# Operators

Berikut operator-operator yang sering digunakan:

|  |  |
| --- | --- |
| **Operator** | **Deskripsi** |
| **AND, &&** | Logical AND |
| **BETWEEN ... AND ...** | Mengecek apakah suatu nilai berada pada range nilai. |
| **/** | Operator pembagian |
| **=** | Operator penyamaan |
| **>,>=** | Lebih besar, lebih besar sama dengan |
| **<, <=** | Lebih kecil, lebih kecil sama dengan |
| **IS** | Menguji suatu nilai terhadap suatu boolean |
| **IS NOT** | Menguji suatu nilai terhadap suatu boolean |
| **IS NOT NULL** | Menguji nilai NOT NULL |
| **IS NULL** | Menguji nilai NULL |
| **LIKE** | Pembandingan pola yang sederhana |
| **-** | Operator pengurangan |
| **%, MOD** | Operator sisa hasil bagi |
| **NOT, !** | Menegasikan nilai |
| **NOT BETWEEN ... AND**  **...** | Mengecek apakah sutu nilai tidak berada pada range nilai |
| **||, OR** | Logical OR |
| **+** | Operator Penjumlahan |
| **REGEXP** | Pencocokan pola menggunakan *regular expressions* |
| **!=, <>** | Operator tidak sama dengan |
| **\*** | Operator perkalian |
| **XOR** | Logical XOR |

Fungsi string sering digunakan untuk mengelola kolom dan isi data yang akan ditampilkan (digunakan di klausa SELECT) maupun untuk melengkapi filter data (digunakan di klausa WHERE). Berikut ini operator string yang sering digunakan:

|  |  |
| --- | --- |
| **Fungsi** | **Deskripsi** |
| **CHAR\_LENGTH()** | Mengembalikan jumlah karakter |
| **CONCAT(*str1, str2, ...*)** | Menggabungkan string |
| **INSERT(*str, pos, len, newstr*)** | Menyisipkan *newstr* pada posisi dan panjang tertentu dari string |
| **LOWER(), LCASE()** | Mengubah string menjadi lower case |
| **LEFT(*str, len*)** | Mengembalikan sekian karakter dari sisi kiri string |
| **LOCATE(*substr, str, pos*)** | Mengembalikan letak penemuan pertama dari suatu substring dimulai dari posisi tertentu |
| **LPAD(*str, len, padstr*)** | Menempelkan *padstr* ke kiri *str* dengan total panjang karakter setelah penempelan sejumlah *len* |
| **LTRIM(*str*)** | Menghapus spasi di awal |
| **MID(*str, pos, len*), SUBSTR(*str, pos, len*)** | Mengembalikan suatu substring dimulai dari posisi tertentu |
| **POSITION(*substr* IN *str*)** | Sama dengan LOCATE(substr, str) |
| **REPEAT(*str, count*)** | Mengulangi string sebanyak *count* yang ditentukan |
| **REPLACE(*str, from\_str, to\_str*)** | Mengembalikan *str* dengan menggantikan string awal  *from\_str* menjadi *to\_str* |
| **REVERSE(*str*)** | Membalik susunan karakter dari suatu string |
| **RIGHT(*str, len*)** | Mengembalikan sekian karakter dari sisi kanan string |
| **RPAD(*str, len, padstr*)** | Menempelkan *padstr* ke kanan *str* dengan total panjang karakter setelah penempelan sejumlah *len* |
| **RTRIM(*str*)** | Menghapus spasi di belakang string |
| **STRCMP()** | Membandingkan dua strings |
| **SPACE(*N*)** | Mengembalikan spasi sebanyak *N* |
| **SUBSTRING\_INDEX(*str,***  ***delim, count)*** | Mengembalikan suatu substring dari string sebelum sejumlah delimiter muncul. Nilai *count* minus berarti dihitung dari kanan |
| **TRIM({BOTH|LEADING| TRAILING} [*remstr*] FROM**  ***str*)** | Menghapus spasi atau *remstr* dari *str*, dari sisi kiri (leading), sisi kanan (trailing) atau keduanya (both) |
| **UPPER(*str*)** | Mengubah string menjadi upper case |

# Numeric Functions

Fungsi numerik digunakan untuk mengelola nilai data. Fungsi ini bisa digunakan di klausa SELECT maupun untuk melengkapi filter data di klausa WHERE. Berikut ini fungsi numerik yang sering digunakan:

|  |  |
| --- | --- |
| **Fungsi** | **Deskripsi** |
| **ABS(*x*)** | Mengembalikan nilai absolut |
| **ACOS(*x*), ASIN(*x*), ATAN(*x*)** | Mengembalikan arc cosinus, arc sinus, arc tangen |
| **CEILING(*x*)** | Mengembalikan pembulatan ke atas |
| **COS(*x*), SIN(*x*), TAN(*x*), COT(*x*)** | Mengembalikan cosinus, sinus, tangen, cotangen. Nilai x dalam radian. |
| **DEGREES(*x*)** | Mengkonversi radian ke derajat |
| **DIV** | Pembagian integer |
| **EXP(*x*)** | Mengembalikan bilangan *e* pangkat *x* |
| **FLOOR(*x*)** | Mengembalikan pembulatan ke bawah |
| **LN(*x*)** | Mengembalikan logaritma natural dari *x* |
| **LOG(*b, x*)** | Mengembalikan logaritma basis *b* dari *x* |
| **MOD(*n, m*)** | Mengembalikan sisa hasil bagi dari *n* dibagi *m* |
| **PI()** | Mengembalikan nilai pi |
| **POWER(*x, y*)** | Mengembalikan nilai *x* pangkat *y* |
| **RADIANS(*x*)** | Mengkonversi derajat ke radian |
| **RAND()** | Mengembalikan nilai floating-point acak |
| **ROUND(*x, d*)** | Membulatkan *x* hingga sampai *d* angka desimal |
| **SIGN(*x*)** | Mengembalikan tanda positif/negatif |
| **SQRT(*x*)** | Mengembalikan akar kuadrat dari *x* |
| **TRUNCATE(*x*, *d*)** | Memotong angka desimal hingga *d* angka di belakang  koma |

Selain yang tercantum di atas, masih ada beberapa fungsi numerik yang disebut sebagai fungsi agregasi antara lain SUM, AVG, COUNT, dan lain-lain yang akan kita bahas pada pertemuan berikutnya.

