tabel `mahasiswa_log`.

Berikutnya, sekarang saya akan update tabel `mahasiswa`:

```
UPDATE mahasiswa SET nama = 'Rudi Permana' WHERE nim = '17080225';

SELECT * FROM mahasiswa;
+-----+-----+-----+-----+
| nim      | nama           | asal      | jurusan     | nilai_uan |
+-----+-----+-----+-----+
| 17080225 | Rudi Permana | Jakarta   | Akuntansi   |    288.55  |
+-----+-----+-----+-----+

SELECT * FROM mahasiswa_log;
+-----+-----+-----+-----+
| no   | nama           | waktu        | keterangan   |
+-----+-----+-----+-----+
| 1    | Riana Putria | 2017-12-07 21:19:28 | Ada penambahan mahasiswa baru |
| 2    | Riana Putria | 2017-12-07 21:28:44 | Ada penghapusan mahasiswa   |
| 3    | Husli Khairan | 2017-12-07 22:32:09 | Ada penambahan mahasiswa baru |
| 4    | Rudi Permana  | 2017-12-07 22:52:42 | Husli Khairan -> Rudi Permana |
+-----+-----+-----+-----+
```

Seperti yang terlihat, baris ke-4 untuk tabel mahasiswa_log tampil sebagai efek dari trigger after_mahasiswa_update.

26.4 Implisit Event Trigger

Dari penjelasan sebelumnya, disebutkan bahwa trigger hanya bisa berjalan untuk 3 event, atau 3 query, yakni INSERT, UPDATE dan DELETE. Hanya saja, terdapat beberapa query yang secara implisit (tidak langsung), akan menjalankan salah satu query tersebut.

Sebagai contoh, query REPLACE akan menjalankan query INSERT atau UPDATE tergantung situasi. Jika data yang sama belum ada di dalam database, query REPLACE akan memproses INSERT. Namun jika ditemukan data yang sama, query REPLACE akan menggunakan UPDATE. Berikut percobaannya:

```
REPLACE INTO mahasiswa
VALUES ('17140119', 'Sandri Fatmala', 'Bandung', 'Ilmu Komputer', 322.91);

SELECT * FROM mahasiswa;
+-----+-----+-----+-----+
| nim      | nama           | asal      | jurusan     | nilai_uan |
+-----+-----+-----+-----+
| 17080225 | Rudi Permana | Jakarta   | Akuntansi   |    288.55  |
| 17140119 | Sandri Fatmala | Bandung   | Ilmu Komputer |    322.91  |
+-----+-----+-----+-----+

SELECT * FROM mahasiswa_log;
```

no	nama	waktu	keterangan
1	Riana Putria	2017-12-07 21:19:28	Ada penambahan mahasiswa baru
2	Riana Putria	2017-12-07 21:28:44	Ada penghapusan mahasiswa
3	Husli Khairan	2017-12-07 22:32:09	Ada penambahan mahasiswa baru
4	Rudi Permana	2017-12-07 22:52:42	Husli Khairan -> Rudi Permana
5	Sandri Fatmala	2017-12-07 22:55:13	Ada penambahan mahasiswa baru

Disini saya menjalankan query REPLACE. Karena query ini secara tidak langsung menjalankan query INSERT atau UPDATE, maka trigger yang sesuai akan diproses.

Dalam contoh diatas, tidak ditemukan data dengan nim ‘17140119’ di dalam tabel mahasiswa, sehingga query REPLACE akan diproses sebagai INSERT. Akibatnya, trigger before_mahasiswa_insert akan dipanggil.

Namun tidak semua query menggunakan pemanggilan implisit seperti ini. Contohnya query TRUNCATE yang biasa dipakai untuk mengosongkan tabel. Prinsip kerja dari TRUNCATE adalah dengan menghapus lalu membuat ulang tabel, sehingga trigger untuk query DELETE tidak akan berjalan.

Sebagai tambahan, trigger akan diproses baris per baris. Jika terdapat penambahan atau penghapusan lebih dari 1 baris sekaligus, trigger akan dijalankan untuk semua data yang terhapus. Sebagai contoh, saya akan menghapus seluruh isi tabel mahasiswa dengan query DELETE:

```
DELETE FROM mahasiswa;
```

```
SELECT * FROM mahasiswa;
```

```
Empty set (0.00 sec)
```

```
SELECT * FROM mahasiswa_log;
```

no	nama	waktu	keterangan
1	Riana Putria	2017-12-07 21:19:28	Ada penambahan mahasiswa baru
2	Riana Putria	2017-12-07 21:28:44	Ada penghapusan mahasiswa
3	Husli Khairan	2017-12-07 22:32:09	Ada penambahan mahasiswa baru
4	Rudi Permana	2017-12-07 22:52:42	Husli Khairan -> Rudi Permana
5	Sandri Fatmala	2017-12-07 22:55:13	Ada penambahan mahasiswa baru
6	Rudi Permana	2017-12-07 22:58:35	Ada penghapusan mahasiswa
7	Sandri Fatmala	2017-12-07 22:58:35	Ada penghapusan mahasiswa

Baris ke 6 dan 7 dari tampilan diatas berasal dari trigger after_mahasiswa_delete. Karena ada 2 baris mahasiswa yang dihapus, trigger ini juga akan dijalankan dua kali, masing-masing untuk Rudi Permana dan Sandri Fatmala.

26.5 Menghapus Trigger

Di dalam MySQL, sebuah trigger “melekat” ke dalam tabel tempat trigger itu dibuat.

Ketiga trigger yang sudah kita rancang sebelum ini menggunakan perintah seperti AFTER UPDATE ON mahasiswa, artinya trigger ini adalah milik tabel mahasiswa. Ketika tabel mahasiswa dihapus, ketiga trigger tersebut juga akan terhapus.

Untuk melihat apa saja trigger yang tersedia di dalam tabel mahasiswa, bisa menggunakan query berikut:

```
SHOW TRIGGERS FROM belajar WHERE `table` = 'mahasiswa' \G

***** 1. row *****
Trigger: before_mahasiswa_insert
Event: INSERT
Table: mahasiswa
Statement: BEGIN
INSERT INTO mahasiswa_log VALUES
(0, NEW.nama, NOW(), 'Ada penambahan mahasiswa baru');
END
Timing: BEFORE
Created: NULL
sql_mode: STRICT_ALL_TABLES,NO_AUTO_CREATE_USER, ...
Definer: root@localhost
character_set_client: cp850
collation_connection: cp850_general_ci
Database Collation: latin1_swedish_ci

***** 2. row *****
Trigger: after_mahasiswa_update
Event: UPDATE
Table: mahasiswa
.....: .....
```

Jika anda menjalankan query diatas, akan tampil 3 trigger yang sudah kita buat sebelumnya. Perhatikan bahwa untuk kondisi WHERE, keyword table harus ditulis di dalam tanda *backtick*, kalau tidak akan menghasilkan error.

Untuk melihat apa saja trigger yang terdapat di seluruh database belajar, bisa menggunakan perintah berikut:

```
SHOW TRIGGERS FROM belajar \G
```

Dan untuk melihat seluruh trigger di semua database, jalankan query SHOW TRIGGERS tanpa menulis nama database:

```
SHOW TRIGGERS \G
```

Untuk menghapus trigger, tersedia query `DROP TRIGGER` dengan format dasar sebagai berikut:

```
DROP TRIGGER [IF EXISTS] nama_trigger;
```

Sebagai contoh praktek, saya akan menghapus ketiga trigger yang sudah kita buat sebelumnya:

```
DROP TRIGGER before_mahasiswa_insert;
-- Query OK, 0 rows affected (0.79 sec)
```

```
DROP TRIGGER after_mahasiswa_delete;
-- Query OK, 0 rows affected (1.00 sec)
```

```
DROP TRIGGER after_mahasiswa_update;
-- Query OK, 0 rows affected (0.44 sec)
```

```
SHOW TRIGGERS FROM belajar WHERE `table` = 'mahasiswa' \G
-- Empty set (0.63 sec)
```

Di dalam MySQL, tidak tersedia query `ALTER TRIGGER`. Jika kita ingin memodifikasi isi sebuah trigger, hapus trigger tersebut kemudian buat kembali.

