Guida MySQL

Una guida per usare uno dei più importanti sistemi di gestione di data base sulla scena open-source, imparando a sfruttarne anche le caratteristiche più avanzate

Primi passi con MySQL

Creare il database

Usare il database

Quali sono e a cosa servono gli Storage Engine disponibili su MySQL:...

2. 13. INSERT: inserimento dei dati
   Conoscere le direttivi INSERT, REPLACE e LOAD DATA INFILE per...
3. 14. UPDATE e DELETE: modifica e cancellazione dei dati
   Imparare ad aggiornare ed eliminare i dati dalle tabelle di un...
4. 15. SELECT: interrogare MySQL
   La principale istruzione per interrogare un database MySQL è...
5. 16. JOIN, creare query relazionali
   Imparare l'utilizzo corretto delle JOIN, uno strumento di primaria...
6. 17. Operatori e funzioni
   Conoscere ed utilizzare i principali operatori e le funzioni...
7. 18. Gestire date e orari
   Le principali funzioni per gestire, combinare e formattare...
8. 19. Ricerche full-text
   Utilizzare le funzionalità di ricerca full-text sui campi testo dei...
9. 20. Funzioni di aggregazione e UNION
   Conoscere ed imparare ad utilizzare le funzioni di aggregazione e la...
10. 21. Le subquery
    Una panoramica che mostra come creare ed eseguire query annidate,...

Funzioni avanzate

1. 22. Transazioni e lock
   Garantire l'esecuzione di operazioni in serie su MySQL, nonchè...
2. 23. Le viste (views)
   Utilizzare e conoscere il costrutto delle viste (o views) su MySQL,...
3. 24. Stored Procedures e Stored Functions
   Utilizzare Stored Procedures e Stored Functions per implementare (e...
4. 25. Trigger
   Utilizzare i Trigger su MySQL per associare a particolari eventi di...
5. 26. Cifratura e decifratura dei dati
   Conoscere le potenzialità di MySQL in ambito di sicurezza dei dati:...

Amministrare il database

1. 27. GRANT e REVOKE, gestire i permessi
   Gestire i permessi di accesso ad un database MySQL tramite le...
2. 28. Gestire gli utenti
   Conoscere i meccanismi alla base della gestione degli utenti che si...
3. 29. Dump, backup e recovery
   Impare a gestire ed effettuare le operazioni di backup, dump e...
4. 30. Ottimizzare il database
   Una serie di utili consigli e accorgimenti per migliorare le...
5. 31. Configurazioni Cluster
   Imparare a configurare un'installazione distribuita di MySQL, tramite...

API per l'accesso al database

1. 32. Connectors e API: una panoramica
   Un'introduzione alle API (da C a Java, passando per C++ e PHP) ed ai...
2. 33. PHP: API per l'accesso al DB
   Un confronto tra le principali API per l'accesso, tramite PHP, ad un...
3. 34. Java: API per l'accesso al DB
   Interagire con un database MySQL tramite il linguaggio Java,...
4. 35. C#: API per l'accesso al DB
   Utilizzare Connector/NET e ADO.NET per l'interazione con un database...

Appendice

1. 36. Esempio: struttura del DB di un albergo
   Un esempio che mostra come strutturare un database MySQL per la...
2. 37. Esempio: interrogare il DB di un albergo
   Un esempio che mostra come interagire con un database MySQL per la...
3. 38. MariaDB
   Le principali differenze tra MySQL e MariaDB, uno dei più diffusi ed...

Edizione precedente (2006)

1. 105. Il client mysql
   Lo strumento base per connettersi al server

Introduzione agli RDBMS e a MySQL

Uno dei più grandi contributi che i sistemi informatici offrono al genere umano è la memorizzazione di dati in maniera persistente. Quotidianamente, immense quantità di informazioni vengono affidate a tecnologie che ne garantiscono la conservazione duratura ed un recupero efficiente che ne permetta l'analisi. Da anni, questo ruolo viene interpretato molto bene da un prodotto software completo, efficiente ed affidabile: **MySQL**. Nel seguito chiariremo sin da subito che cos'è esattamente, a cosa serve e come utilizzarlo, illustrandone anche le principali caratteristiche e potenzialità.

Database e DBMS

I concetti centrali in tema di memorizzazione di dati sono due: **database** e DMBS.

Il primo indica un sistema di file finalizzato a memorizzare informazioni a supporto di un qualsivoglia software. La struttura interna di un database deve rispettare una certa architettura di immagazzinamento dei dati per poterne permettere il corretto salvataggio, il rispetto dei tipi impiegati e soprattutto agevolarne il recupero, un'operazione generalmente molto onerosa.

Un DBMS è un servizio software, realizzato in genere come server in esecuzione continua, che gestisce uno o più database. I programmi che dovranno interagire quindi con una base di dati non potranno farlo direttamente, ma dovranno dialogare con il DBMS. Esso sarà l'unico ad accedere fisicamente alle informazioni.

Quanto detto implica che il DBMS è il componente che si occupa di tutte le politiche di accesso, gestione, sicurezza ed ottimizzazione dei database.

Database relazionali e RDBMS

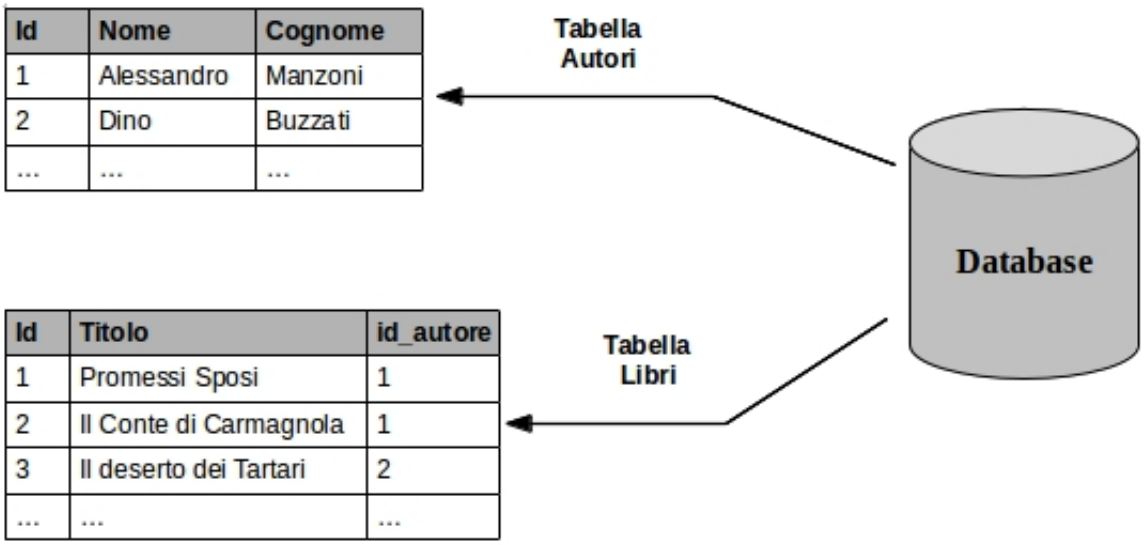
I DBMS esistenti non sono tutti della stessa tipologia. Al giorno d'oggi, ad esempio, si parla molto di **DBMS NoSQL**, nati per venire incontro alle esigenze dei più recenti servizi Web. Eppure un filone molto nutrito di DBMS, cui si deve il funzionamento della maggior parte dei prodotti informatici esistenti oggi, è quello dei cosiddetti **RDBMS** (Relational DBMS), ispirati dalla Teoria Relazionale. Questa nacque nel 1970 ad opera