# Date & Time Functions

Struktur data penanggalan dan pewaktuan di MySQL lebih mudah dikelola dibandingkan di Oracle maupun DBMS lainnya, karena kita bisa memperlakukannya layaknya karakter biasa. Namun tentunya untuk menghindari kesalahan pengolahan, kita tetap membutuhkan fungsi-fungsi khusus penanggalan dan pewaktuan

|  |  |  |
| --- | --- | --- |
| **Fungsi** | **Deskripsi** | |
| **ADDDATE(*date,* INTERVAL *n***  ***expr\_unit*)** | Menambah *n expr\_unit* dari *date* yang dimasukkan.  *Expr\_unit* bisa berupa DAY/WEEK/MONTH/ QUARTER/YEAR | |
| **ADDTIME(*expr1, expr2*)** | Menambah waktu *expr2* terhadap waktu *expr1* | |
| **CONVERT\_TZ(*datetime, from\_tz, to\_tz*)** | Mengkonversi zona waktu | |
| **CURDATE(), CURRENT\_DATE()** | Mengembalikan tanggal saat perintah dijalankan, diberikan dalam bentuk YYYY-MM-DD | |
| **CURTIME(), CURRENT\_TIME()** | Mengembalikan waktu saat perintah dijalankan | |
| **CURRENT\_TIMESTAMP(), NOW()** | Mengembalikan timestamp saat perintah dijalankan | |
| **DATE(*expr*)** | Mengekstrak date dari format datetime | |
| **DATEDIFF(*expr1, expr2*)** | Menghitung selisih hari antara *expr1* dan *expr2* | |
| **DATE\_FORMAT(*date, format*)** | Memformat date berdasarkan format tertentu. Penulisan format dalam tanda petik ‘’. | |
| **%a** | Singkatan nama hari (Sun..Sat) |
| **%b** | Singkatan nama bulan (Jan..Dec) |
| **%c** | Bulan, numerik (0..12) |
| **%D** | Tanggal dalam bentuk ke- (1st, 2nd, ...) |
| **%d** | Tanggal, numerik (00..12) |
| **%e** | Tanggal, numerik (0..12) |
| **%f** | Mikrodetik |
| **%H** | Jam, 24 jam (00..23) |
| **%h** | Jam, 12 jam (01..12) |
| **%i** | Menit, numerik (00..59) |
| **%j** | Hari dari tahun (001..366) |
| **%k** | Jam (0..23) |
| **%l** | Jam (1..12) |
| **%M** | Nama bulan (Januari..December) |
| **%m** | Bulan, numerik (00..12) |
| **%p** | AM atau PM |
| **%r** | Waktu, 12 jam diikuti AM atau PM |
| **%S** | Detik (00..59) |
| **%s** | Detik (00..59) |
| **%T** | Waktu, 24 jam (hh:mm:ss) |
| **%U** | Pekan (00..53), Sunday hari pertama |
| **%u** | Pekan (00..53), Monday hari pertama |
| **%V** | Pekan (01..53), Sunday hari pertama |
| **%v** | Pekan (01..53), Monday hari pertama |
| **%W** | Nama weekday (Sunday..Saturday) |
| **%w** | Hari dari pekan (0=Sunday..6=Saturday) |
| **%X** | Tahun untuk pekan, Sunday hari pertama |
| **%x** | Tahun untuk pekan, Monday hari pertama |
| **%Y** | Tahun, numerik 4 digit |
| **%y** | Tahun, numerik 2 digit |
| **DAY(*date*), DAYOFMONTH(*date*)** | Mengembalikan tanggal dari bulan | |
| **DAYOFWEEK(*date*)** | Mengembalikan hari ke- dari pekan (1..7) | |
| **DAYOFYEAR(*date*)** | Mengembalikan hari ke- dari tahun (1..366) | |

|  |  |
| --- | --- |
| **EXTRACT(*unit* FROM *date*)** | Mengekstrak *unit* dari *date*. Unit berupa DAY/MONTH/YEAR/HOUR/MINUTE/SECOND atau  gabungan (misal YEAR\_MONTH atau DAY\_MINUTE) |
| **HOUR(*time*)** | Mengekstrak jam dari time |
| **LAST\_DAY(*date*)** | Mengembalikan tanggal terakhir dari bulan |
| **MINUTE(*time*)** | Mengekstrak menit dari time |
| **MONTH(*date*)** | Mengekstrak bulan dari date (1..12) |
| **MONTHNAME(*date*)** | Mengekstrak nama bulan dari date |
| **QUARTER(*date*)** | Mengembalikan paruh semester dari date |
| **SECOND(*time*)** | Mengembalikan detik dari time |
| **STR\_TO\_DATE(*str, format*)** | Mengubah string menjadi date format tertentu. Contoh: STR\_TO\_DATE(‘May 1, 2017’, ‘%M %d, %Y’)  menghasilkan 2017-03-01 |
| **SUBDATE(*date,* INTERVAL *n***  ***expr\_unit*)** | Kebalikan dari ADDDATE |
| **SUBTIME(*expr1, expr2*)** | Kebalikan dari ADDTIME |
| **SYSDATE()** | Hampir mirip dengan NOW(), namun SYSDATE() bisa  memberikan hasil yang berbeda walau dijalankan dalam satu statemen |
| **TIME(*datetime*)** | Mengembalikan time |
| **TIMEDIFF(*datetime1, datetime2*)** | Mengurangkan *datetime1* dengan *datetime2* dalam bentuk time |
| **TIMESTAMP(*date*)** | Membentuk *date* menjadi YYYY-MM-DD HH:MM:SS |
| **TIMESTAMPDIFF(*unit,***  ***datetime1, datetime2*)** | Mengurangkan *datetime1* dengan *datetime2* dalam *unit* yang diinginkan. Unit bisa berupa DAY/MONTH/YEAR/HOUR/MINUTE/SECOND |
| **TIMEFORMAT(*time, format*)** | Cara kerja mirip DATEFORMAT |
| **TO\_DAYS(*date*)** | Mengembalikan hitungan hari dimulai dari tahun 0 gregorian (1582 Masehi) |
| **UTC\_DATE()** | Mengembalikan tanggal universal (Temps Universel Coordonne |
| **UTC\_TIME()** | Mengembalikan waktu universal |
| **UTC\_TIMESTAMP()** | Mengembalikan timestamp universal |
| **WEEK(*date*)** | Mengembalikan pekan (hari pertama Sunday, 0..53) |
| **WEEKDAY(*date*)** | Mengembalikan indeks weekday (0=Monday..6=Sunday) |
| **WEEKOFYEAR(*date*)** | Sama dengan WEEK dengan hari pertama Monday,  1..53 |
| **YEAR(*date*)** | Mengembalikan tahun |
| **YEARWEEK(*date*)** | Mengembalikan tahun dan pekan |

