Operatori e funzioni – I

Operatori e funzioni vengono utilizzati in diversi punti delle istruzioni SQL. Ad esempio per determinare i valori da selezionare, per determinare le condizioni in una WHERE, o nelle clausole ORDER BY, GROUP BY, HAVING. Vedremo ora i principali, tenendo a mente un paio di regole generali:

- Un'espressione che contiene un valore *NULL* restituisce sempre *NULL* come risultato, salvo poche eccezioni.
- Fra il nome di una funzione e le parentesi che contengono i parametri non devono rimanere spazi. È possibile modificare questo comportamento con l'opzione *–sql-mode=IGNORE_SPACE*, ma in questo caso i nomi di funzione diventano parole riservate.

Quelle che stiamo per elencare, come detto, sono solo alcune delle funzioni disponibili: per una lista e una descrizione completa vi rimandiamo al manuale ufficiale.

Per cominciare, i classici **operatori aritmetici**:

- "+" (addizione)
- "-" (sottrazione)
- "*****" (moltiplicazione)
- "/" (divisione)
- "**%**" (modulo – resto della divisione)

Ricordate che una divisione per zero dà come risultato (e modulo) NULL.

Passiamo agli **operatori di confronto**: il risultato di un'espressione di confronto può essere "**1**" (vero), "**0**" (falso), o NULL.

Gli operatori sono:

- "=" (uguale)
- "<>" o "**!**=" (diverso)
- "<" (minore)
- ">" (maggiore)
- "<=" (minore o uguale)
- ">=" (maggiore o uguale)
- "<=>" (uguale *null-safe*)

Con quest'ultimo operando otteniamo il valore 1 se entrambi i valori sono null, e 0 se uno solo dei due lo è.

Abbiamo quindi i classici **operatori logici**:

- NOT
- AND
- OR
- XOR (OR esclusivo)

Come sinonimo di NOT possiamo usare "**!**"; "**&&**" al posto di AND, e "||" al posto di OR.

Abbiamo poi **IS NULL** e **IS NOT NULL** per verificare se un valore è (o non è) NULL; **BETWEEN** per test su valori compresi fra due estremi (inclusi); **IN** per verificare l'appartenenza di un valore ad una lista di valori dati.

Vediamo un esempio:

```
SELECT a,b,c,d,e,f,g FROM t1
WHERE a=b AND a<=c
AND (d=5 OR d=8)
AND e BETWEEN 7 and 9
AND f IN('a','b','c')
AND g IS NOT NULL;
```

Questa query estrae le righe di t1 in cui *a* è uguale a *b* ed è minore o uguale a *c*, *d* è uguale a 5 o a 8, *e* è compreso fra 7 e 9, *f* ha uno dei valori espressi fra parentesi e *g* non ha un valore NULL.

Come avete visto abbiamo usato le parentesi per indicare che l'espressione "d=5 or d=8″ deve essere valutata prima delle altre. Il consiglio è di utilizzarle sempre in questi casi, invece di imparare a memoria il lungo elenco delle precedenze.

Molto importante è l'operatore **LIKE**, utilizzabile per trovare corrispondenze parziali sulle stringhe. Possiamo usare due caratteri jolly nella stringa da trovare: "**%**" che rappresenta "qualsiasi numero di caratteri o nessun carattere", e "**_**" che invece corrisponde esattamente ad un carattere.

Quindi, ad esempio:

```
SELECT * FROM tab1 WHERE colonna LIKE 'pao%';
SELECT * FROM tab1 WHERE colonna LIKE '_oro';
SELECT * FROM tab1 WHERE colonna LIKE '%oro';
```

La prima query troverà 'paolo', 'paola' e 'paolino'; la seconda troverà 'moro' ma non 'tesoro' perchè si aspetta esattamente un carattere in testa alla stringa; l'ultima invece troverà 'moro', 'tesoro' e anche 'oro'.

La funzione **CAST** converte un dato in un tipo diverso da quello originale (ad esempio un numero in stringa). Il tipo di dato ottenuto può essere: DATE, DATETIME, TIME, DECIMAL, SIGNED INTEGER, UNSIGNED INTEGER, BINARY, CHAR. Con questi ultimi due può essere specificata anche la lunghezza richiesta.

```
SELECT CAST(espressione AS DATE)
-> converte in formato data
SELECT CAST(espressione AS BINARY(5))
-> converte in una stringa binaria di 5 byte
```

È anche possibile utilizzare l'operatore **BINARY** come sintassi veloce per considerare la stringa seguente come binaria; questo fa sì che eventuali confronti vengano fatti byte per byte e non carattere per carattere, rendendo sempre significativa la differenza fra maiuscole e minuscole, così come gli spazi in fondo alle stringhe.