del britannico Edgar f. Codd. Nonostante i 40 anni abbondanti trascorsi, i suoi principi si dimostrano tuttora attuali.

Un database relazionale è costituito da tabelle, ognuna delle quali è composta da righe identificate da un codice univoco denominato chiave. Le tabelle che compongono il database non sono del tutto indipendenti tra loro ma relazionate da legami logici.

La figura seguente mostra un esempio di database finalizzato a custodire i dati di una biblioteca.

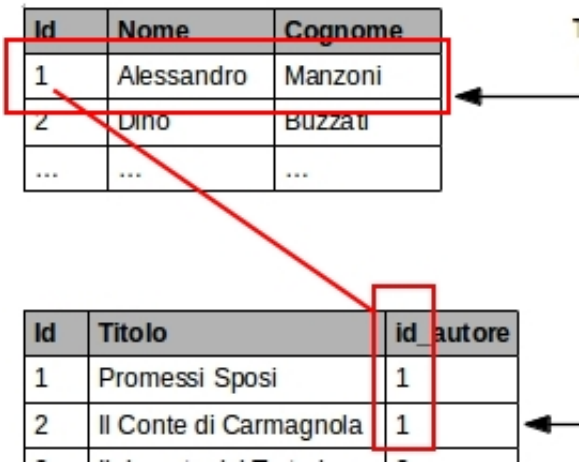
Figura 1. Tabelle in un database relazionale



Il database ha due tabelle: *Autori* e *Libri*. Ognuna di esse ha una colonna di nome *id* contenente l'identificativo univoco, la **chiave primaria**, che permette di riconoscere una riga senza possibilità di confusione. I valori nel campo *id_autore* della tabella *Libri* permettono di associare un autore alle sue opere. Ad esempio, "I Promessi Sposi" e "Il Conte di Carmagnola" hanno nel campo *id_autore* il valore *1*, cioè la chiave primaria che nell'altra tabella permette di riconoscere il loro autore, Alessandro Manzoni.

Questo legame creato tramite chiavi prende il nome di **relazione**.

Figura 2. Relazione tra tabelle in dettaglio (click per ingrandire)



Esistono vari tipi di relazioni. Quello appena descritto è un esempio di **relazione uno-a-molti**, in quanto ad un autore possono corrispondere più libri. Nel nostro esempio si è assunto che un libro possa avere un solo autore, ma sfruttando relazioni di tipo differente si potranno rappresentare situazioni più vicine alla realtà.

Nel seguito della guida, si approfondiranno tali argomenti fino a vedere una progettazione completa di un database.

Protagonista importante di questa guida, e strumento fondamentale nell'interazione con i DBMS relazionali, è il **linguaggio SQL** (Structured Query Language). Si tratta di un formalismo che permette di indicare al DBMS quali operazioni svolgere sui database che gestisce. Tramite SQL si può attivare qualsiasi tipo di operazione, sia sui dati che sulla struttura interna del database, sebbene le principali (e più frequenti) operazioni ricadono in una delle seguenti quattro tipologie: inserimento, lettura, modifica e cancellazione di dati, tipicamente indicate con l'acronimo **CRUD** (Create-Read-Update-Delete). Il seguito della guida mostrerà un'ampia panoramica dei comandi SQL, nonché diversi esempi di utilizzo.

MySQL

MySQL è un RDBMS **open source** e **libero**, e rappresenta una delle tecnologie più note e diffuse nel mondo dell'IT. MySQL nacque nel 1996 per opera dell'azienda svedese Tcx, basato su un DBMS relazionale preesistente, chiamato mSQL. Il progetto venne distribuito in modalità open source per favorirne la crescita.

Dal 1996 ad oggi, MySQL si è affermato molto velocemente prestando le sue capacità a moltissimi software e siti Internet. I motivi di tale successo risiedono nella sua capacità di mantenere fede agli impegni presi sin dall'inizio:

- alta efficienza nonostante le moli di dati affidate;
- integrazione di tutte le funzionalità che offrono i migliori DBMS: *indici*, *trigger* e *stored procedure* ne sono un esempio, e saranno approfonditi nel corso della

guida;
- altissima capacità di integrazione con i principali linguaggi di programmazione, ambienti di sviluppo e suite di programmi da ufficio.

Scaricare e Installare MySQL

Per ottenere MySQL, il nostro punto di riferimento sarà la **pagina di download del sito ufficiale**. Da qui possiamo subito notare che esistono tre versioni principali:

- **MySQL Enterprise Edition**: una versione molto completa di MySQL, rivolta all'azienda che vuole gestire tutti i propri flussi informativi tramite questo prodotto. La sua installazione mette a disposizione il server oltre a varie componenti per il *Partitioning*, la *Replication*, lo sviluppo di applicazioni client nonché strumenti rivolti alla sicurezza (MySQL Enterprise Security, MySQL Enterprise Backup) e alla gestione della disponibilità di dati (MySQL Enterprise HA e MySQL Enterprise Scalability). Questa versione è soggetta a licenza commerciale;
- **MySQL Cluster GCE**: è un'altra versione commerciale che avvicina il DBMS alle esigenze più moderne, permettendo di attivare velocemente funzionalità di *clustering*. Questa versione contiene MySQL Cluster, MySQL Cluster Manager più tutto ciò che è incluso anche nella versione Enterprise;
- **MySQL Community Edition**: a differenza delle precedenti, è sottoposta a licenza GPL, ed è quindi gratuitamente scaricabile. Mette comunque a disposizione tutte le componenti principali: il server ma anche MySQL Cluster e MySQL Fabric (per le funzionalità di *High Availability* e *Sharding*).