# Cast Functions

Fungsi cast adalah fungsi untuk memaksakan perubahan tipe data dari tipe data yang sudah berlaku sebelumnya. Satu cast function yang paling sering dipakai adalah **CONVERT(*expr*, *type*)** dengan *type* sebagai berikut:

|  |  |
| --- | --- |
| **Type CONVERT** | **Deskripsi** |
| **BINARY(*N*)** | Menghasilkan string dengan tipe data binary |
| **CHAR(*N*)** | Menghasilkan string dengan tipe data char |
| **DATE** | Menghasilkan nilai date |
| **DATETIME** | Menghasilkan nilai datetime |
| **DECIMAL(*M,D*)** | Menghasilkan nilai desimal dengan total panjang digit *M* dan  menggunakan *D* angka di belakang koma |
| **JSON** | Menghasilkan nilai JSON |
| **SIGNED INTEGER** | Menghasilkan nilai integer bertanda |
| **UNSIGNED INTEGER** | Menghasilkan nilai integer tak bertanda |
| **TIME** | Menghasilkan nilai time |

Latihan:

* 1. Ambil nilai terbesar dari NIM dari prodi Informatika, lalu lakukan increment nilai tersebut sebanyak 1 untuk menciptakan NIM mahasiswa yang baru saja mendaftar.

# Control-Flow Functions

Seperti bahasa pemrograman pada umumnya, SQL juga memiliki fungsi kontrol alur.

|  |  |
| --- | --- |
| **Fungsi** | **Deskripsi** |
| **CASE WHEN *condition* THEN *result* [WHEN *condition* THEN *result*] ... [ELSE *result*] END** | Fungsi selector CASE pada umumnya. Contoh penggunaan:  **SELECT CASE WHEN 1>0 THEN 'true' ELSE**  **'false' END** |
| **IF(*expr1, expr2, expr3*)** | Fungsi percabangan IF pada umumnya. Jika *expr1*  TRUE, maka akan mengembalikan *expr2,* jika FALSE maka akan mengembalikan *expr3.* |
| **IFNULL(*expr1, expr2*)** | Jika *expr1* tidak NULL, maka akan mengembalikan *expr2*. Jika NULL maka akan mengembalikan *expr2.* |
| **NULLIF(*expr1, expr2*)** | Mengembalikan NULL jika *expr1 = expr2* adalah TRUE. Jika FALSE maka akan mengembalikan  *expr1.* |

**OPERASI HIMPUNAN**

Operasi himpunan adalah operasi untuk menggabungkan dua tabel atau lebih sesuai prinsip matematika. Ada tiga jenis operasi himpunan yang akan dibahas, yaitu operasi union, minus, dan intersect.

# UNION

Union digunakan untuk menggabungkan hasil dari beberapa statemen SELECT menjadi satu set hasil. Nama kolom dari statemen SELECT yang pertama digunakan sebagai nama kolom dari hasil query. Pada SELECT berikutnya harus memuat kolom yang identik dengan kolom dari statemen SELECT yang pertama.

Konsep penggabungan sesuai dengan prinsip matematika berikut:

Prima = { 2, 3, 5, 7 } dan Genap = { 2, 4, 6, 8, 10 } maka Prima  Genap = { 2, 3, 5, 7, 4, 6, 8, 10 }

Operasi union sendiri memiliki 2 macam, yaitu **UNION** dan **UNION ALL**. Bedanya, UNION ALL memperbolehkan data yang sama dari 2 tabel muncul 2 kali. Menggunakan contoh yang tadi, hasil UNION ALL adalah {**2**, 3, 5, 7, **2**, 4, 6, 8, 10}

Sintaks dari UNION adalah:

**SELECT ... UNION [ALL] SELECT ...**

Contoh UNION:

**SELECT \* FROM prima UNION SELECT \* FROM genap**

Contoh UNION ALL:

**SELECT \* FROM prima UNION ALL SELECT \* FROM genap**

# MINUS

Minus digunakan untuk “memotong” himpunan dari hasil SELECT pertama oleh SELECT kedua. Konsep pemotongan adalah sebagai berikut:

Prima = { 2, 3, 5, 7 } dan Genap = { 2, 4, 5, 6, 7, 8, 10 } maka Prima – Genap = {3, 5, 7} Operator minus tidak didukung oleh MySQL, namun ada di Oracle dengan sintaks:

**SELECT ... MINUS SELECT ...**

Alternatifnya, kita bisa menggunakan statemen berikut:

**SELECT [kolom] FROM tabel1 LEFT JOIN tabel2 ON [penghubung antar tabel] WHERE tabel2.id IS NULL**

Contoh:

**SELECT \* FROM prima LEFT JOIN genap ON prima.bilangan = genap.bilangan WHERE genap.bilangan IS NULL**

# INTERSECT

Intersect digunakan untuk menunjukkan irisan dari himpunan hasil SELECT pertama yang juga ada di hasil SELECT kedua. Konsep irisan sesuai konsep matematika berikut:

Prima = { 2, 3, 5, 7 } dan Genap = { 2, 4, 6, 8, 10 } maka Prima  Genap = { 2 } Operator intersect didukung oleh MariaDB MySQL, dengan sintaks:

**SELECT ... INTERSECT SELECT ...**

Alternatifnya, kita bisa menggunakan statemen berikut:

**SELECT DISTINCT [kolom] FROM tabel1 INNER JOIN tabel2 USING(id)**

Contoh:

**SELECT DISTINCT \* FROM prima INNER JOIN genap USING(bilangan)**

Atau:

**SELECT \* FROM prima JOIN genap ON prima.bilangan = genap.bilangan**

Atau menggunakan subquery:

**SELECT \* FROM prima WHERE EXISTS(SELECT \* FROM genap WHERE prima.bilangan = genap.bilangan)**

# SUBQUERY

Subquery adalah statemen SELECT di dalam statemen lainnya. Berikut ini adalah contoh dari subquery:

**SELECT \* FROM t1 WHERE column1 = (SELECT column1 FROM t2);**

Pada contoh di atas, **SELECT \* FROM t1 WHERE column1 =** disebut sebagai *outer query* dan **(SELECT column1 FROM t2)** disebut sebagai *inner query* atau *subquery.* Outer query bisa berupa statemen SELECT, INSERT, UPDATE, DELETE, SET atau DO.