26.6 BEFORE atau AFTER?

Terdapat 2 pengaturan waktu mengenai kapan sebuah trigger dijalankan, yakni sebelum query (BEFORE) dan sesudah query (AFTER). Yang mana sebaiknya dipakai?

Ini tergantung dari fungsi trigger itu sendiri. Sebagai contoh, jika sebuah trigger ditulis `BEFORE INSERT ON mahasiswa` artinya trigger itu akan diproses sebelum data ditambahkan ke dalam tabel `mahasiswa`.

Waktu BEFORE cocok dipakai untuk membuat sebuah proses validasi. Jika data yang diinput tidak sesuai kita bisa membatalkan transaksi. Ini tidak bisa dilakukan dengan AFTER karena data itu telah berhasil diinput ke dalam tabel.

Sebagai contoh, saya ingin membuat sebuah trigger untuk tabel `mahasiswa` yang akan memeriksa `nilai_uan`. Jika `nilai_uan` mahasiswa kurang dari 250, maka transaksi dibatalkan. Berikut kode program untuk membuat trigger ini:

```

DELIMITER $$

CREATE TRIGGER before_mahasiswa_insert
BEFORE INSERT ON mahasiswa
FOR EACH ROW
BEGIN
    IF (NEW.nilai_uan < 250) THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Nilai UAN dibawah minimum!';
    END IF;
END$$
DELIMITER ;

```

Trigger `before_mahasiswa_insert` menggunakan `BEFORE INSERT ON mahasiswa`. Artinya trigger ini akan diproses tepat sebelum query `INSERT` berjalan.

Di dalam trigger saya membuat sebuah kondisi `IF` untuk memeriksa apakah `nilai_uan` yang diinput kurang dari 250. Referensi untuk nilai uan ini didapat dari `NEW.nilai_uan`.

Jika ternyata nilai uan mahasiswa tersebut kurang dari 250, jalankan perintah `SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Nilai UAN dibawah minimum!'`.

Perintah `SQLSTATE '45000'` dipakai untuk menghasilkan pesan error dengan kode `45000`. Di dalam [MySQL Manual¹](#), kode error '`45000`' berarti *unhandled user-defined exception*, yakni sebuah error generik yang dibuat oleh programmer (*user-defined*) dan bukan error bawaan MySQL. Isi dari error tersebut berupa pesan '`Nilai UAN dibawah minimum!`'.

Perintah `SIGNAL SQLSTATE` akan memaksa MySQL menghentikan proses yang saat ini terjadi. Akibatnya, query `INSERT` tidak jadi diproses. Mari kita test apakah validasi ini berjalan atau tidak:

```

INSERT INTO mahasiswa
VALUES ('17140133', 'Ikhsan Prayoga', 'Jakarta', 'Ilmu Komputer', 200.99);
-- ERROR 1644 (45000): Nilai UAN dibawah minimum!

```

```

INSERT INTO mahasiswa
VALUES ('17080305', 'Rina Kumala Sari', 'Jakarta', 'Akuntansi', 249.99);
-- ERROR 1644 (45000): Nilai UAN dibawah minimum!

```

```

INSERT INTO mahasiswa
VALUES ('17080225', 'Husli Khairan', 'Jakarta', 'Akuntansi', 288.55);
-- Query OK, 1 row affected (0.06 sec)

```

```

SELECT * FROM mahasiswa;
+-----+-----+-----+-----+
| nim      | nama          | asal       | jurusan     | nilai_uan |
+-----+-----+-----+-----+
| 17080225 | Husli Khairan | Jakarta   | Akuntansi   |      288.55 |
+-----+-----+-----+-----+

```

¹<https://dev.mysql.com/doc/refman/5.5/en/signal.html>

Dua query INSERT pertama akan menghasilkan pesan *ERROR 1644 (45000): Nilai UAN dibawah minimum!*, inilah pesan dari trigger karena nilai_uan untuk mahasiswa dibawah 250. Query INSERT ketiga lolos validasi karena Husli Khairan memiliki nilai uan 288.55, sehingga tidak memenuhi syarat kondisi NEW.nilai_uan < 250.

Proses validasi seperti ini kurang cocok jika menggunakan AFTER INSERT, karena proses INSERT sudah terjadi. Meskipun, kita bisa saja merancang isi trigger yang didalamnya akan menghapus data dari tabel mahasiswa, namun cara ini tidaklah ideal.

Waktu proses AFTER lebih cocok untuk proses pembuatan log, atau suatu proses yang melibatkan tabel lain. Sebagai contoh, proses untuk mahasiswa_log sebenarnya lebih pas jika menggunakan AFTER INSERT, AFTER DELETE, atau AFTER UPDATE. Hanya saja untuk trigger yang sederhana seperti tabel mahasiswa_log, perbedaan waktu antara AFTER atau BEFORE tidak akan banyak berpengaruh.

26.7 Latihan Trigger

Untuk menguji pemahaman seputar trigger serta SQL programming (yang sudah kita pelajari dalam bab tentang stored procedure), saya ingin mengajak anda untuk merancang sebuah trigger. Trigger kali ini akan menggunakan tabel nilai_mahasiswa.

Berikut perintah untuk membuat tabel nilai_mahasiswa:

```
DROP TABLE IF EXISTS nilai_mahasiswa;

CREATE TABLE nilai_mahasiswa (
    nim CHAR(8) PRIMARY KEY,
    nama VARCHAR(50),
    semester_1 DECIMAL(4,2),
    semester_2 DECIMAL(4,2),
    semester_3 DECIMAL(4,2)
);
```

Idenya adalah, ketika data mahasiswa diinput ke dalam tabel nilai_mahasiswa, buat sebuah trigger yang akan menghitung nilai IPK dari mahasiswa tersebut lalu simpan ke tabel baru.

Agar lebih menantang, selain nilai IPK, tampilkan juga kombinasi nama dan nim mahasiswa dengan format NAMA_MAHASISWA (nim) seperti RIANA PUTRIA (17090113). Nama mahasiswa ditulis menggunakan huruf besar.

Anda bebas untuk merancang tabel baru yang akan menampung nilai nama (nim) dan IPK ini. Sebagai contoh, misalkan saya input dua data berikut ke tabel nilai_mahasiswa:

```
INSERT INTO nilai_mahasiswa
VALUES ('17090113', 'Riana Putria', 3.12, 2.98, 3.45);

INSERT INTO nilai_mahasiswa
VALUES ('17140143', 'Rudi Permana', 2.56, 3.14, 3.22);
```

Maka hasilnya adalah sebagai berikut:

```
SELECT * FROM nilai_mahasiswa;
+-----+-----+-----+-----+
| nim      | nama          | semester_1 | semester_2 | semester_3 |
+-----+-----+-----+-----+
| 17090113 | Riana Putria |      3.12 |      2.98 |      3.45 |
| 17140143 | Rudi Permana |      2.56 |      3.14 |      3.22 |
+-----+-----+-----+-----+

SELECT * FROM ipk_mahasiswa;
+-----+-----+
| nama_dan_nip      | nilai_ipk |
+-----+-----+
| RIANA PUTRIA (17090113) |    3.18 |
| RUDI PERMANA (17140143) |    2.97 |
+-----+-----+
```

Dari tampilan diatas, saya menggunakan tabel ipk_mahasiswa untuk menampung hasil trigger. Silahkan rancang trigger ini. Sebagai tips, di dalamnya perlu menggunakan beberapa fungsi bawaan MySQL, seperti UPPER(), CONCAT() dan ROUND().