Nel prosieguo della guida toccheremo tutti gli aspetti più significativi di MySQL ed utilizzeremo come versione di riferimento la Community Edition.

MySQL trova in Linux e nei sistemi Unix-like il suo habitat naturale, ma è comunque disponibile per ogni piattaforma. Ne vedremo quindi prima l'installazione su sistemi Windows e successivamente su distribuzioni Linux.

Installazione su Windows

La versione di MySQL utilizzabile su Windows mette a disposizione tutti gli strumenti fondamentali citati in precedenza (server, Workbench, eccetera) ma anche ulteriori potenzialità che permettono al DBMS di dialogare con importanti tecnologie di sviluppo degli ambienti Microsoft, tra cui la suite da ufficio Office, l'IDE Visual Studio, il

protocollo ODBC ed il framework .NET.

Pertanto, un'installazione di MySQL Community Edition per Windows si compone di un certo numero di strumenti:

- **MySQL Server**: il vero e proprio servizio per la gestione dei database;
- **MySQL for Excel**: plugin per Excel per la gestione di tabelle MySQL tramite fogli di calcolo;
- **MySQL Notifier**: una piccola utility che permette di monitorare i servizi MySQL dal proprio desktop Windows;
- **MySQL Workbench**: strumento unico di accesso ai dati e di monitoraggio;
- **MySQL For Visual Studio**: un plugin di Visual Studio che permette ai programmatori .NET di lavorare ai propri database senza abbandonare l'IDE di sviluppo;
- **Connectors**: driver per l'integrazione di database MySQL nei propri programmi.

Tutti questi strumenti sono resi disponibili tramite un unico installer, scaricabile a**questo link**. Qui potremo scaricare due file: *mysql-installer-web-community* e *mysql-installer-community*.

Il primo dei due ha un peso molto ridotto (attualmente 1,6 MB) al contrario del secondo, 262 MB. Le capacità di installazione dei due sono le stesse. La differenza consiste nel fatto che il primo scaricherà i software necessari da Internet, mentre il secondo sarà un pacchetto di installazione completo. Per questo motivo, la seconda opzione è preferibile solo nel caso in cui non fosse disponibile una connessione ad Internet.

L'installazione viene svolta mediante una procedura guidata. Verranno proposte varie configurazioni:

- **Developer Default**: saranno installati tutti gli strumenti utili allo sviluppo di applicazioni;
- **Server-only**: si installerà solo il server;
- **Client-only**: solo gli strumenti client, ad esempio i Connectors ed il Workbench;
- **Full**: installa tutta la distribuzione;
- **Custom**: permette di selezionare ogni singolo strumento da installare.

Una fase molto importante è il controllo dei prerequisiti. Verranno installate solo quelle componenti che saranno in grado di rispettarli tutti. Ad esempio, per installare i plugin per Excel e Visual Studio dovranno già essere installati i software ai quali si riferiscono, mentre in altri casi, come il Workbench, si dovrà prima installare Microsoft Visual C++ 2013 Runtime.

Una volta colmate le eventuali lacune, MySQL potrà procedere all'installazione.

Installazione su Linux

Esistono molte versione di MySQL per Linux, adatte alle varie distribuzioni del sistema operativo. Può essere scaricato come archivio *.tar.gz* ma anche sotto forma di pacchetti compatibili con *yum* o *apt*.

Su Ubuntu, si potrà avere a disposizione il server MySQL ed il client tramite il seguente comando:

```
aptitude install mysql-server mysql-client
```

Durante l'installazione, verrà richiesta la password per l'utente *root*. Dopo averla inserita e confermata sarà bene custodirla con cura.

Per avere informazioni specifiche sull'installazione su altre distribuzioni, si rimanda all'apposita pagina della **documentazione ufficiale**. Esistono comunque anche versioni per Linux generiche, scaricabili sempre dalla pagina di download come archivio compresso *.tar.gz*.

All'interno è presente un file di testo contenente le istruzioni di installazione che, in sintesi, consistono nei seguenti passi:

- decomprimere l'archivio scaricato, e collocarlo in una posizione a scelta del sistema;
- creare un utente ed un gruppo tramite i quali il DBMS agirà: tipicamente verranno denominati entrambi *mysql*;
- è necessario che nel sistema sia installata la libreria *libaio*. Qualora essa non fosse disponibile, si dovrà soddisfare questo prerequisito. L'operazione non risulta generalmente molto complessa, poiché tale libreria è disponibile tra i repository delle principali distribuzioni Linux;
- a questo punto, non rimane che eseguire lo script *mysql_install_db* nella cartella *scripts*.

Al termine della procedura, il DBMS sarà installato e potrà essere avviato con il comando:

```
bin/mysqld_safe --user=mysql &
```

A questo punto sarà già possibile fare accesso al server tramite il client con:

Ciò permetterà di avere accesso come utente *root*, amministratore del DBMS, senza l'uso di password. Questa potrà essere assegnata tramite i seguenti comandi SQL, il

cui senso diverrà più chiaro con le prossime lezioni:

```
mysql> UPDATE mysql.user SET Password = PASSWORD('nuova password') WHERE User =
'root';

mysql> FLUSH PRIVILEGES;
```

mysqld e i tool di base

L'installazione di MySQL porta con sè un gran numero di programmi che riguardano tutte le attività principali di amministrazione del DBMS. Si tratta, in genere, di strumenti a riga di comando sebbene, come vedremo in seguito, esistono diversi tool visuali altrettanto completi.

In questa lezione presenteremo una panoramica complessiva degli strumenti, per poi proseguire con approfondimenti specifici.

Panoramica degli strumenti

Come molti altri DBMS, MySQL viene eseguito come servizio o, in altre parole, come *daemon*. Un servizio o demone è un programma in esecuzione continua nel sistema operativo, il cui compito è quello di rimanere in attesa di richieste finalizzate alla fruizione di determinate funzionalità. Nel caso dei DBMS, tutto lo scambio di dati con il demone avrà come scopo la gestione dei database.

Il demone alla base del DBMS prende il nome di **mysqld**. Insieme ad esso vengono forniti altri programmi per l'avvio del server: *mysqld_safe*, pensato per un avvio più sicuro del server, e *mysqld_multi*, che permette l'avvio di più server installati nel sistema.

Tra gli altri strumenti messi a disposizione, dedicati a varie attività dello sviluppatore e dell'amministratore del DBMS, vi sono:

- **mysql**: il client ufficiale per interagire con i database. Verrà trattato in seguito;
- **mysqladmin**, per lo svolgimento di ogni genere di operazione di configurazione del server;
- **mysqlcheck**, che si occupa della manutenzione delle tabelle;
- **mysqldump**, indispensabile per il backup. Anch'esso verrà trattato nel corso della guida;
- **mysqlimport**, che permette di importare tabelle nei database;
- **mysqlshow**, che fornisce informazioni su database, tabelle, indici e molto altro.