Kita bisa mengatakan bahwa subquery *nested/bersarang* di dalam outer query, dan sangat dimungkinkan untuk menyarangkan subquery lagi di dalam subquery, sampai batas kedalaman yang wajar. Subquery harus selalu berada di dalam tanda kurung (parentheses).

Keuntungan menggunakan subquery adalah:

* Menyediakan cara alternatif yang relatif lebih mudah untuk mendapatkan hasil yang diinginkan tanpa menggunakan join atau union yang kompleks.
* Membuat struktur penulisan query lebih mudah dibaca dan dibayangkan dibandingkan menggunakan join atau union.

Aturan Subquery:

* Klausa ORDER BY tidak boleh digunakan di subquery, namun ORDER BY dapat digunakan, di outer query (query induk).
* Ketika subquery diletakkan di klausa SELECT, maka hasil subquery semestinya adalah nama kolom. Kecuali untuk subquery-subquery menggunakan kata kunci EXIST.
* Ketika subquery diletakkan di klausa FROM, maka hasil subquery semestinya berupa tabel.
* Ketika subquery diletakkan di klausa WHERE, maka hasil subquery semestinya selaras melengkapi kondisi yang diberikan.
* Saat subquery adalah salah satu dari dua operan yang dilibatkan di pembandingan, subquery harus muncul di sisi kanan pembandingan.

Subquery bisa menghasilkan satu nilai skalar tunggal, satu kolom, satu baris, atau satu tabel. Outer query menggunakan operator yang berbeda-beda tergantung pada hasil subquery.

*Tabel 1.* Operator Pembanding

|  |  |  |
| --- | --- | --- |
| **Operator** | **Deskripsi** | **Row yang dihasilkan Subquery** |
| <> atau != | Tidak sama dengan | Single row |
| > / < | Lebih besar/lebih kecil dari |
| >= / <= | Lebih besar/lebih kecil atau sama dengan |
| ANY atau SOME, IN | Membandingkan sebuah nilai dengan SETIAP nilai dari hasil subquery. Penggunaannya harus diawali dengan operator =, !=, >, <, >=, atau <=.  Query akan mengembalikan nilai FALSE jika tidak ada row yang dihasilkan.  Operator “= ANY” ekuivalen dengan IN. | Multiple rows Multiple columns |
| ALL | Membandingkan sebuah nilai dengan SEMUA nilai dari hasil subquery. Penggunaannya harus diawali dengan operator =, !=, >, <, >=, atau <=.  Query akan mengembalikan nilai TRUE jika tidak ada row yang dihasilkan. | Multiple rows Multiple columns |
| EXISTS | Operator logikal jika digunakan untuk melihat apakah  nilai yang dihasilkan query utama BERADA dalam subquery. TRUE, jika mengembalikan row. |  |
| NOT EXISTS | Operator logikal jika digunakan untuk melihat apakah nilai yang dihasilkan query utama TIDAK BERADA  dalam subquery. TRUE, jika tidak mengembalikan row. |  |

Berikut ini contoh subquery menggunakan tabel-tabel dalam file

# prak\_sbd\_11\_subquery\_emp.sql.

Contoh penggunaan operator single row:

* Menampilkan nama mahasiswa yang memiliki nilai\_angka = 90.

**SELECT nama FROM mahasiswa WHERE nim = (SELECT nim FROM registrasi WHERE nilai\_angka = 90)**

* Menampilkan nama mahasiswa yang memiliki berat badan lebih berat daripada LUKE SKYWALKER.

**SELECT nama, beratbadan FROM mahasiswa WHERE beratbadan > (SELECT beratbadan FROM mahasiswa WHERE nama LIKE "LUKE%")**

* Menampilkan nama mahasiswa yang memiliki berat badan di bawah berat badan rata- rata mahasiswa.

**SELECT nama, beratbadan FROM mahasiswa WHERE beratbadan < (SELECT AVG(beratbadan) FROM mahasiswa)**

Menampilkan nama mahasiswa dari hasil pencarian nim dan berat badan dari mahasiswa yang memiliki berat badan di bawah 55 kg.

**SELECT nama FROM mahasiswa WHERE (nim, beratbadan) = ANY (SELECT nim, beratbadan FROM mahasiswa WHERE beratbadan < 55)**

ekuivalen dengan

**SELECT nama FROM mahasiswa WHERE (nim, beratbadan) IN (SELECT nim, beratbadan FROM mahasiswa WHERE beratbadan < 55)**

* Menampilkan nim dan nilai angka dari registrasi dimana nilai angka lebih besar dari setiap nilai angka antara 70 hingga 80.

**SELECT nim, nilai\_angka FROM registrasi WHERE nilai\_angka > ANY (SELECT nilai\_angka FROM registrasi WHERE nilai\_angka BETWEEN 70 AND 80) ORDER BY 1**

* Menampilkan nim dan nilai angka dari registrasi dimana nilai angka lebih besar dari semua nilai angka antara 70 hingga 80.

**SELECT nim, nilai\_angka FROM registrasi WHERE nilai\_angka > ALL (SELECT nilai\_angka FROM registrasi WHERE nilai\_angka BETWEEN 70 AND 80) ORDER BY 1**

* Bila ada mahasiswa yang nilai angkanya lebih besar dari 90, maka tampilkan seluruh isi tabel registrasi.