26.8 Jawaban Latihan Trigger

Sebagai penampung data trigger, saya menggunakan tabel ipk_mahasiswa. Berikut query untuk membuat tabel ini:

```
DROP TABLE IF EXISTS ipk_mahasiswa;

CREATE TABLE ipk_mahasiswa (
  nama_dan_nip VARCHAR(70) PRIMARY KEY,
  nilai_ipk DECIMAL(4,2)
);
```

Dan berikut kode program untuk trigger:

```
DELIMITER $$  
CREATE TRIGGER after_nilai_mahasiswa_insert  
AFTER INSERT ON nilai_mahasiswa  
FOR EACH ROW  
BEGIN  
  
DECLARE nama_nim_mahasiswa VARCHAR(50);  
DECLARE nilai_ipk DECIMAL(4,2);  
  
SET nama_nim_mahasiswa = CONCAT(UPPER(NEW.nama), " (" ,NEW.nim, ")");  
SET nilai_ipk = ROUND(((NEW.semester_1+NEW.semester_2+NEW.semester_3)/3),2);  
INSERT INTO ipk_mahasiswa VALUES (nama_nim_mahasiswa, nilai_ipk);  
  
END$$  
DELIMITER ;
```

Saya menamakan trigger ini dengan `after_nilai_mahasiswa_insert` karena trigger akan berjalan pada event `AFTER INSERT ON nilai_mahasiswa`.

Di dalam trigger, variabel `nama_nim_mahasiswa` dan `nilai_ipk` dipakai untuk menampung hasil penggabungan string `nama + nim` serta menampung hasil perhitungan IPK. Kedua isi variabel ini selanjutnya diinput ke dalam tabel `ipk_mahasiswa`.

Dengan demikian, setiap ada data mahasiswa baru yang diinput ke dalam tabel `nilai_mahasiswa`, otomatis akan mengisi tabel `ipk_mahasiswa`.

Dalam bab ini kita telah mempelajari cara membuat trigger di dalam MySQL. Umumnya trigger dipakai untuk membuat validasi, yakni memastikan data yang diproses sudah memenuhi kriteria tertentu. Selain itu trigger juga dipakai untuk membuat log (catatan aktifitas) dari suatu tabel.

Bab ini menutup materi lanjutan MySQL tentang **View**, **Stored Procedure** dan **Trigger**.

27. Pengantar phpMyAdmin

phpMyAdmin merupakan aplikasi MySQL client berbasis web dengan tampilan grafis. phpMyAdmin sangat populer dipakai karena praktis dan cepat untuk merancang database MySQL. Bukan tidak mungkin sebagian dari pembaca buku ini lebih familiar dengan phpMyAdmin daripada menulis query MySQL secara manual.

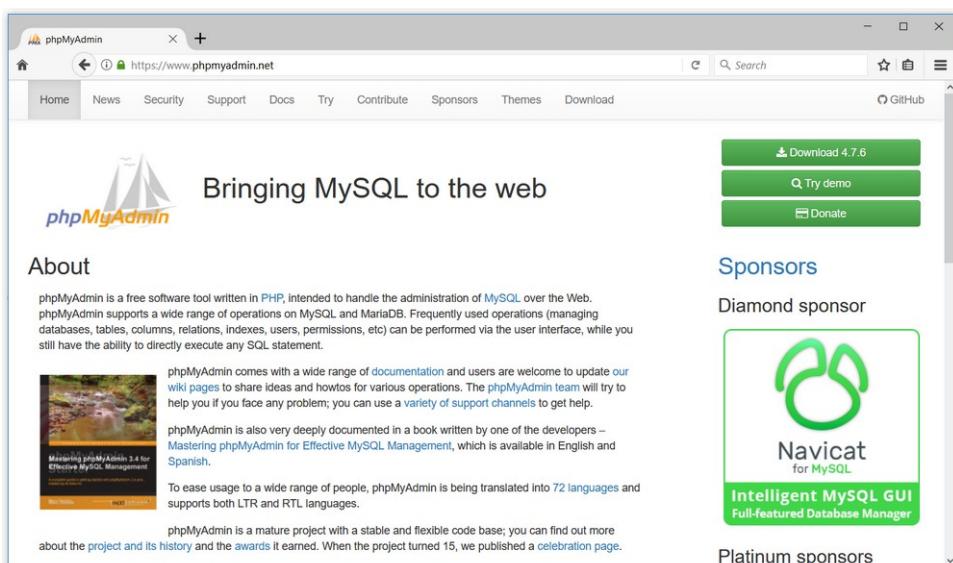
Dalam bab kali ini saya mencoba menyajikan materi pengantar seputar phpMyAdmin, yakni sekedar berkenalan dan membuat 1 tabel. phpMyAdmin sendiri sangat luas sehingga lebih cocok dibahas dalam 1 buku tersendiri.

27.1 Pengertian phpMyAdmin

phpMyAdmin adalah sebuah aplikasi manajemen database MySQL dan MariaDB yang berbasis web. Dengan phpMyAdmin kita bisa membuat database, tabel, export dan import data, menulis trigger, serta berbagai hal lain. Semuanya dilakukan menggunakan interface grafis yang sangat praktis.

Saat ini phpMyAdmin menjadi aplikasi standar pengelolaan database di dalam web hosting (menggunakan **cPanel**), yakni ketika ingin meng-onlinekan sebuah website. Aplikasi XAMPP sendiri juga menyertakan phpMyAdmin yang bisa langsung kita gunakan.

phpMyAdmin dibuat pada tahun 1998 oleh **Tobias Ratschiller**, namun pada tahun 2001 menjadi project open source yang dikembangkan oleh banyak programmer dari seluruh dunia. Website resmi phpMyAdmin berada di www.phpmyadmin.net¹.



Gambar: Website resmi phpMyAdmin di www.phpmyadmin.net

¹[https://www.phpmyadmin.net](http://www.phpmyadmin.net)

phpMyAdmin dirancang menggunakan bahasa pemrograman PHP dengan penggunaan AJAX yang cukup banyak, sehingga lebih pas disebut kombinasi PHP dan JavaScript.

27.2 Keunggulan dan Kekurangan phpMyAdmin

Tidak dapat dipungkiri membuat database dan tabel dengan phpMyAdmin jauh lebih mudah daripada menggunakan cmd Windows, terutama untuk database yang sederhana.

Dengan phpMyAdmin, kita disajikan tampilan grafis serta fitur tambahan seperti men-export database dalam bentuk zip dan pdf. Semua ini bisa dilakukan tanpa harus mengetahui perintah SQL apapun. Sangat praktis!

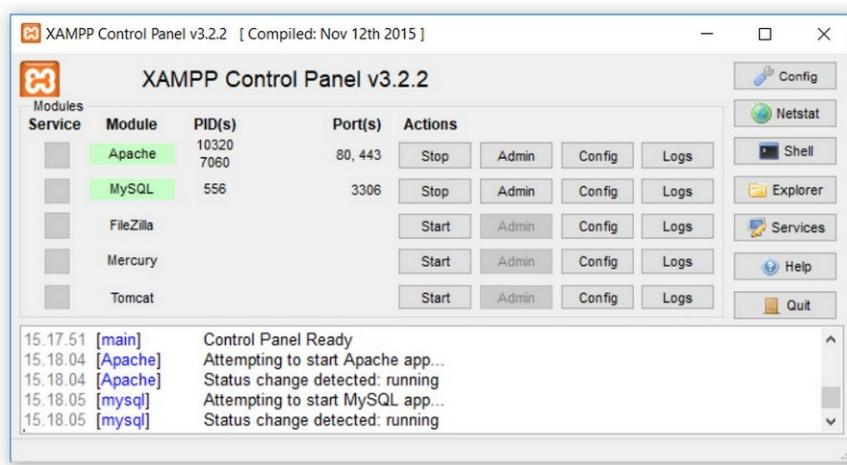
Hanya saja, saya berpendapat bahwa phpMyAdmin kurang cocok untuk proses belajar. phpMyAdmin baru pas dipakai untuk proses produksi, yakni setelah kita paham apa yang sebenarnya di jalankan oleh phpMyAdmin. Untuk web developer, skill menulis query SQL secara manual sangat penting, karena query inilah yang harus ditulis ke dalam kode program PHP.