SELECT 'a' = 'A'
SELECT BINARY 'a' = 'A'

Il primo di questi due test sarà vero, il secondo sarà falso. "BINARY 'a'" equivale a "CAST('a' AS BINARY)".

Esiste poi una funzione **CONVERT (… USING …)** utile per convertire una stringa fra diversi character set (vedere lez.10). Ad esempio:

SELECT CONVERT('abc' USING utf8)

Restituisce la stringa 'abc' nel set di caratteri utf8. Attenzione: esiste un'altra sintassi della funzione CONVERT, che però è un sinonimo di CAST: ad esempio CONVERT(*espressione*,DATE) corrisponde al primo esempio visto in precedenza su CAST.

Funzioni per il controllo di flusso

Sono utili quando vogliamo eseguire dei test sui valori contenuti in una tabella e decidere cosa estrarre in base al risultato. Le indichiamo con la loro sintassi:

– CASE *valore* WHEN [*valore1*] THEN *risultato1* [WHEN [*valore2*] THEN *risultato2* ….] [ELSE *risultatoN*] END
– CASE WHEN [*condizione1*] THEN *risultato1* [WHEN [*condizione2*] THEN *risultato2* …] [ELSE *risultatoN*] END
- IF(*espressione1*,*espressione2*,*espressione3*)
– IFNULL(*espressione1*,*espressione2*)
– NULLIF(*espressione1*,*espressione2*)

Le prime due (**CASE**) sono quasi uguali: nel primo caso viene specificato un valore che sarà confrontato con quelli espressi dopo la WHEN; il primo che risulta uguale determinerà il risultato corrispondente (quello espresso con THEN).

Nel secondo caso non c'è un valore di riferimento, ma vengono valutate le varie condizioni come espressioni booleane: la prima che risulta vera determina il risultato corrispondente. In entrambi i casi, se è presente il valore ELSE finale viene usato nel caso in cui nessuna delle condizioni precedenti sia soddisfatta (in mancanza di ELSE verrebbe restituito NULL).

Con la **IF** viene valutata la prima espressione: se vera viene restituita la seconda, altrimenti la terza. **IFNULL** restituisce la prima espressione se diversa da NULL, altrimenti la seconda. **NULLIF** restituisce NULL se le due espressioni sono uguali; in caso contrario restituisce la prima.

Funzioni sulle stringhe

CONCAT e **CONCAT_WS** si utilizzano per concatenare due o più stringhe, nel secondo caso aggiungendo un separatore.

LOWER e **UPPER** consentono di trasformare una stringa, rispettivamente, in tutta minuscola o tutta maiuscola.

LEFT e **RIGHT** estraggono n caratteri a sinistra o a destra della stringa.

LENGTH e **CHAR_LENGTH** restituiscono la lunghezza di una stringa, con la differenza che la prima misura la lunghezza in byte, mentre la seconda restituisce il numero di caratteri; evidentemente i valori saranno diversi per le stringhe che contengono caratteri multi-byte.

LPAD e **RPAD** aggiungono, a sinistra (LPAD) o a destra, i caratteri necessari a portare la stringa alla lunghezza specificata (eventualmente accorciandola se più lunga).

LTRIM e **RTRIM** eliminano gli spazi a sinistra (LTRIM) o a destra.

SUBSTRING restituisce una parte della stringa, a partire dal carattere specificato fino alla fine della stringa o, se indicato, per un certo numero di caratteri.

FIND_IN_SET, infine, è una funzione particolarmente utile con i campi di tipo SET, per verificare se un dato valore è attivo.