**SELECT \* FROM registrasi WHERE EXISTS (SELECT nilai\_angka FROM registrasi WHERE nilai\_angka > 90)**

* EXISTS juga bisa digunakan sebagai operator intersect.

**SELECT \* FROM a WHERE EXISTS(SELECT \* FROM b WHERE a.data=b.data)**

1. **Pengertian Trigger**

Trigger termasuk dalam *stored* program yang khusus pada basis data. Trigger merupakan salah satu bentuk *stored procedure* (yang akan kita pelajari pada pertemuan berikutnya) tapi perilakunya sedikit berbeda. *Stored program* pada basis data adalah

suatu urutan perintah yang ditulis, disimpan, dan dieksekusi di dalam sebuah basis data server, dalam hal ini adalah MySQL server.

Trigger adalah *stored program* yang **diaktifkan (invoke)** sebagai akibat dari suatu aktivitas yang terjadi di dalam suatu basis data. Diaktifkan dalam bahasa Inggris disebut **di-trigger**. Biasanya trigger akan di-invoke akibat dari operasi DML (INSERT, UPDATE, DELETE) pada suatu tabel dalam basis data. Trigger dapat juga digunakan dalam validasi data. Perbedaan utama antara *stored procedure* dan trigger adalah bahwa trigger dipanggil/dieksekusi otomatis saat terjadi operasi DML pada suatu tabel, sedangkan *stored procedure* harus dipanggil secara eksplisit.

**PERHATIAN:**

Pada MySQL Trigger tidak akan dipanggil / diinvoke jika tidak ada perintah SQL yang diproses pada database server MySQL. Sehingga trigger tidak akan bisa dipanggil misalnya dengan menggunakan API yang tidak melalui MySQL.

Trigger diimplementasikan dalam bentuk **fungsi/prosedur** yang akan dijalankan saat terjadi perubahan pada tabel. Karena trigger berada di dalam suatu tabel, maka suatu program di luar basis data tidak akan bisa mem-bypass trigger. Artinya sekali trigger dibuat di dalam tabel, maka programmer yang membuat program dari luar basis data (misalnya kode PHP) tidak akan bisa melewati trigger yang sudah dibuat sebelumnya.

1. **Keuntungan dan Kelemahan Trigger**

Keuntungan Trigger:

* 1. Trigger dapat menjadi cara alternatif untuk mengecek integritas data.
  2. Trigger dapat memvalidasi input data.
  3. Trigger dapat menangkap error pada aturan bisnis (*business logic*).
  4. Trigger dapat menjalankan program yang dijadwalkan karena secara otomatis dapat dipanggil saat terjadi operasi DML pada tabel.
  5. Trigger sangat berguna untuk mengaudit perubahan data pada tabel. Kelemahan Trigger:

1. Trigger hanya bisa menambah kemampuan validasi tambahan, tidak bisa menangani semua validasi. Artinya validasi utama tetap harus dibuat di level aplikasi (program), misalnya menggunakan PHP / Javascript, atau bahasa pemrograman lain.
2. Trigger dipanggil dan dieksekusi secara tidak terlihat dari aplikasi client, sehingga tidak dapat diketahui secara pasti trigger apa yang dibuat di dalam tabel dari program.
3. Trigger dapat menambah *overhead* pada basis data server, seperti MySQL.
4. **Implementasi Trigger**

Pada MySQL terdapat 6 macam trigger:

* BEFORE INSERT – dipanggil sebelum data dimasukkan ke dalam tabel
* AFTER INSERT – dipanggil setelah data dimasukkan ke dalam tabel.
* BEFORE UPDATE – dipanggil sebelum data diupdate di dalam tabel.
* AFTER UPDATE – dipanggil setelah data diupdate di dalam tabel.
* BEFORE DELETE – dipanggil sebelum data dihapus di dalam tabel.
* AFTER DELETE – dipanggil setelah data dihapus di dalam tabel. Pada Oracle, trigger bisa berisi SQL ataupun PL/SQL program.

Ilustrasi trigger adalah sebagai berikut:

Gambar 1. Trigger

Trigger biasanya digunakan untuk:

* Menghasilkan nilai kolom turunan (derived) secara otomatis
* Mencegah transaksi yang invalid / salah
* Memaksakan otorisasi yang kompleks
* Memaksakan referential integrity pada semua node dalam database terdistribusi
* Memaksakan aturan bisnis yang kompleks
* Menyediakan event logging yang transparan
* Menyediakan audit pada basis data
* Menjaga replikasi tabel agar tetap sinkron
* Mengumpulkan statistik terhadap akses terhadap suatu tabel

1. **Pembuatan Trigger**

Penamaan trigger mengikuti aturan berikut:

**(BEFORE | AFTER)\_tableName\_(INSERT| UPDATE | DELETE)**

Contoh nama trigger: before\_mahasiswa\_insert Atau

**tablename\_(BEFORE | AFTER)\_(INSERT| UPDATE | DELETE)**

Contoh nama trigger versi dua: mahasiswa\_after\_update Sintaks pembuatan trigger pada MySQL:

**CREATE TRIGGER trigger\_name trigger\_time trigger\_event ON table\_name**

**FOR EACH ROW BEGIN**

**... END;**

A screenshot of a computer

Description automatically generated

**CREATE TABLE employeesaudit (**

**id INT AUTO\_INCREMENT PRIMARY KEY,**

**employeeNumber INT NOT NULL, firstName VARCHAR(50) NOT NULL,**

**lastName VARCHAR(50),**

**changeDate DATETIME DEFAULT NULL, action VARCHAR(50) DEFAULT NULL**

**);**

Keterangan:

* **Trigger name**: sesuai penamaan pada penjelasan sebelumnya
* **Trigger time**: before atau after
* **Trigger event**: event insert, atau update, atau deletePerhatikan tabel berikut:

Buat trigger pada tabel employeesaudit:

**DELIMITER $$**

**CREATE TRIGGER before\_employee\_update BEFORE UPDATE ON employees**

**FOR EACH ROW BEGIN**

**INSERT INTO employeesaudit SET action = 'update',**

**employeeNumber = OLD.employeeNumber, firstName = OLD.firstName, lastName = OLD.lastName, changeDate = NOW();**

**END$$ DELIMITER ;**

**PERHATIAN:**

Ada beberapa perbedaan sintaks dari INSERT/UPDATE/DELETE/SELECT ketika digunakan di dalam sebuah prosedur/trigger. Di sini terlihat bahwa ada SET dalam INSERT. Fungsi SET di sini serupa dengan VALUES, yang bedanya di SET diikuti dengan nama-nama atribut yang akan diisikan nilainya.