Idealnya, kita mesti paham query dasar MySQL, serta juga bisa menggunakan phpMyAdmin untuk mempercepat proses pembuatan database, terlebih untuk website online.

27.3 Mengakses phpMyAdmin dari Localhost

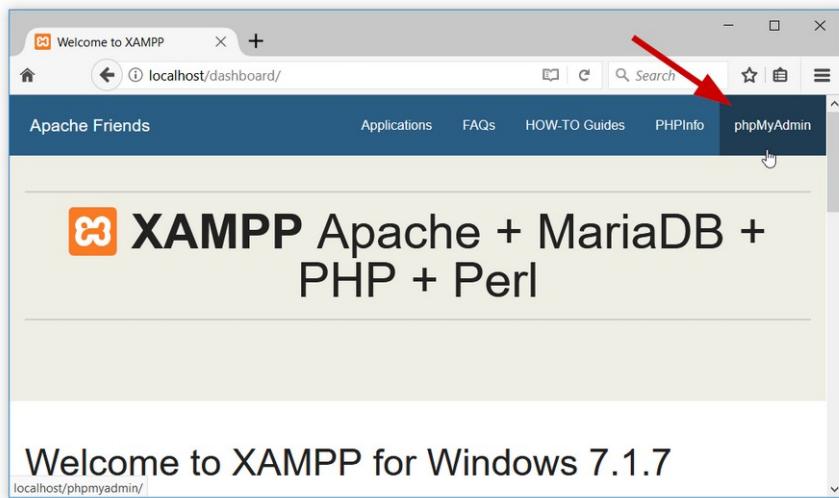
Jika anda memiliki hosting berbayar, phpMyAdmin bisa diakses dari dalam cPanel. Selain itu aplikasi XAMPP juga sudah menyertakan phpMyAdmin yang bisa berjalan secara offline,

Agar bisa mengakses phpMyAdmin, silahkan buka XAMPP Control Panel dan jalankan web server apache serta database server MySQL/MariaDB. Pastikan kedua server ini sudah running dan memiliki warna background hijau di dalam XAMPP Control Panel.



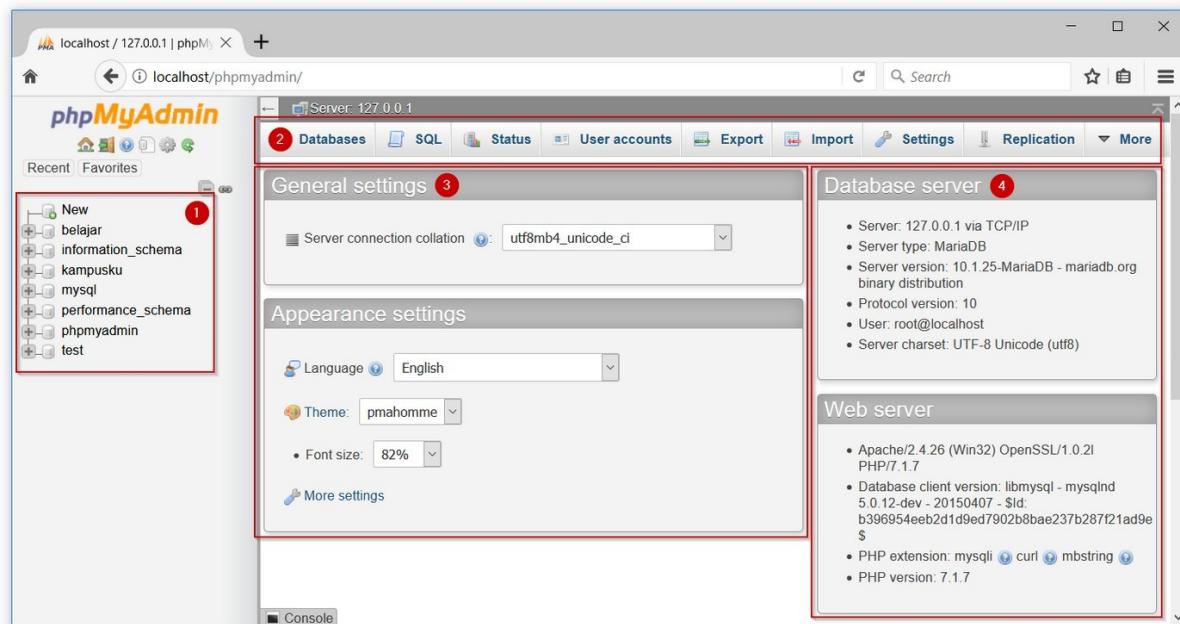
Gambar: Tampilan XAMPP Control Panel dengan apache dan mysql aktif

Berikutnya buka web browser lalu akses alamat: <http://localhost>. Jika tidak ada masalah, akan tampil halaman home dari XAMPP. Klik menu phpMyAdmin di sudut kanan atas.



Gambar: Menu link phpMyAdmin dari halaman localhost XAMPP

Sesaat kemudian akan tampil halaman awal dari phpMyAdmin:



Gambar: Halaman home dari phpMyAdmin

Halaman home atau halaman utama dari phpMyAdmin bisa dibagi menjadi 4 bagian.

Bagian pertama (1), merupakan menu yang menampilkan daftar seluruh database, tabel, view, stored procedure serta trigger yang ada di dalam MySQL. Tanda tambah (+) bisa diklik untuk menampilkan isi setiap database dengan lebih detail.

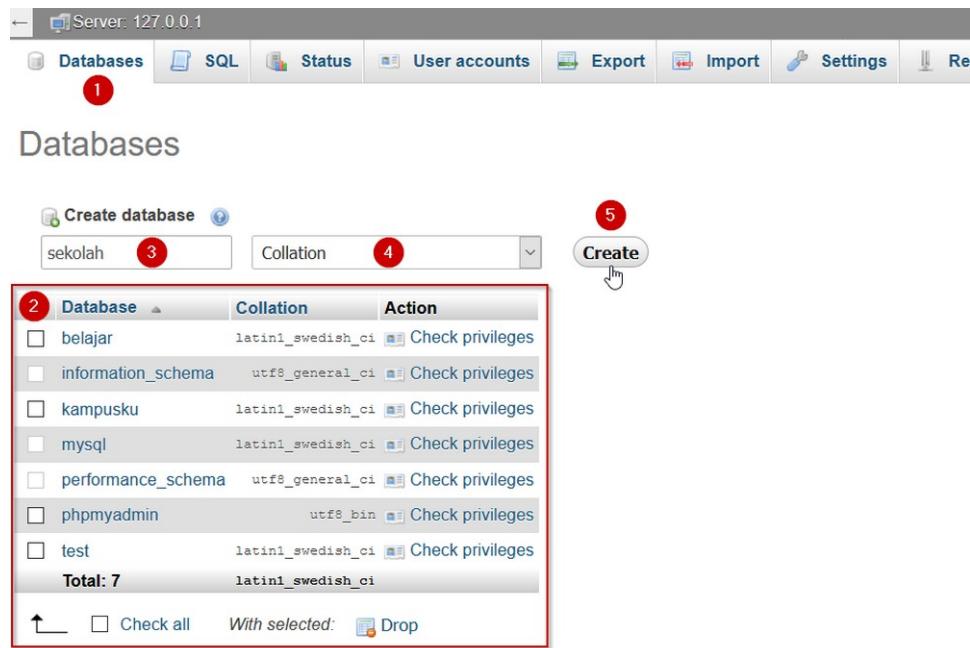
Bagian kedua (2), adalah menu utama dari phpMyAdmin. Pilihan menu ini bisa berubah-ubah tergantung apa yang sedang dipilih.