Il server: avvio, arresto e controllo

Nelle installazioni più comuni, *mysqld* viene avviato in automatico all'avvio del sistema. Possiamo verificare che esso sia in esecuzione con *mysqladmin*, che possiede un'apposita funzionalità di ping:

```
mysqladmin -u root -p ping
```

Con questa istruzione, verrà richiesto da riga di comando, su qualunque sistema operativo, di effettuare un ping sul servizio. L'opzione *-u* specifica che la richiesta è fatta come utente *root*, amministratore del DBMS, mentre *-p* impone la richiesta di password per l'utente. Fornendo i corretti dati di autenticazione, se il DBMS è attivo, sarà stampato il messaggio "*mysql is alive*"; se invece non è attivo, verrà sollevato un errore di connessione.

Un altro modo per controllare lo stato del demone è verificare la sua presenza tra i processi in esecuzione. Su Windows lo si potrà fare con il Task Manager, mentre su Linux sarà solitamente sufficiente verificare l'output prodotto dal comando:

```
ps aux | grep mysql
```

Il risultato mostrato dovrebbe presentare più righe, di cui una contenente il percorso al programma *mysqld*.

Qualora il server non fosse in esecuzione, per avviarlo sarà necessario ricorrere ad uno degli strumenti predisposti appositamente per questo scopo, come *mysqld_safe*. In molte installazioni attuali, il DBMS viene predisposto tra i servizi di sistema. In questi casi il modo migliore per avviarlo o arrestarlo è utilizzare le apposite interfacce. Ad esempio, su Windows, si può utilizzare lo strumento *Servizi di sistema* accessibile tramite il Pannello di Controllo. Tra gli altri servizi sarà presente anche quello relativo a MySQL e sarà sufficiente verificarne lo stato e modificarlo. Anche sui sistemi Linux si possono avere opportunità simili. Ad esempio, nelle distribuzioni che usano il meccanismo *Upstart* come Ubuntu, Fedora ed altre, la consueta installazione del DBMS permette di avviare, arrestare e verificare lo stato da riga di comando, rispettivamente, con i seguenti comandi:

```
service mysql start
service mysql stop
service mysql status
```

Il client: interagire con MySQL

Per poter svolgere operazioni sui dati, il tipico client è il programma da riga di comando *mysql*.

Per prima cosa dobbiamo farci riconoscere. Le opzioni più comunemente utili in questo caso sono:

Opzione	Descrizione
`-u` o `--user`	Per introdurre il nome utente

`-p` o `--password`	Per richiedere l'accesso con password, che sarà richiesta a meno che non venga fornita come parametro. Si faccia attenzione al fatto che, usando la forma `-p`, la password deve essere concatenata all'opzione (esempio: `-ptopolino`) mentre nella forma estesa si deve usare il simbolo `=` (esempio: `--password=topolino`), che risulta preferibile in quanto più leggibile
`-h` o `--host`	Consente di specificare l'indirizzo IP o il nome della macchina su cui è in esecuzione il DBMS; il valore di default è `localhost`
`-P` o `--port`	Consente di specificare la porta TCP
`-D` o `--database`	Indica il nome del database cui si vorrà fare accesso

Un esempio di autenticazione tramite il comando `mysql` potrebbe quindi essere:

```
mysql -h mysql.mioserver.com -u dbadmin -p
```

In risposta, verrà richiesto di fornire la password. Si avrà così accesso all'istanza in esecuzione all'indirizzo *mysql.mioserver.com*, come utente *dbadmin*.

Dopo il login, ci troveremo di fronte il prompt di MySQL. I primi comandi utili da imparare sono:

Comando	Descrizione
`quit`	Per uscire dal programma *mysql*
`show databases;`	Per vedere i database accessibili tra quelli gestiti dal DBMS
`use`	Seguito dal nome del database, specifica su quale DB si vuole lavorare

Nel prompt di MySQL, come vedremo, potranno essere eseguiti anche comandi e query SQL. In questo caso non va dimenticato che il comando dovrà essere concluso da un punto e virgola.

MySQL, i file di configurazione

MySQL può essere configurato nei modi più disparati. I medesimi parametri possono essere passati sia al demone che agli altri tool in varie modalità: tramite file di configurazione (*Option file*), da riga di comando o tramite variabili d'ambiente.

Queste tre casistiche non hanno tutte la stessa priorità. Eventuali opzioni passate da riga di comando prevalgono su tutte le altre in quanto dotate di un carattere di maggiore contingenza. Il grosso della configurazione viene specificata tramite file di configurazione del servizio, le cui impostazioni superano eventuali valori assegnati tramite variabili d'ambiente.

Una lista completa delle opzioni disponibili per *mysqld* è disponibile su **questa pagina**. Di seguito vedremo comunque come utilizzare le principali funzionalità di configurazione.

File di configurazione

Il consueto nome dell'*Option file* è *my.cnf*, ma ogni programma che compone MySQL può accedere a più file di configurazione. Per conoscerne l'elenco, si può invocare da riga di comando il nome del tool, seguito dall'opzione *–help*.

Ad esempio:

fornisce diverse informazioni, tra cui anche l'elenco degli *Option file* cui attinge attualmente il client *mysql*, ed i parametri che ne ha ricevuto.

Nel caso di *mysqld*, la configurazione crea un profilo di funzionamento per tutto il server. Per conoscerlo è consigliabile usare il comando seguente:

oppure:

Saranno elencati diversi file che, a seconda del sistema operativo, potranno riferirsi a cartelle dedicate a configurazioni di sistema (*C:|Windows* nel sistema di Microsoft, o */etc* sui sistemi Linux) oppure a cartelle di proprietà di un singolo utente. Il valore dei parametri avrà quindi un senso globale o personalizzato, a seconda della posizione del file che li contiene.

Opzioni del server

Qui di seguito, vengono elencate alcune delle opzioni più importanti che determinano il profilo di un server MySQL, tentando di classificarle in base alla loro finalità.