Pada trigger update dan delete bisa digunakan keyword **OLD**, sedangkan pada trigger insert digunakan keyword **NEW** untuk mengakses data yang terkena trigger.

Trigger pada MySQL dapat dilihat dengan perintah:

**SHOW TRIGGERS;**

Coba tambahkan satu employees baru dengan data:

|  |  |
| --- | --- |
| employeeNumber | 1056 |
| lastName | Fili[ana](mailto:filiana@informatics.ukdw.ac.id) |
| firstName | Agatha |
| extension | 123 |
| Email | [filiana@informatics.ukdw.ac.id](mailto:filiana@informatics.ukdw.ac.id) |
| officeCode | 100 |
| reportsTo | 1 |
| jobTitle | MGR |

Kemudian untuk menguji trigger, silakan coba melakukan update employee :

**UPDATE employees SET**

**firstName = 'Agata' WHERE**

**employeeNumber = 1056;**

Perubahan akan tersimpan di tabel employeesaudit.

**Materi Praktikum (45 menit)**

* 1. **Pengertian SP**

SP adalah kumpulan instruksi-instruksi / program untuk suatu hal tertentu yang spesifik dan ditulis dalam bahasa SQL, disimpan dalam basis data serta dapat dipanggil secara spesifik menggunakan suatu nama tertentu dan bahkan dipanggil dari suatu bahasa pemrograman. SP dapat dipanggil oleh triggers, SP lain, dan aplikasi lain. SP juga dapat bersifat rekursif. Beberapa DBMS besar mendukung SP, seperti MySQL dan Oracle. Gambar 1 merupakan gambar ilustrasi SP.

* 1. **Keuntungan dan Kelemahan SP**

Keuntungan SP antara lain:

* + 1. SP sangat mengurangi overhead pemrosesan SQL pada basis data
    2. SP memiliki kecepatan yang tinggi dalam pemrosesan SQL karena disimpan di dalam basis data dan langsung dijalankan di server.
    3. SP bersifat reusable seperti layaknya fungsi / prosedur dalam bahasa pemrograman.
    4. SP bersifat Write Once and Run Everywhere. Sekali ditulis, SP akan disimpan di server dan dipanggil dengan mudah serta memiliki sintaks standar.
    5. SP digunakan dalam aspek keamanan database. Pada kasus tertentu SP menjembatani akses data terhadap suatu tabel di mana user tidak bisa melakukan query terhadap tabel. Permission select terhadap suatu tabel kadang tidak diperbolehkan karena masalah data sensitif. Untuk menjembatani hal tersebut, cukup dibuatkan priviledge EXECUTE tanpa perlu melakukan SELECT secara eksplisit terhadap user tersebut, sehingga user tersebut hanya perlu menjalankan stored procedure untuk hal-hal tertentu saja.

Kelemahan SP antara lain:

1. SP tidak memiliki version control, sehingga modifikasi terhadap suatu stored procedure tidak ada historinya, dan tidak bisa dikembalikan ke kondisi semula.
2. SP hanya bisa di export / backup dengan priviledge administrator database, tidak bisa pengguna biasa.
3. Jika menggunakan banyak SP, memory yang digunakan cukup besar.
4. Sulit untuk mendebug SP karena jarang ada editor yang memiliki fitur tersebut.
   1. **Implementasi SP**

Untuk membuat SP pada MySQL, bentuk umumnya adalah sebagai berikut:

**CREATE**

**[OR REPLACE]**

**PROCEDURE [IF NOT EXISTS] [nama procedure] ([IN|OUT|INOUT] [param1] [tipedata], )**

**{ CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA } BEGIN**

**[perintah–perintah]**

**END**

Jika klausa IF NOT EXISTS digunakan, maka procedure hanya akan dibuat jika sebuah procedure dengan nama sama belum pernah ada. Jika sudah ada, maka akan muncul peringatan.

Penjelasan parameter pada SP adalah:

* **IN** – parameter IN akan menerima nilai masukan ke dalam procedure. Procedure bisa memodifikasi nilainya, namun modifikasi ini tidak terlihat dari sisi pemanggil saat procedure selesai melakukan tugasnya.
* **OUT** – parameter OUT akan melempar nilai dari procedure kembali ke sisi pemanggil. Nilai awal di dalam procedure secara default adalah NULL. Karena sifatnya, maka nilai parameter ini otomatis terlihat dari sisi pemanggil saat procedure selesai melakukan tugasnya.
* **INOUT** – gabungan dari kedua jenis di atas. Pemanggil melemparkan suatu nilai, bisa diproses oleh procedure, kemudian hasil modifikasi nilai ini akan terlihat dari sisi pemanggil.

Pernyataan CONTAINS SQL, NO SQL, READS SQL DATA, atau MODIFIES SQL DATA

hanyalah klausa untuk memberitahu server mengenai apa yang dilakukan oleh procedure. Jika kontennya tidak sesuai dengan pernyataan yang ditulis, MariaDB tidak akan mempermasalahkan. Jika tidak ditulis, defaultnya CONTAINS SQL.

Contoh SP sederhana untuk menghitung rata-rata gaji adalah:

**DELIMITER $$**

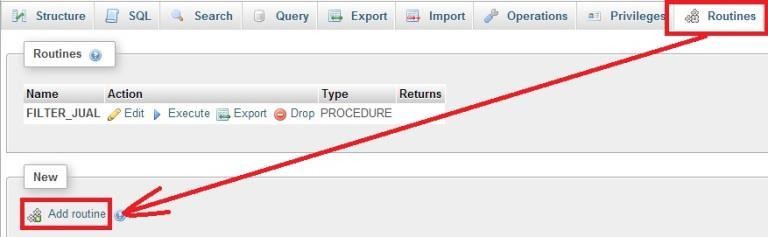
**CREATE PROCEDURE `avg\_sal`(OUT avg\_sal decimal) BEGIN**

**SELECT AVG(sal) INTO avg\_sal FROM salary;**

**END$$ DELIMITER ;**

Cara pemanggilan SP pada SQL query:

Pembuatan SP dalam phpMyAdmin juga dapat melalui menu Routine.