Bagian ketiga (3), adalah jendela utama yang juga akan berubah-ubah tergantung menu yang dipilih. Karena saya belum men-klik menu apapun, yang tampil adalah jendela "General Setting" MySQL.

Bagian keempat (4), berisi informasi tentang status web server yang sedang aktif. Disini tertera versi MySQL atau MariaDB yang dipakai, termasuk versi apache dan PHP. Dalam contoh ini saya menggunakan **MariaDB 10.1.25**, **Apache 2.4.26**, serta **PHP 7.1.7**.

27.4 Membuat Database dan Tabel

Kali ini kita akan coba membuat database dan tabel dari phpMyAdmin. Mari mulai dari database terlebih dahulu.



Gambar: Membuat database menggunakan phpMyAdmin

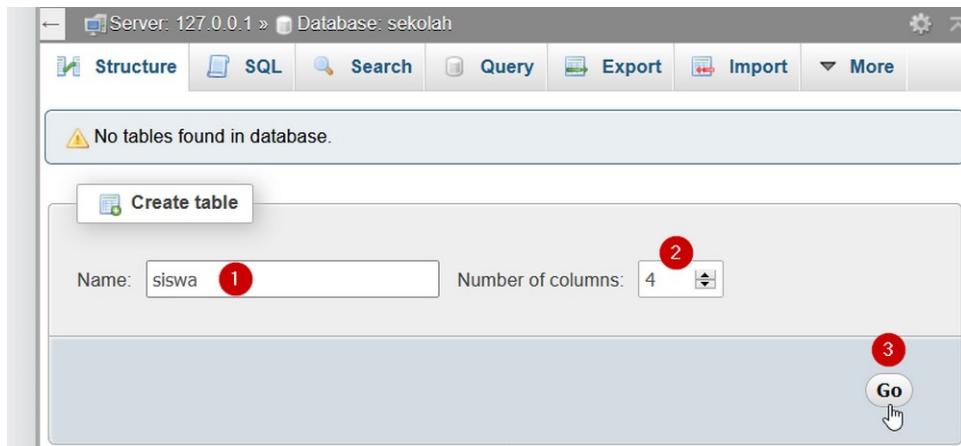
Untuk membuat database, klik menu **Databases** di bagian atas halaman home (1). Selanjutnya akan tampil jendela seperti gambar diatas.

Halaman ini dipakai untuk me-manajemen database (2). Di bawah kotak input pembuatan database, terdapat daftar seluruh database yang ada di dalam MySQL. Kita bisa menghapus database, atau klik salah satu database untuk melihat tabel apa saja yang ada di dalamnya.

Untuk membuat database baru, ketik nama database di isian kotak input **Create database**. Dalam contoh ini saya akan membuat database **sekolah** (3).

Disampingnya (4), terdapat menu pilihan untuk **Collation**. Pilihan ini sebenarnya sudah gabungan antara *charset* dan *collation*. Disini kita bisa menentukan apa charset dan collation yang akan dipakai. Atau biarkan pilihan default (Collation) untuk menggunakan settingan bawaan phpMyAdmin.

Terakhir, klik tombol **Create** untuk membuat database **sekolah** (5).



Gambar: Membuat tabel menggunakan phpMyAdmin

Setelah tombol **Create** dipilih, kita langsung diarahkan ke form untuk pembuatan tabel. Silahkan ketik nama tabel di kotak input **Name**. Sebagai contoh, saya akan membuat tabel **siswa** (1).

Disebelahnya (2), terdapat pilihan **Number of columns**. Ini adalah jumlah kolom untuk tabel yang akan dibuat. Saya berencana membuat **4 kolom** untuk tabel **siswa**.

Setelah isian nama dan jumlah kolom tabel, klik tombol **Go** untuk masuk ke jendela pembuatan struktur kolom (3).

Name	Type	Length/Values	Default	Collation	Attributes	Null	Index	A_I	Comments
nls	VARCHAR	10	None			PRIMARY			
nama	VARCHAR	100	None						
tanggal_lahir	DATE		None						
tinggi_badan	TINYINT		NULL		UNSIGNED				

Gambar: Membuat struktur tabel **siswa**

Jendela berikutnya adalah untuk menginput detail struktur kolom. Karena saya memilih 4 kolom, maka akan tampil 4 baris kotak input untuk setiap kolom pada tabel **siswa**. Jika ternyata anda berubah pikiran dan ingin menambah kolom lain, bisa mengisi kotak input di bagian paling atas (1).

Setiap baris terdapat berbagai isian yang bisa kita pilih (2). Berikut penjelasannya :

- **Name:** untuk inputan nama kolom.
- **Type:** tipe data dari kolom, seperti INT, VARCHAR, atau DATE. Hampir semua tipe data ini sudah kita pelajari, kecuali tipe data khusus Spatial untuk GIS (sistem informasi geografis).
- **Length/Values:** untuk batasan tipe data. Pilihan ini hanya berlaku untuk tipe data tertentu saja, seperti INT, CHAR atau VARCHAR. Sebagai contoh untuk membuat VARCHAR (50), angka 50 diisi pada kotak input ini.
- **Default:** Memilih nilai default untuk kolom.
- **Collation:** Memilih nilai collation untuk kolom (termasuk charset).
- **Attributes:** Menambahkan atribut khusus seperti UNSIGNED, BINARY, atau ZEROFILL. Sama seperti Length/Values, pilihan ini hanya untuk tipe data tertentu saja. Misalkan UNSIGNED dan ZEROFILL hanya bisa dipakai untuk tipe data number.
- **Null:** Apakah kolom bisa diisi nilai NULL atau tidak.
- **Index:** Apakah kolom akan menggunakan index tertentu, seperti PRIMARY KEY atau UNIQUE. Ketika memilih index, akan muncul jendela tambahan untuk menentukan nama index.
- **A_I :** Apakah kolom di set sebagai AUTO_INCREMENT. Pilihan ini hanya berlaku untuk tipe data number, terutama INT.
- **Comments:** Komentar untuk kolom, yakni berupa penjelasan tambahan. Komentar ini tidak akan diproses oleh MySQL, fungsinya hanya untuk catatan.

Selain pilihan diatas, terdapat juga **Virtuality**, **MIME type**, dan beberapa kolom lain. Kolom-kolom ini termasuk kategori advanced dan boleh diabaikan.

Tidak semua kolom harus berisi nilai, kita cukup memilih apa yang dirasa perlu saja. Untuk tabel siswa, saya membuat pengaturan sebagai berikut (3):

- Kolom nis: untuk NIS (nomor induk siswa), berupa tipe data VACHAR(10) dan diset sebagai PRIMARY KEY (pilih dari inputan index).
- Kolom nama: untuk nama siswa, berupa tipe data VARCHAR(100).
- Kolom tanggal_lahir: untuk tanggal lahir siswa, berupa tipe data DATE.
- Kolom tinggi_badan: untuk tinggi badan siswa, berupa TINYINT, nilai default NULL, atribut UNSIGNED, serta bisa diisi dengan nilai NULL.