Esistono, innanzitutto, configurazioni che riguardano il modo in cui il DBMS è contestualizzato nel sistema che lo ospita:

Opzione	Descrizione
`basedir`	Percorso alla cartella principale di MySQL. Importante perché altri percorsi indicati possono riferirsi a questa come punto di partenza
`datadir`	La cartella in cui sono realmente contenuti i database. Al suo interno si troveranno varie sottocartelle, ognuna delle quali corrispondente ad un database
`user`	L'utente con cui il DBMS agirà all'interno del sistema operativo
`tmpdir`	La cartella che conterrà i file temporanei. Questa solitamente coincide con la cartella finalizzata al medesimo scopo nel sistema operativo

Le seguenti opzioni determinano, invece, l'**accessibilità in rete del DBMS**:

Opzione	Descrizione
`bind-address`	Indica l'indirizzo di rete sul quale il server può essere contattato. Di default, troveremo l'indirizzo locale, *127.0.0.1*, il che significherà che MySQL potrà essere contattato solo dalla stessa macchina. Si potranno specificare ovviamente altri indirizzi, con alcune eccezioni come *, che permetterà di accettare connessioni su qualunque interfaccia dotata di indirizzo IPv4 o IPv6, oppure *0.0.0.0* per permettere connessioni su tutte le interfacce dotate di indirizzo IPv4
`port`	La porta TCP su cui il server rimarrà in ascolto. Il valore di default è *3306*

Un'altra categoria di opzioni riguarda la **gestione dei file di log**. Ad esempio, *log_error* indica dove è collocato nel sistema il file a cui vengono accodati i messaggi riguardanti il DBMS. Elenca eventi di arresto e avvio, oltre ad errori ed informazioni sullo stato del servizio. I log svolgono un ruolo molto importante in un server come MySQL, e ciò è testimoniato anche dal gran numero di opzioni che li riguardano. Esiste anche un formato di log binario che tiene traccia delle modifiche fatte ai dati (utile per la verifica del corretto funzionamento di funzionalità infrastrutturali come la *replication* dei dati tra diverse istanze di MySQL).

Nel corso della guida, trattando argomenti avanzati e più specifici del DBMS, vedremo altre opzioni di configurazione, dalle quali dipenderà il corretto funzionamento di MySQL.

Client visuali per MySQL

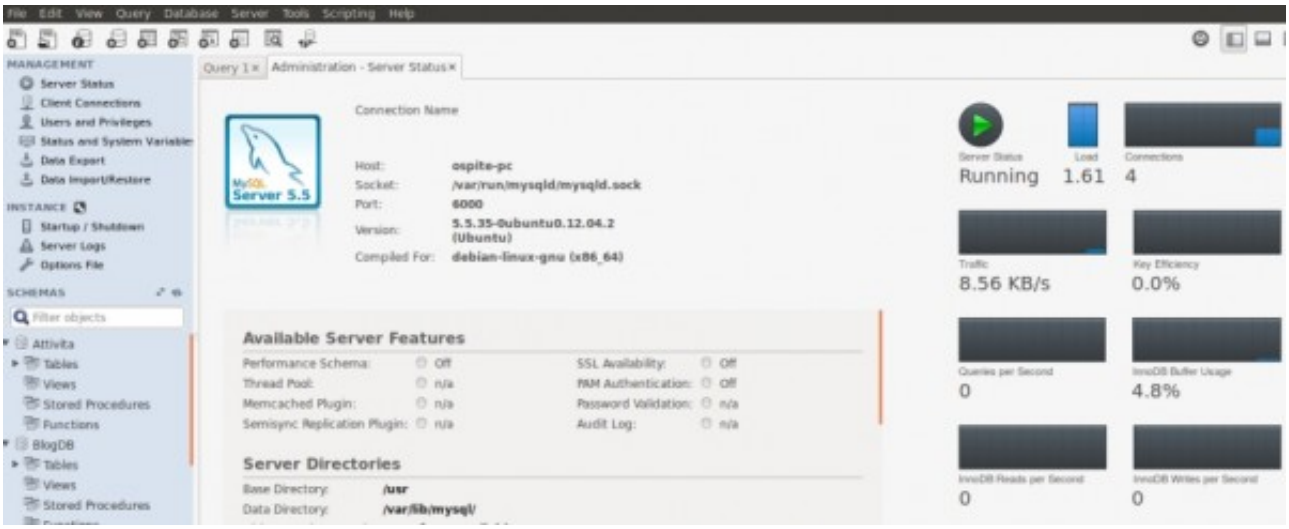
Nelle lezioni precedenti si è parlato delle componenti che costituiscono un'installazione MySQL e delle varie possibilità di configurazione. Un elemento che si nota è la completezza degli strumenti forniti (server, client testuale, strumenti di amministrazione ed ottimizzazione) che, d'altro canto, sono tutti utilizzabili solo da riga di comando. Nella realtà quotidiana, anche per amministratori e sviluppatori professionisti, risultano molto comodi alcuni strumenti visuali, dotati di interfaccia utente. In questa lezione vedremo i più noti, forniti ufficialmente da MySQL o da produttori di terze parti. Se ne valuteranno le caratteristiche e:

- funzionalità offerte;
- tipologia di installazione: desktop, portable, applicativo web;
- disponibilità sulle varie piattaforme;
- eventuale costo.

MySQL Workbench

Si è già accennato che MySQL Workbench accompagna ogni installazione del DBMS. Si tratta di uno strumento molto completo che cerca di offrire supporto a tutte le principali funzionalità del server, dalla versione 5.0 in poi.

Figura 3. L'interfaccia di MySQL Workbench (click per ingrandire)



Le tematiche su cui si concentra il Workbench sono cinque:

- **Sviluppo SQL**: in questo ruolo sostituisce pienamente il client *mysql*. Permette di connettersi ad un database, fornendo tutti i parametri necessari (credenziali di accesso, coordinate TCP/IP ed altro) e di inoltrare comandi SQL per interrogazioni e modifiche di dati;
- **Data modeling**: questo termine si riferisce alla progettazione visuale di un