**CALL avg\_sal(@out); SELECT @out;**

* 1. **Cara menampilkan SP pada Database MySQL**

Cara untuk menampilkan stored procedure yang ada di database MySQL adalah menggunakan sintaks:

**SHOW PROCEDURE STATUS [LIKE 'pattern' | WHERE expr];**

Contoh:

**SHOW PROCEDURE STATUS;**

**SHOW PROCEDURE STATUS WHERE db = 'dbmahasiswa'; SHOW PROCEDURE STATUS WHERE name LIKE '%product%';**

Sedangkan untuk menampilkan source code dari SP adalah:

**SHOW CREATE PROCEDURE stored\_procedure\_name; SHOW CREATE FUNCTION function\_name;**

Function akan kita bahas pada pertemuan berikutnya.

Selain menjalankan dengan cara CALL, SP pada phpMyAdmin juga dapat dijalankan melalui interface EXECUTE.

A close-up of a box

Description automatically generated

* 1. **Penggunaan variable pada SP**
  2. **Variabel Lokal Procedure**

Seperti pada pemrograman pada umumnya, kita bisa menggunakan variabel lokal pada function dan procedure. Pendeklarasian variabel memiliki sintaks sebagai berikut:

**DECLARE var\_name [, var\_name] ... type [DEFAULT value]**

Contoh:

**DECLARE total\_sale INT DECLARE x, y INT DEFAULT 0**

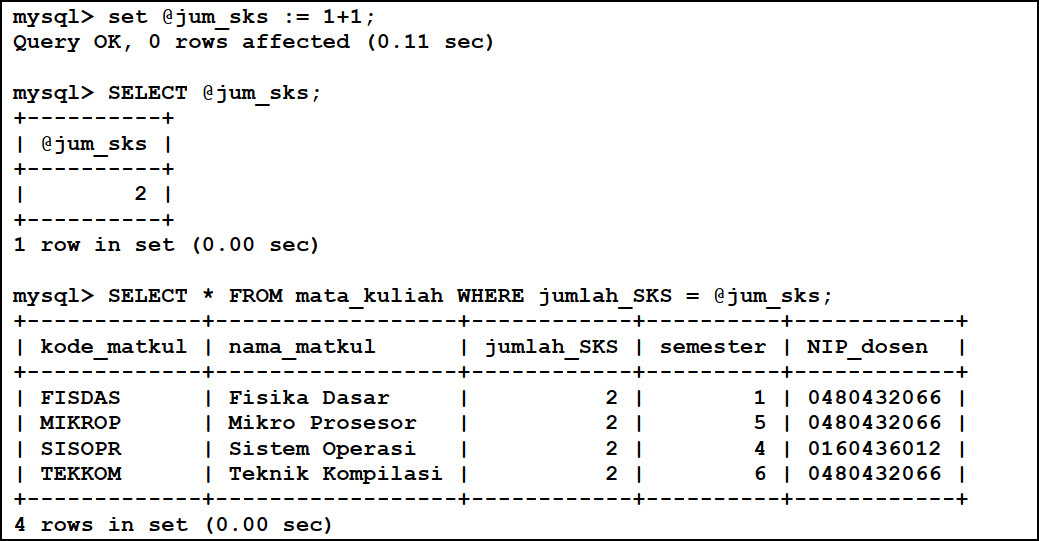
Pemberian nilai ke sebuah variabel dilakukan dengan menggunakan statement SET. Hasil dari query juga dapat dimasukkan ke dalam variabel menggunakan SELECT ... INTO Berikut adalah beberapa contoh pemberian nilai ke variabel.

**SET total\_sale = 50; DECLARE numPekerja INT;**

**SELECT COUNT(\*) INTO numPekerja FROM pekerja;**

* 1. **Variabel Lokal User**

Variabel lokal pada level user dibuat dengan menggunakan simbol @ diikuti dengan nama variabelnya. Pendeklarasian tersebut bisa dilakukan di console dan di baris- baris SQL. Cara membaca data dari variabel user tersebut dengan menggunakan perintah **SELECT @nama\_variabel**.

Contoh:

* 1. **Penggunaan Flow Control IF pada SP**

SP pada prinsipnya seperti program/fungsi pada pemrograman biasa. Sehingga pada MySQL kita juga dapat membuat SP dimana di dalamnya mengandung unsur percabangan atau IF. Dengan menggunakan IF maka kita bisa mengeksekusi kumpulan statement SQL berbasis pada kondisi tertentu atau nilai dari suatu ekspresi. Untuk membentuk ekspresi pada MySQL, kita dapat menggabungkan literal, variabel, operator, dan bahkan fungsi. Suatu ekspresi dapat mengembalikan hasil TRUE, FALSE, atau NULL.

Bentuk umum perintah IF pada MySQL:

**IF expression THEN**

**statements; END IF;**

Jika ada dua kondisi yang akan dibandingkan:

**IF expression THEN**

**statements;**

**ELSE**

**else-statements; END IF;**

Jika ada lebih dari dua kondisi lain yang akan dibandingkan:

**IF expression THEN**

**statements;**

**ELSEIF elseif-expression THEN elseif-statements;**

**...**

**ELSE**

**else-statements; END IF;**

* 1. **Penggunaan Flow Control CASE pada SP**

Selain menggunakan model IF, SP MySQL juga mendukung CASE, mirip dengan SWITCH CASE pada bahasa pemrograman biasa. Terdapat dua model CASE: Simple CASE dan Searched CASE.