Alcuni esempi, seguiti dai rispettivi risultati:

SELECT CONCAT_WS(‘;’,'Primo’,'Secondo’,'Terzo’);
-> Primo;Secondo;Terzo
SELECT LOWER(‘Primo’);
-> primo
SELECT RIGHT(‘Primo’,2);
-> mo
SELECT LENGTH(‘Primo’);
-> 5
SELECT LPAD(‘Primo’,7,’_');
-> __Primo
SELECT LTRIM(‘ Primo’);
-> Primo
SELECT SUBSTRING(‘Primo’,2);
-> rimo
SELECT SUBSTRING(‘Primo’,2,3);
-> rim
SELECT * FROM *tabella* WHERE FIND_IN_SET(‘Primo’,*col1*) > 0
-> (restituisce le righe in cui il valore ‘Primo’ è attivo nella colonna col1, ipotizzando che si tratti di una colonna di tipo SET)

Funzioni matematiche

ABS restituisce il valore assoluto (non segnato) di un numero; **POWER** effettua l'elevamento a potenza (richiede base ed esponente); **RAND** genera un valore casuale compreso tra 0 e 1.

Abbiamo poi le funzioni di arrotondamento, che sono:

- **FLOOR** (arrotonda all'intero inferiore)
- **CEILING** (all'intero superiore)
- **ROUND** (arrotonda all'intero superiore da .5 in su, altrimenti all'inferiore)
- **TRUNCATE** che tronca il numero (non arrotonda) alla quantità specificata di decimali

Ecco gli esempi:

```
SELECT ABS(-7.9);
-> 7.9
SELECT POWER(3,4);
-> 81
SELECT RAND();
-> 0.51551992494196 (valore casuale)
SELECT CEILING(6.15);
-> 7
SELECT ROUND(5.5);
-> 6
SELECT TRUNCATE(6.15,1);
-> 6.1
```

Se abbiamo bisogno di generare un intero compreso fra x e y, possiamo usare questa formula: "FLOOR(x + RAND() * (y – x + 1))". Ad esempio, per avere un numero compreso fra 1 e 100:

```
SELECT FLOOR(1 + RAND() * 100);
```

Operatori e funzioni – II

Funzioni su date e ore

Teniamo presente che le date vengono fornite da MySQL nel formato 'AAAA-MM-GG', mentre le ore sono nel formato 'HH:MM:SS'. Il timestamp è formato dalle due stringhe separate da uno spazio.

CURRENT_DATE e **CURRENT_TIME** restituiscono, rispettivamente, la data e l'ora attuali; **CURRENT_TIMESTAMP** e **NOW**, che sono sinonimi, restituiscono data e ora. **DATE** estrae la parte data da un timestamp; **TIME** fa lo stesso con la parte ora.

Ci sono poi le funzioni che estraggono da una data, da un orario o da un timestamp le singole informazioni: **DAY** restituisce il giorno del mese, **DAYOFWEEK** il giorno della settimana (1 è la domenica e 7 il sabato), **DAYOFYEAR** il giorno dell'anno (da 1 a 366), **MONTH** il mese, **YEAR** l'anno; **HOUR**, **MINUTE** e **SECOND**, rispettivamente, ore, minuti e secondi.

In questi esempi ipotizziamo che in questo momento siano le 17.35.21 del 28 dicembre 2005:

```
SELECT NOW();
-> 2005-12-28 17:35:21
SELECT CURRENT_TIME();
-> 17:35:21
SELECT DATE(NOW()); (equivale a CURRENT_DATE())
-> 2005-12-28
SELECT MONTH(NOW());
-> 12
SELECT DAYOFWEEK(NOW());
-> 4
SELECT MINUTE(NOW());
-> 35
```

È possibile anche lavorare con il timestamp Unix (cioè il numero di secondi trascorsi dal 1 gennaio 1970): la funzione **FROM_UNIXTIME** accetta in input tale timestamp e restituisce un timestamp MySQL, mentre **UNIX_TIMESTAMP** lavora al contrario, cioè restituisce il timestamp Unix a partire da quello MySQL o da una semplice data.

```
SELECT UNIX_TIMESTAMP('2005-12-28 17:35:21');
-> 1135787721
SELECT FROM_UNIXTIME(1135787721);
-> 2005-12-28 17:35:21
```

Ci sono infine due funzioni (**DATE_FORMAT** e **TIME_FORMAT**) che permettono di formattare i valori di date e ore. Esse richiedono in input una data o un timestamp (DATE_FORMAT) o un orario (TIME_FORMAT), nonchè una stringa contenente i simboli per la formattazione voluta.