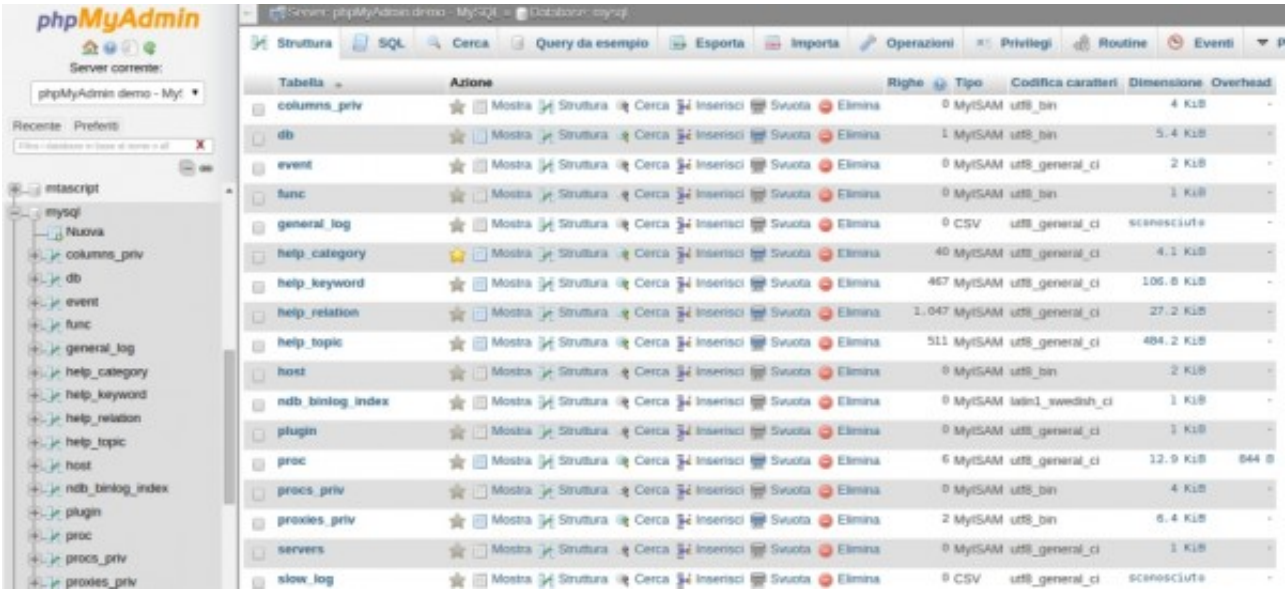
database. I DBMS relazionali come MySQL si basano sulla creazione di tabelle, suddivise in campi, e collegate tra loro da relazioni. La struttura di un database può facilmente diventare complessa, ed utilizzare solo strumenti testuali potrebbe non essere sufficientemente agevole;
- **Amministrazione del Server**: è possibile amministrare il server tramite MySQL Workbench, passando dalla gestione degli utenti all'esecuzione di backup e recovery. Il tool fornisce una modalità per gestire ogni attività che permetta di rendere più efficiente, sicuro e controllato il lavoro del DBMS;
- **Migrazione dei dati**: non sempre i dati che popolano un database vengono generati da zero. Spesso sono importati da versioni precedenti di MySQL o provengono da DBMS di altre piattaforme. La migrazione dei dati in questi casi deve essere realizzata con cura per non perdere il patrimonio informativo, pertanto il Workbench offre alcune funzionalità che consentono tale attività;
- **Supporto alle funzionalità Enterprise**: MySQL Workbench mira ad essere un tool completo, in grado di andare oltre le finalità più comuni. Per questo motivo, supporta strumenti dedicati alla sfera Enterprise, come MySQL Enterprise Backup e MySQL Audit.

Come molti altri tool, il Workbench esiste in Community Edition e Commercial Edition. Il primo è disponibile gratuitamente nonostante la ricchezza di potenzialità offerte, mentre il secondo fornisce un supporto più ampio alla sfera di attività Enterprise.

PhpMyAdmin

PhpMyAdmin è uno strumento *web-based* molto diffuso ed in continua evoluzione da 16 anni. È un progetto open source realizzato in PHP, con un'interfaccia disponibile in moltissime lingue, incluso l'italiano. La sua natura di applicazione web richiede, per l'installazione, la disponibilità di un server web con l'interprete PHP configurato ed integrato.

Figura 4. L'interfaccia di PhpMyAdmin (click per ingrandire)



Il contesto ideale per il suo utilizzo è una piattaforma comunemente denominata **LAMP** (**L**inux, **A**pache, **M**ySQL, **P**HP). È molto semplice da utilizzare in un'installazione web del DBMS e viene spesso messo a disposizione dai fornitori di hosting.

I suoi vantaggi principali sono, oltre alla stabilità ed efficienza ormai collaudate, la ricchezza di funzioni che rispecchiano la maggior parte delle caratteristiche di MySQL.

Può essere utilizzato per visualizzare e modificare i dati, per scrivere query in linguaggio SQL e, con le interfacce di supporto che offre, risulta adatto anche agli utenti meno esperti. Altre caratteristiche molto utili consistono nella possibilità di esportare i dati in vari formati, sia per realizzare report che backup di sicurezza, e di importarli da varie fonti.

Installazione e cenni sulla configurazione

Una volta scaricato PhpMyAdmin dal **sito ufficiale**, lo si può installare in una directory del web server ed impostare i dati del file di configurazione (*config.inc.php*), scegliendo fra i possibili modi di autenticazione degli utenti.

È consigliabile scegliere fra l'autenticazione http e quella basata sui cookie, in quanto l'autenticazione di tipo *config* comporta la memorizzazione della password dell'utente in chiaro nel file di configurazione; in questo caso, inoltre, la connessione al server avverrà sempre con lo stesso nome utente.

La configurazione di un server MySQL su *config.inc.php* avviene compilando l'apposito blocco di codice con i dati essenziali per la connessione: l'indirizzo host su cui gira il server, l'eventuale porta diversa da quella di default (*3306*), l'estensione di PHP da utilizzare (*mysql* o *mysqli*), il tipo di autenticazione scelta. Nel caso in cui quest'ultima sia di tipo *config* dovrete anche scrivere qui il nome utente e la password col quale intendete collegarvi; nel caso invece delle autenticazioni http o via cookie è richiesta

l'indicazione di un nome utente e password che abbia i permessi di *SELECT* sul database *mysql* (quello dei permessi; per motivi di sicurezza sarà bene che tale utente non abbia altri privilegi). Tuttavia PhpMyAdmin funzionerà anche senza quest'ultima configurazione.

Una possibilità interessante è quella di configurare più di un server sullo stesso file: ciò consentirà agli utilizzatori di PhpMyAdmin di selezionare, attraverso una casella a discesa nella home page, il server al quale intendono collegarsi.

Altri client desktop

Di client MySQL ne esistono molti, e citarli tutti sarebbe impossibile. Ne verranno nominati solo alcuni che eccellono per le loro caratteristiche.

Probabilmente uno dei più leggeri è **HeidiSQL**. Si tratta di un programma molto snello, rapido, con funzionalità che accelerano molto il lavoro dell'amministratore e dello sviluppatore. Una volta avviato, il suo *Session Manager* richiede di aprire un connessione preconfigurata o di crearne una nuova. Eseguita l'operazione, si avrà accesso all'elenco delle tabelle, sarà possibile leggerne i dati e modificarli velocemente come se si trattasse di un foglio di calcolo Excel. Esiste il supporto a funzionalità di backup e restore, importazione ed esportazione di dati, e sviluppo ed esecuzione di stored procedure.