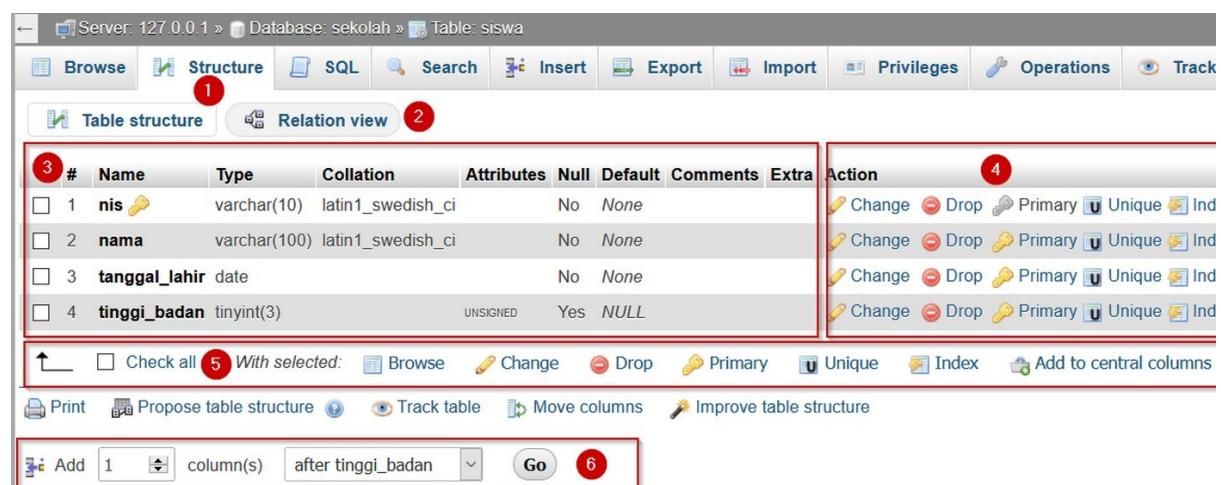
Setelah inputan struktur setiap kolom, dibawahnya (4) terdapat pilihan untuk **Table comments**, **Collation** serta **Storage Engine**. Disini kita bisa menginput komentar untuk kolom (hanya sekedar catatan), memilih collation, serta memilih storage engine. Misalnya ingin membuat tabel dengan tipe MyISAM, maka pengaturannya bisa diubah disini.

Untuk melihat apa query yang akan dipakai untuk membuat tabel, klik tombol **Preview SQL**. Berikut hasil query yang saya dapat (5):

```
CREATE TABLE `sekolah`.`siswa` (
  `nis` VARCHAR(10) NOT NULL ,
  `nama` VARCHAR(100) NOT NULL ,
  `tanggal_lahir` DATE NOT NULL ,
  `tinggi_badan` TINYINT UNSIGNED NULL DEFAULT NULL ,
  PRIMARY KEY (`nis`)
) ENGINE = InnoDB;
```

Tidak berbeda dengan query yang sudah kita pelajari sepanjang buku ini. Saya yakin anda juga sudah paham arti semua kode SQL ini.

Berikutnya, klik tombol **Save** untuk membuat tabel **siswa**.



Gambar: Pengelolaan struktur tabel **siswa**

Kita sampai ke halaman **Structure**. Ini adalah halaman untuk pengelolaan struktur kolom tabel (1).

Di sisi atas terdapat tombol **Relation view**, tombol ini bisa dipakai untuk menginput relasi ke dalam tabel, yakni membuat batasan *referential integrity* (2).

Struktur kolom sendiri ditampilkan di sisi kiri tabel (3), yakni berbagai informasi seperti nama kolom, tipe data, serta berbagai atribut kolom.

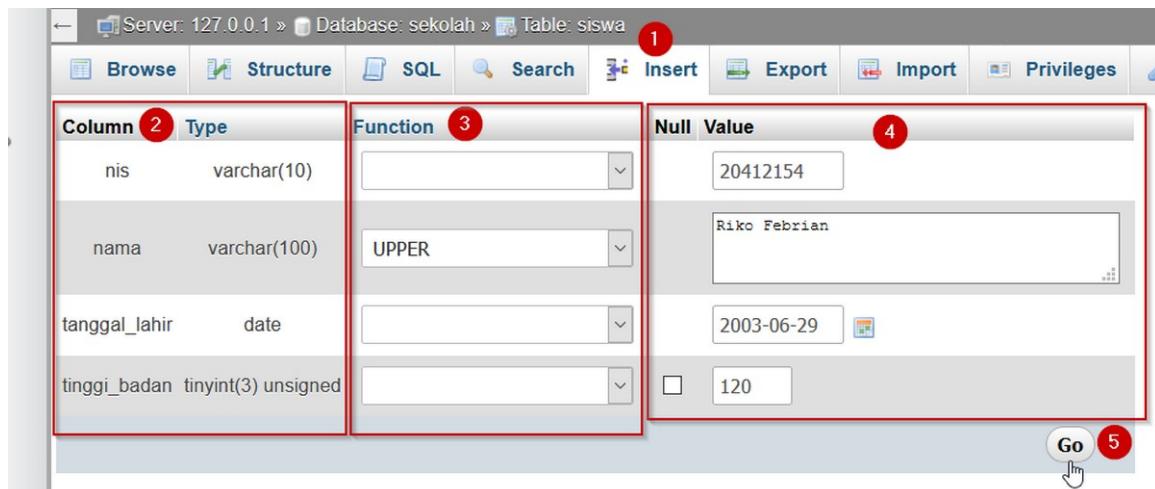
Di sebelah kanan tabel (4), terdapat beberapa tombol yang bisa dipakai untuk memanajemen struktur kolom, seperti mengubah kolom (CHANGE), menghapus kolom (DROP), menambahkan key, dll. Jika anda ingin mengubah kolom, maka akan dibawa ke tampilan seperti halaman sebelum ini.

Untuk mengubah banyak kolom sekaligus, bisa dengan men-checklist setiap kolom lalu klik tombol yang ada di bawah tabel (5). Jika ingin menambahkan 1 kolom, bisa diinput di baris paling bawah (6).

27.5 Mengisi Data Tabel

Tabel **siswa** sudah berhasil dibuat. Saatnya kita input beberapa data ke dalam tabel ini. Dari halaman struktur sebelumnya, klik menu **Insert**. Akan muncul jendela tampilan sebagai berikut

(1):



Gambar: Menambah data baru ke dalam tabel siswa

Pada halaman ini, terdapat 4 baris inputan sesuai dengan jumlah kolom di dalam tabel `siswa`. Paling kiri diawali dengan keterangan nama kolom (**Column**) serta tipe data (**Type**) kolom tersebut (2).

Selanjutnya diikuti dengan kolom **Function**. Kolom ini berisi pilihan berbagai fungsi bawaan MySQL yang bisa kita pakai (3). Mayoritas fungsi-fungsi ini sudah kita bahas. Sebagai contoh, saya memilih fungsi `UPPER` untuk kolom `nama` siswa.

Paling kanan berupa kotak isian nilai tabel (**Value**). Disinilah kita menginput data ke dalam tabel `siswa` (4). Sebagai data sample, saya akan menginput nilai berikut:

- nis: 20412154
- nama: Riko Febrian
- tanggal_lahir: 2003-06-29
- tinggi_badan: 120

Setelah seluruh kotak input diisi, klik tombol **GO** untuk memproses inputan data (5).