Elenchiamo alcuni di questi simboli seguiti da qualche esempio:

- Anno: '**%Y**' (4 cifre), '**%y**' (2 cifre)
- Mese: '**%m**' (2 cifre), '**%c**' (1 o 2 cifre), '**%M**' (nome intero in inglese), '**%b**' (nome abbreviato)
- Giorno del mese: '**%d**' (due cifre), '**%e**' (1 o 2 cifre)
- Giorno della settimana: '**%W**' (nome intero in inglese), '**%a**' (nome abbreviato), **'%w'** (numerico con 0=domenica e 6=sabato)
- Ore: '**%H**' (2 cifre), '**%k**' (1 o 2 cifre), '**%h**' (2 cifre da 01 a 12), '**%l**' (1 o 2 cifre da 1 a 12)
- Minuti: '**%i**' (2 cifre)
- Secondi: '**%s**' (2 cifre)

SELECT DATE_FORMAT('2006-01-09 08:59:15′,'%d/%m/%y %H.%i');
-> 09/01/06 08.59
SELECT DATE_FORMAT('2006-01-09 08:59:15′,'%W %e %b %Y – %k:%i');
-> Monday 9 Jan 2006 – 8:59

Funzioni per il criptaggio e la decriptazione dei dati

Le funzioni che effettuano **criptaggio irreversibile** sono: **PASSWORD**, utilizzata da MySQL per le password degli utenti; **OLD_PASSWORD**, da utilizzare per ottenere le password codificate con l'algoritmo utilizzato fino a MySQL 4.0 (vedere lez.7); **MD5** e **SHA1**, che sfruttano noti algoritmi di hashing il cui risultato sono stringhe di, rispettiamente, 32 e 40 caratteri. é consigliabile usare uno di questi ultimi due per memorizzare le password delle vostre applicazioni.

Abbiamo poi le funzioni **AES_ENCRYPT** e **AES_DECRYPT**, che sfruttano un algoritmo reversibile per codificare e decodificare una stringa attraverso una password. Se volete memorizzare le stringhe codificate è consigliabile utilizzare una colonna di tipo BLOB, in quanto si tratta di stringhe binarie. MySQL consiglia questa coppia di funzioni come il metodo più sicuro disponibile per il criptaggio di stringhe.

SELECT PASSWORD('albero');
-> *7D69DA9FAAC2BF8B8181DCABB4E38B9DA1916389
SELECT MD5('albero');
-> 338a96591f778e7af4cce7b601d785d2
SELECT SHA1('albero');
-> 0389de785fafb5418e0221c49bb9545a9a82a2b1
INSERT INTO *tabella* SET *campoBlob* = AES_ENCRYPT('albero','pw');
SELECT AES_DECRYPT(*campoBlob*,'pw') FROM *tabella*
-> albero

Nell'ultimo esempio salviamo su un campo di tipo BLOB la stringa 'albero' criptata con la password 'pw'; successivamente possiamo recuperarla utilizzando la funzione di decodifica con la stessa password.

Funzioni di aggregazione

Le funzioni di aggregazione non lavorano, come quelle viste finora, su un solo dato, ma su insiemi di dati. Ciò significa che ci restituiranno un unico valore come "sintesi" di *n* valori. Normalmente i valori NULL vengono ignorati.

Ecco le principali di queste funzioni:

- **COUNT** conta i valori trovati
- **AVG** ne calcola la media
- **MIN** e **MAX** trovano il minore e il maggiore
- **SUM** effettua la somma

Esempi:

SELECT COUNT(*) FROM *tab1*
-> conta il numero di righe della tabella
SELECT COUNT(*colonna*) FROM *tab1*
-> conta i valori non NULL di colonna
SELECT COUNT(DISTINCT *colonna*) FROM *tab1*
-> conta i diversi valori (duplicati valgono per 1)
SELECT AVG(*colonna*) FROM *tab1*
-> calcola la media dei valori
SELECT AVG(DISTINCT *colonna*) FROM *tab1*
-> calcola la media conteggiando una sola volta i duplicati
SELECT MAX(*colonna*) FROM *tab1*
-> trova il valore massimo
SELECT SUM(*colonna*) FROM *tab1*
-> calcola la somma

Le funzioni di aggregazione vengono usate spesso in combinazione con la clausola GROUP BY (vedere lez.14)).
In questi casi le aggregazioni non vengono svolte su tutte le righe estratte dalla WHERE, ma singolarmente su ogni gruppo di righe formato dalla GROUP BY. In questo caso è possibile aggiungere alla query la clausola **WITH ROLLUP** che produce subtotali ad ogni cambiamento di valori dei campi raggruppati.