HeidiSQL è gratuito, open source e disponibile anche in versione portable. L'unica pecca è che si tratta di un prodotto disponibile solo per Windows, sebbene sia possibile utilizzarlo su Linux e Mac tramite appositi emulatori come Wine.

Altro prodotto molto professionale è **Navicat for MySQL**, realizzato da un'azienda che offre soluzioni client anche per Oracle, PostgreSQL e tutti gli altri maggiori DBMS. Completo anch'esso di alcuni wizard che aiutano a leggere, modificare ed elaborare dati, è disponibile per ogni sistema operativo. Per usare i software Navicat è necessario acquistare una licenza che ne estenda l'uso oltre i 30 giorni di prova gratuita.

Creazione e gestione di database e tabelle

Iniziamo con questa lezione a **costruire un database**. Il lavoro verrà essenzialmente svolto impartendo ordini al DBMS attraverso il linguaggio SQL.

Per eseguire gli esempi presentati qui e nelle lezioni successive, si può utilizzare il client testuale *mysql*. Se ne è già parlato in precedenza, pertanto ricordiamo solo che è sufficiente, da riga di comando, invocarlo fornendo nome utente e password:

```
mysql -u root -p
```

Per i nostri scopi useremo l'account *root*, mentre la password sarà richiesta interattivamente.

I comandi SQL verranno impartiti direttamente nel prompt del client *mysql* dopo l'autenticazione o, in caso di comandi troppo lunghi come la creazione di tabelle, si potrà preparare un file ed inviarlo direttamente al client con l'operatore '<':

```
mysql -u root -p < test.sql;
```

Creare database

Il primo passo da affrontare è la creazione di un database. Dal prompt di *mysql* possiamo innanzitutto vedere quanti database abbiamo già a disposizione, tramite il comando seguente:

```
show databases;
```

Se non ne abbiamo mai creati, probabilmente verranno mostrati solo alcuni database "di servizio", che occorrono allo stesso MySQL per svolgere il suo lavoro.

Ecco come **creare un database con SQL**:

```
CREATE DATABASE nuovodb;
```

dove `nuovodb` è il nome che è stato scelto per il database da creare.

Conviene puntualizzare che anche questo comando deve terminare con un punto e virgola (`;`). Inoltre, sebbene `CREATE DATABASE` è stato scritto interamente in maiuscolo per rispettare le consuetudini e migliorare la leggibilità, MySQL non è case-sensitive per quanto riguarda il nome dei comandi. In altre parole, scrivere `CReaTe DaTAbaSE`

`nuovodb` avrebbe avuto lo stesso effetto.

La prova che il database è stato creato si può quindi ottenere tramite `SHOW DATABASES`.

Il nuovo database non diventa però automaticamente quello attivo (cioè quello su cui i comandi SQL saranno indirizzati). Per potervi iniziare a lavorare è dunque necessario selezionare il nostro database, tramite il comando `USE`:

```
USE nuovodb;
```

In alternativa a *mysql*, si può utilizzare il manager universale da riga di comando, *mysqladmin*.

Anche in questo caso dovremo fornire credenziali di accesso da riga di comando e successivamente utilizzare il comando `CREATE` seguito dal nome del nuovo database:

```
mysqladmin -u root -p CREATE nuovodatabase
```

Per verificare l'esito del comando impartito, si può di nuovo usare `SHOW DATABASES`, oppure un altro programma a disposizione del DBMS, *mysqlshow*. Ci si dovrà far riconoscere con le opzioni `-u` e `-p` come di consueto, e tanto sarà sufficiente a vedere l'elenco dei database gestiti al momento.

```
mysqlshow  -u root -p
```

Creare tabelle

Il database fornisce il contesto alla creazione del nostro progetto ma i veri nodi che costituiranno la sua rete di informazioni sono le tabelle. Di seguito impareremo a crearle, tramite un'opazione che può sembrare un'operazione semplice in molti casi, soprattutto con l'aiuto di strumenti visuali, ma che invece è un lavoro delicato e molto importante.

L'esempio proposto può essere svolto all'interno del client *mysql* dopo aver creato un database ed aver dichiarato di volerlo usare (direttiva *USE*).

```
CREATE TABLE `Persone` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `nome` VARCHAR(45) NULL,
  `cognome` VARCHAR(45) NULL,
  `dataDiNascita` DATE NULL,
  `sesso` ENUM('M','F') NULL,
  PRIMARY KEY (`id`))
```

```
ENGINE = InnoDB;
```

Il codice precedente crea una tabella denominata `Persone` con cinque campi di vario tipo. Durante la scrittura di un comando così lungo, nel prompt di *mysql* si può andare a capo; sarà poi il punto e virgola (`;`) a segnalare la fine dell'istruzione.

La sintassi mostra che `CREATE TABLE`, seguito dal nome della nuova tabella, è il comando che esegue l'operazione.

Tra le parentesi tonde vengono specificati i campi che la compongono; per ognuno di essi si indica:

- **nome** del campo: nell'esempio i nomi sono, rispettivamente, `id`, `nome`, `cognome`, `dataDiNascita`, `sesso`;
- **tipo di dato** del campo: il formato dell'informazione che vi verrà inserita. I tipi di dato saranno oggetto di approfondimento in una lezione futura della guida; al momento si consideri che quelli mostrati sono alcuni dei tipi più comuni. `INT` serve a specificare un numero intero, `VARCHAR` si utilizza per le stringhe, `DATE` per le date ed `ENUM` definisce un tipo di dato personalizzato che contempla l'assegnazione di un elemento di un limitato insieme di valori (in questo caso, si è deciso di specificare `M` ed `F` per indicare, rispettivamente, maschio e femmina);
- parametri che seguono il tipo di dato e possono specificare **vincoli** attribuiti al campo: `NULL` indica che il campo può essere lasciato vuoto; `NOT NULL`, viceversa, obbliga ad assegnare un valore; `AUTO_INCREMENT` indica che il numero del campo può essere generato in autonomia dal DBMS in ordine progressivo.

Infine si vede che è stato specificato il campo `id` come **chiave primaria**, mediante il comando `PRIMARY KEY`. Una chiave primaria, argomento approfondito nel seguito, indica un valore composto da uno o più campi che individua univocamente il record in cui è collocato.

È possibile anche creare una tabella temporanea utilizzando il comando `CREATE TEMPORARY TABLE` specificando i campi come di consueto. Questo tipo di tabelle esiste per il tempo di una sessione, il che permette a più utenti collegati, appartenenti quindi a sessioni diverse, di poter utilizzare le stesse tabelle. La loro utilità si esplica per lo più nell'immagazzinamento di dati temporanei a supporto di elaborazioni lunghe.