Bentuk umum perintah Simple CASE adalah:

**CASE case\_expression**

**WHEN value1 THEN commands WHEN value2 THEN commands**

**...**

**ELSE commands END CASE;**

Bentuk umum perintah Searched CASE adalah:

**CASE**

**WHEN expression\_value1 THEN commands WHEN expression\_value2 THEN commands**

**...**

**ELSE commands END CASE;**

* 1. **Penggunaan Flow Control LOOP pada SP**

Perulangan pada SP ada dua macam:

– Perulangan WHILE Bentuk umum:

**WHILE expression DO**

**statements END WHILE;**

Contoh:

**DECLARE income INT;**

**SET income = 0;**

**WHILE income <= 3000 DO**

**SET income = income + starting\_value; END WHILE;**

– Perulangan REPEAT Bentuk umum:

**REPEAT**

**statements; UNTIL expression END REPEAT;**

Contoh:

**DECLARE income INT; SET income = 0; REPEAT**

**SET income = income + starting\_value; UNTIL income >= 4000**

**END REPEAT;**

### Function pada MySQL

Seperti pada bahasa pemrograman, fungsi adalah program yang **mengembalikan nilai**. Pada MySQL juga dapat dibuat program yang dimasukkan ke dalam fungsi, mirip seperti pada stored procedure, bisa memiliki parameter input, namun di dalam fungsi harus mengembalikan nilai dalam suatu tipe data tertentu. Proses pemanggilannya dapat dilakukan di dalam sintaks statemen lain secara langsung, misalnya pada statemen SELECT Contoh:

**DELIMITER $$**

**CREATE FUNCTION countCustCountry (countryIn VARCHAR(15)) RETURNS INT BEGIN**

**DECLARE result INT;**

**SELECT COUNT(\*) INTO result FROM customers WHERE country = countryIn; RETURN result;**

**END$$**

**DELIMITER;**

Sedangkan pemanggilannya adalah:

**SELECT countCustCountry('USA');**

Contoh lain untuk menghitung komisi 5% berdasarkan batas kredit customer:

**DELIMITER $$**

**CREATE FUNCTION addCommission(credLim DECIMAL(10,2)) RETURNS DECIMAL(10,2)**

**BEGIN**

**DECLARE comm FLOAT DEFAULT 0.05;**

**RETURN credLim \* comm; END$$**

**DELIMITER;**

Berikut ini cara menampilkan nama lengkap kontak customer dengan komisi credit limitnya dengan format rupiah (asumsi $1 = Rp 14.500,00):

**SELECT CONCAT(contactFirstName,' ',contactLastName) AS contact, CONCAT('Rp ',FORMAT(addCommission(creditLimit)\*14500,2,'de\_DE')) AS commission FROM customers;**

### MySQL Cursor

Untuk bisa mengambil data hasil dari stored procedure**,** kita dapat menggunakan **cursor**. Cursor memungkinkan kita untuk melakukan iterasi ke dalam hasil query SELECT per baris. Sifat cursor pada MySQL adalah:

* + **Read only**: tidak bisa digunakan untuk menulis / update / delete, hanya bisa baca data
  + **Non scrollable**: tidak bisa mengambil data secara mundur, hanya bisa iterasi maju saja dan tidak bisa melompat ke baris tertentu.
  + **Asensitive**: menunjuk ke data aktual, bukan temporary copy/insensitive.

Cursor bisa digunakan di stored procedure, stored function maupun pada trigger. Tahapan penggunaan kursor:

1. Buat cursor:

**DECLARE cursor\_name CURSOR FOR SELECT\_statement;**

Sebuah kursor selalu berhubungan dengan pernyataan SELECT.

1. Kemudian kursor harus di open dengan statement:

**OPEN cursor\_name;**

1. Ambil data ke cursor:

**FETCH cursor\_name INTO variables list;**

1. Tutup kursor

**CLOSE cursor\_name;**

Ketika membuat cursor harus mendeklarasikan kondisi saat cursor tidak ketemu data sama sekali, misalnya seperti berikut:

**DECLARE CONTINUE HANDLER FOR NOT FOUND SET finished = 1;**

A white rectangular object with black text

Description automatically generatedIlustrasi kursor adalah sebagai berikut:

Contoh: Buatlah procedure berisi cursor untuk menampilkan semua email dari employees. JAWAB :

**CREATE PROCEDURE buildEmailList (INOUT emailList VARCHAR(4000))**

**BEGIN**

**DECLARE vFinished INTEGER DEFAULT 0;**

**DECLARE vEmail varchar(100) DEFAULT "";**

**-- declare cursor for employee email DECLARE emailCursor CURSOR FOR**

**SELECT email FROM employees;**

**-- declare NOT FOUND handler DECLARE CONTINUE HANDLER**

**FOR NOT FOUND SET vFinished = 1;**

**OPEN emailCursor; getEmail: LOOP**

**FETCH emailCursor INTO vEmail; IF vFinished = 1 THEN**

**LEAVE getEmail; END IF;**

**-- build email list**

**SET emailList = CONCAT(vEmail,"; ",emailList);**

**END LOOP getEmail; CLOSE emailCursor;**

**END;**

Cara penggunaan :

**SET @emailList = "";**

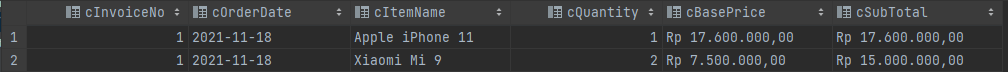
**CALL buildEmailList(@emailList); SELECT @emailList;**

1. dengan penyajian PERSIS sebagai berikut:

|  |  |  |
| --- | --- | --- |
| **officeCode** | **city** | **employees** |
| 1 | San  Fransisco | Thompson; Jennings; Bow; Firrelli; Patterson;  Murphy; |
| 2 | Boston | Patterson; Firrelli; |
| 3 | NYC | Vanauf; Tseng; |
| 4 | Paris | Gerard; Castillo; Hernandez; Bondur; Bondur; |
| dst.. |  |  |

1. A screen shot of a computer

   Description automatically generatedDengan menggunakan cursor, buatlah tabel tampungan **tempOrderList** yang berisi barang yang dibeli sebagai berikut:



Parameter input di prosedur **getDetailInvoice** hanyalah **invoice\_no**. Prosedur berisi SELECT untuk mengisi cursor, kemudian FETCH isi cursor untuk meng-INSERT ke tabel **tempOrderList**. Berikut DDL untuk tabel **tempOrderList**:

CREATE TABLE tempOrderList ( cInvoiceNo INT, cOrderDate DATE, cItemName VARCHAR(50), cQuantity INT, cBasePrice VARCHAR(50), cSubTotal VARCHAR(50)

)

Skema retail standar (yang sebelumnya dipakai di materi JOIN):