SELECT anno, paese, prodotto, SUM(profitti)
FROM vendite GROUP BY anno, paese, prodotto WITH ROLLUP

Questa query genererà totali suddivisi per prodotto, per paese e per anno, ma fornirà anche i totali per paese, per anno e infine un totale generale. Nella tabella risultato le colonne su cui vengono calcolati i totali generali sono mostrate con valore NULL (ad esempio nella riga finale con il totale generale saranno a NULL i valori di anno, paese e prodotto).

Lo standard SQL vuole che in una SELECT che contiene funzioni di aggregazione tutte le colonne su cui tali funzioni non lavorano siano comprese nella GROUP BY. Ad esempio questa query non sarebbe valida, perchè manca la GROUP BY sul campo nome:

```
SELECT ordini.idCliente, clienti.nome, MAX(pagamenti)
FROM ordini,clienti
WHERE ordini.idCliente = clienti.id
GROUP BY ordini.idCliente
```

La query ha tuttavia un senso logico, perchè il valore di nome, dipendendo dalla join, sarà sempre lo stesso in ogni gruppo di righe con lo stesso 'idCliente': di conseguenza è superfluo inserirlo nella GROUP BY. MySQL quindi ci consente di usare questa sintassi: bisogna però fare attenzione a non omettere la GROUP BY su un campo che non ha un valore unico, in quanto ne otterremmo un risultato non prevedibile.

Ricerche full-text

Le ricerche full-text sono ricerche basate sul "linguaggio naturale", effettuabile su campi di tipo CHAR, VARCHAR e TEXT. Si tratta del tipo di interrogazione che normalmente facciamo quando utilizziamo un motore di ricerca. Per poterla effettuare è necessario che le colonne interessate facciano parte di un indice FULLTEXT (vedi lez.12) e che la tabella sia di tipo MyISAM. Le ricerche vengono effettuate con la funzione **MATCH**:

```
SELECT * FROM articoli
WHERE MATCH(titolo,testo) AGAINST('database')
```

Questa query effettua la ricerca del termine 'database' sulle colonne titolo e testo della tabella, ed estrae quelle che hanno una qualche rilevanza. Le due colonne devono far parte dello stesso indice FULLTEXT.

```
SELECT id, titolo, MATCH(titolo,testo) AGAINST('database') as rilevanza
FROM articoli
WHERE MATCH(titolo,testo) AGAINST('database')
```

In questo caso, oltre all'id e al titolo ci verrà fornito, nella colonna 'rilevanza', anche il risultato della funzione MATCH, che è un numero compreso fra 0 e 1.

Segnaliamo anche la funzione **FOUND_ROWS**, da utilizzare per ottenere il numero di righe che sarebbero state trovate da una SELECT in cui abbiamo usato la LIMIT, se tale clausola non fosse stata presente:

```
SELECT SQL_CALC_FOUND_ROWS * FROM tabella LIMIT 10;
SELECT FOUND_ROWS();
```

Nella prima istruzione selezioniamo il contenuto di una tabella limitandolo alle prime 10 righe; tuttavia l'uso della clausola **SQL_CALC_FOUND_ROWS** indica a MySQL che vogliamo sapere quante sarebbero le righe estratte senza la LIMIT; otteniamo poi tale valore con la seconda istruzione, che **deve** essere immediatamente successiva.

UNION

Concludiamo con la sintassi della **UNION**, che non è una funzione ma una clausola che ci consente di unire due query in una unica tabella risultato. La condizione per poter fare questo è che le due query selezionino lo stesso numero di colonne e che le colonne in posizione corrispondente siano dello stesso tipo.

Vediamo l'esempio più semplice:

SELECT *cola*, *colb*, *colc* FROM *tab1*
UNION
SELECT *col1*, *col2*, *col3* FROM *tab2*

Da notare che, in una UNION, viene applicata di default la clausola DISTINCT, cioè le righe duplicate non vengono restituite. Se volete il comportamento opposto dovete specificare UNION ALL.