Tampilan akan berganti ke halaman **Query**, seperti tampilan dibawah ini (1):



Gambar: Konfirmasi query penambahan data

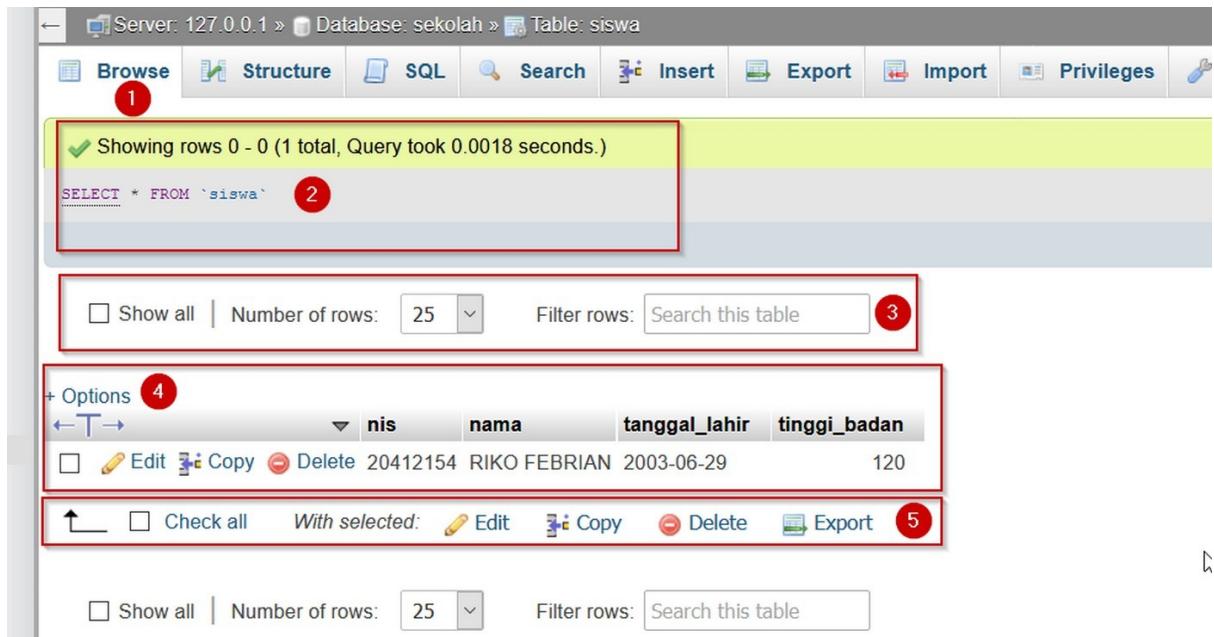
Halaman ini sekedar mengkonfirmasi bahwa query penambahan data sudah berhasil. Query yang dijalankan oleh phpMyAdmin adalah sebagai berikut:

```
INSERT INTO `siswa` (`nis`, `nama`, `tanggal_lahir`, `tinggi_badan`)
VALUES ('20412154', UPPER('Riko Febrian'), '2003-06-29', '120');
```

Kembali, jika anda sudah paham bab-bab sebelumnya, kode SQL diatas bisa dipahami dengan mudah. Perhatikan bawah phpMyAdmin menggunakan tanda backtick ke dalam semua nama kolom.

27.6 Menampilkan Isi Tabel

Untuk menampilkan isi tabel menggunakan phpMyAdmin, akses menu **Browse** dan akan tampil jendela sebagai berikut (1):



Gambar: Menampilkan isi tabel siswa

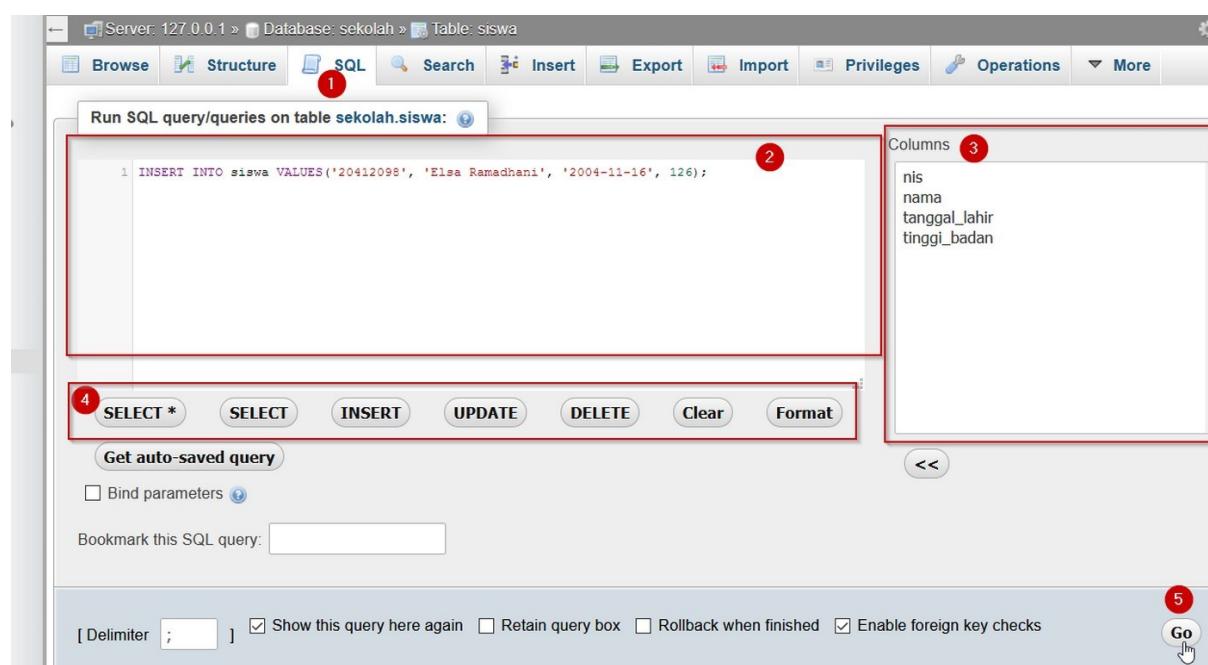
Di bagian atas (2), terdapat review dari query yang dijalankan oleh phpMyAdmin. Dalam contoh ini berupa `SELECT * FROM siswa` yang artinya akan menampilkan seluruh isi tabel `siswa`.

Di bawahnya (3) terdapat pilihan untuk mengatur jumlah baris tabel serta kotak input pencarian.

Isi tabel itu sendiri ada di bagian (4). Disini terdapat beberapa tombol untuk perubahan data, seperti **Edit**, **Copy** dan **Delete**. Jika ingin mengubah banyak data sekaligus, bisa dengan memilih baris yang diinginkan (tombol checklist), kemudian gunakan link di bagian bawah (5).

27.7 Menjalankan Kode SQL

Selain membuat tabel dan menambahkan data dengan tampilan grafis, kita juga bisa menginput manual kode SQL ke dalam phpMyAdmin. Caranya, klik menu **SQL** dan akan tampil jendela sebagai berikut (1):



Gambar: Menjalankan query **INSERT** secara manual

Dalam jendela ini terdapat kotak inputan untuk menulis query MySQL (2). Disini kita bisa mengetik kode query secara manual. Sebagai contoh, saya ingin menginput satu data lagi ke dalam tabel `siswa` menggunakan query berikut:

```
INSERT INTO siswa VALUES('20412098', 'Elsa Ramadhani', '2004-11-16', 126);
```

Di sampingnya (3), terdapat daftar nama kolom. Dengan men-klik salah satu daftar ini, nama kolom akan muncul di kotak input query. Ini tidak lain merupakan ‘shortcut’ untuk mempercepat penulisan nama kolom serta menghindari salah ketik.

Di bawah (4) terdapat beberapa tombol seperti **SELECT ***, **SELECT**, **INSERT**, **UPDATE** dan **DELETE**. Kelima tombol ini juga merupakan shortcut atau template dari perintah dasar SQL. Sebagai contoh, ketika saya men-klik tombol **INSERT**, tampil kode berikut:

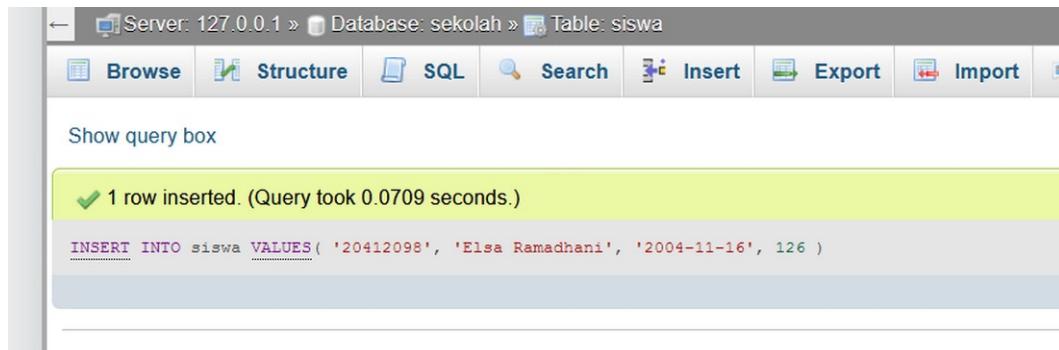
```
INSERT INTO `siswa`(`nis`, `nama`, `tanggal_lahir`, `tinggi_badan`)
VALUES ([value-1],[value-2],[value-3],[value-4])
```

Dengan demikian, untuk menginput data baru ke tabel siswa kita tinggal mengganti nilai [value-1], [value-2], [value-3] dan [value-4] dengan nilai yang sebenarnya. Ini berguna seandainya kita lupa seperti apa penulisan query INSERT.

Tombol **Clear** berfungsi untuk menghapus/membersihkan kotak inputan SQL. Tombol **Format** untuk men-format teks query akan menjadi lebih rapi. Berikut hasil format dari query `INSERT INTO siswa VALUES('20412098', 'Elsa Ramadhani', '2004-11-16', 126)` :

```
INSERT
INTO
    siswa
VALUES(
    '20412098',
    'Elsa Ramadhani',
    '2004-11-16',
    126
);
```

Untuk menjalankan kode SQL, klik tombol **GO** yang ada di sudut kanan bawah (5).



Gambar: Konfirmasi query

Halaman berikutnya akan menampilkan pesan konfirmasi apakah query berhasil, atau terdapat error.

Untuk melihat isi tabel siswa, kembali klik menu **Browser (1)**:

nis	nama	tanggal_lahir	tinggi_badan
20412098	Elsa Ramadhani	2004-11-16	126
20412154	RIKO FEBRIAN	2003-06-29	120

Gambar: Isi dari tabel siswa

Isi dari tabel siswa akan tampil di bagian tengah halaman (2).

Dalam bab ini kita telah melihat penggunaan singkat dari **phpMyAdmin**. Banyaknya fitur-fitur yang ada di dalam MySQL membuat tampilan phpMyAdmin cukup “ramai”.

Silahkan coba telusuri lebih jauh menu-menu yang ada di dalam phpMyAdmin. Terutama untuk melakukan hal-hal berikut:

- Membuat dan menghapus database.
- Membuat dan menghapus tabel.
- Menambah dan menghapus data tabel.
- Update data tabel.
- Export dan import.

Kelima hal inilah yang akan sering kita gunakan sehari-hari. Dikombinasikan dengan pemahaman perintah SQL dasar (yang sudah dibahas sepanjang buku ini), keduanya menjadi modal yang kuat untuk mulai merancang aplikasi dengan database MySQL / MariaDB.

Penutup: MySQL Uncover

Tidak terasa, sudah lebih dari 600 halaman saya membahas MySQL di buku **MySQL Uncover** ini.

Sedikit banyak semoga anda bisa mendapat gambaran tentang apa itu MySQL dan apa yang bisa dilakukan dengan MySQL. Materi yang ada sudah bisa jadi pondasi kuat untuk merancang aplikasi yang melibatkan database MySQL maupun MariaDB.

Tapi ini sebenarnya belum cukup. Jika anda serius ingin mendalami tentang database, saya sarankan untuk coba lanjut belajar tentang **teori database** itu sendiri. Misalnya, bagaimana cara merancang database yang baik, apa itu diagram ERD, normalisasi database, konsep relasi antar tabel, dst.

Ibarat seorang pelukis, ada 2 hal mesti dikuasai. Pertama, dia harus mahir menggunakan alat lukis dan paham berbagai jenis canvas dan karakteristiknya. Kedua, dia harus punya dasar teori melukis seperti konsep golden ratio, komposisi warna, pencahayaan, dst.

Jika pelukis itu tidak punya teori design, hasil gambarnya tidak akan bagus, meskipun ini bisa dipelajari secara trial-error (coba-coba). Jika pelukis itu tidak paham cara penggunaan alat, lukisannya juga terasa kurang pas. Bisa jadi warnanya akan luntur dalam beberapa hari karena salah memilih jenis canvas dan cat.

Begitu juga dengan database. Apa yang kita pelajari sepanjang buku ini adalah hal teknis cara penggunaan database server, yakni praktek menggunakan **MySQL / MariaDB**. Saya belum membahas dengan detail seputar teori database, yakni bagaimana cara mendesain database yang baik, karena itu ada di sisi teori.

Misalkan saya ingin membuat sebuah aplikasi sistem informasi sekolah. Berapa tabel yang harus dibuat? apakah setiap kelas disimpan ke dalam satu tabel (ada 9 tabel untuk 9 kelas), atau seluruh siswa di gabung dalam 1 tabel saja? bagaimana membuat tabel untuk daftar mata pelajaran? apakah langsung ditulis nama guru, atau dibuat satu tabel khusus yang berisi nama guru? bagaimana dengan pencatatan gaji guru?

Untuk menjawab semua pertanyaan ini, kita harus bahas dengan teori perancangan database. Mudah-mudahan suatu saat nanti dunia ilkom bisa menerbitkan buku khusus tentang teori database seperti ini yang mungkin bisa digabung dengan studi kasus. Dipadu dengan pemahaman teknis dari buku **MySQL Uncover**, kita bisa langsung praktek menggunakan MySQL atau MariaDB.

Skill lain yang sebaiknya perlu dikuasai adalah tentang **phpMyAdmin**, karena ini merupakan standar aplikasi standar untuk manajemen database MySQL atau MariaDB di web hosting, yakni ketika ingin meng-onlinekan sebuah website.

Bagaimana dengan database server lain seperti **Oracle** atau **Microsoft SQL Server**? Itu juga bisa dipelajari, terutama jika anda ingin membuat aplikasi desktop yang cukup besar. Untuk web programming, MySQL masih mendominasi. Selain itu tersedia juga alternatif database lain, seperti **PostgreSQL**, **SQLite** dan **MongoDB**.

Akhir kata, semoga materi yang ada di buku **MySQL Uncover** ini bisa bermanfaat. Mohon maaf jika ada kata-kata yang salah dan kekurangan di sana-sini. Jika ada saran atau koreksi materi, bisa menghubungi saya via email ke duniailkom@gmail.com.

Terimakasih atas dukungannya dengan membeli versi asli eBook **MySQL Uncover**. Semoga ilmu yang di dapat berkah dan bermanfaat. Sampai jumpa di buku DuniaIlkom selanjutnya :)

Daftar Pustaka

Sepanjang penulisan buku **MySQL Uncover**, saya mengumpulkan bahan dari berbagai sumber. Anda bisa mengunjungi daftar pustaka ini untuk menambah pengetahuan seputar MySQL atau MariaDB.

Khusus untuk daftar pustaka yang berbentuk buku / eBook luar negeri, saya akses secara legal dengan berlangganan di safaribooksonline.com (sekitar US \$200 per tahun).

- DuBois, Paul. **MySQL, Fifth Edition**. Addison-Wesley Professional, 2013. ISBN: 9780321833877.
- J.T. Dyer, Russell. **Learning MySQL and MariaDB**. O'Reilly Media, Inc, 2015. ISBN: 9781449362904.
- **MySQL 5.7 Reference Manual**: <https://dev.mysql.com/doc/refman/5.7/en/>
- **MariaDB Documentation**: <https://mariadb.com/kb/en/library/documentation/>
- **W3schools SQL Tutorial**: <https://www.w3schools.com/sql/default.asp>
- **MySQL Tutorial**: <http://www.mysqltutorial.org/>