Altro aspetto molto utile da considerare è che se chiediamo di creare una **tabella che già esiste**, viene restituito un errore. Si può quindi ordinare a MySQL di creare la tabella solo nel caso in cui non ne esista già una omonima:

```
CREATE TABLE IF NOT EXISTS Persone
(
    ...
    ...
)
```

Modificare le tabelle

Abbiamo imparato a creare le tabelle del database. Può però capitare di dovere **cambiare la struttura della tabella**: le modifiche possono riguardare ogni aspetto, ma quelli di cui ci occuperemo in questa lezione riguardano per lo più il nome della tabella stessa o i suoi campi (numero, tipo, denominazione).

Il principale costrutto che utilizzeremo sarà **`ALTER TABLE`**. In generale viene seguito dal nome della tabella oggetto di modifica, e da un elemento che specifica il tipo di azione che verrà eseguita. Quelli che ci interesseranno maggiormente in questa lezione sono:

- `RENAME` per rinominare la tabella;
- `ADD` e `DROP`, rispettivamente, per aggiungere e rimuovere un campo;
- `CHANGE` per modificare il nome, il tipo di dato o altri parametri di un campo.

Considerato che tratteremo la modifica di tabelle, dovremo avere un database a disposizione. Immaginiamo quindi di crearne uno con il seguente script:

```
CREATE DATABASE Biblioteca;
USE Biblioteca;
CREATE TABLE `Libri` (
  `id` INT NOT NULL AUTO_INCREMENT,
  `Titolo` VARCHAR(45) NULL,
  `Autore` VARCHAR(45) NULL,
  PRIMARY KEY (`id`))
ENGINE = InnoDB;
```

Prima di continuare, specifichiamo che il risultato degli esempi che saranno mostrati nel seguito di questa lezione può essere verificato con il comando seguente:

```
SHOW TABLES;
```

Ciò è vero se la modifica riguarda il nome di una tabella. Se invece essa ha effetto su un elemento interno alla tabello, potremo utilizzare:

```
DESCRIBE Libri;
```

Cambiare nome ad una tabella

La prima casistica analizzata è il cambio del nome di una tabella, circostanza che può verificarsi anche per un semplice errore di digitazione durante la progettazione del database.

Immaginiamo di voler modificare il nome della tabella, cambiandolo da `Libri` in `Opere`. A tal fine dovremo far seguire il comando `ALTER TABLE` dall'azione `RENAME`:

```
ALTER TABLE Libri RENAME Opere;
```

La correttezza della modifica apportata potrà essere verificata, in questo caso, con `SHOW TABLES`: se tutto è andato bene, sarà infatti visualizzato il nuovo nome della tabella.

Modifica di nome e tipo di dato dei campi

Utilizzando il comando `CHANGE` in aggiunta ad `ALTER TABLE` è possibile richiedere la modifica di una colonna già esistente.

Ad esempio:

```
ALTER TABLE Libri CHANGE Titolo Titolo varchar(100);
```

La modifica richiesta comporterà la variazione della dimensione del campo. Le stringhe contenute potranno arrivare ora a 100 caratteri anziché 45.

Analizziamo meglio il comando appena impartito:

- `ALTER TABLE Libri`: specifica quale tabella sarà modificata;
- `CHANGE Titolo`: specifica che sarà modificato un elemento esistente (il campo `Titolo`);
- `Titolo varchar(100)`: sono i nuovi attributi assegnati.

Alla stessa maniera sarà possibile modificare il nome di un campo:

```
ALTER TABLE Libri CHANGE Titolo Opera varchar(100);
```

Come si vede, il comando è analogo al precedente. La differenza è rappresentata dalla nuova descrizione che viene assegnata al campo interessato.

Come presumibile, le modifiche ad un campo – qui mostrate in due esempi separati – possono essere apportate con la medesima direttiva.

Aggiunta o rimozione di campi

Il comando `ALTER TABLE` può essere utilizzato per aggiungere o rimuovere i campi di una tabella. Considerando la versione non ancora modificata della tabella `Libri`, il

comando:

```
ALTER TABLE Libri ADD NumeroPagine INT;
```

aggiungerà un nuovo campo alla tabella. È stato sufficiente indicare `ADD` come azione, e fare seguire tale direttiva dalla descrizione del nuovo campo, che in questo caso è un intero denominato `NumeroPagine`.

Analogamente, il comando `DROP` in congiunzione con `ALTER TABLE` permetterà di eliminare un campo:

```
ALTER TABLE Libri DROP NumeroPagine;
```

`DROP` dovrà semplicemente essere seguito dal nome del campo da cancellare.

Eliminare tabelle e database

Dopo avere visto come creare e modificare tabelle e database, è arrivato il momento di imparare a cancellarli.

Si tratta di operazioni irreversibili e quindi particolarmente delicate, considerato anche che, quando eseguite da riga di comando, vengono effettuate senza alcuna richiesta di conferma da parte del client testuale.

Come fatto nelle lezioni precedenti, utilizzeremo un database con una sola tabella, che possiamo creare come segue:

```
CREATE DATABASE Biblioteca;

USE Biblioteca;

CREATE TABLE `Libri` (

  `id` INT NOT NULL AUTO_INCREMENT,

  `Titolo` VARCHAR(45) NULL,

  `Autore` VARCHAR(45) NULL,

  PRIMARY KEY (`id`))

ENGINE = InnoDB;
```

L'**eliminazione di una tabella** può essere effettuata con il seguente comando:

Per verificare il risultato del precedente comando si potrà utilizzare:

Se invece vogliamo **rimuovere l'intero database**, possiamo ancora utilizzare `DROP`, ma nel seguente modo:

```
DROP DATABASE Biblioteca;
```

Anche in questo caso non verrà richiesta alcuna conferma dell'eliminazione.

Sappiamo che, come mostrato nello script di preparazione all'esempio, per poter lavorare con un database è necessario selezionarlo utilizzando il comando `USE`.

L'eliminazione di un database potrebbe riguardare proprio quello in uso, ed anche in questo caso ciò non comporterà alcuna "obiezione" da parte del DBMS. Semplicemente, dopo la cancellazione nessun database risulterà in uso.

Possiamo, infine, **verificare l'avvenuta cancellazione del database**, controllando che esso non compaia nella lista dei database mostrata con il seguente comando:

```
SHOW DATABASES;
```

Tipi di dato

Le colonne che possono essere definite in una tabella MySQL sono, ovviamente, di diversi tipi. Possiamo suddividerle in dati numerici, dati relativi a date e tempo, stringhe e dati geometrici.

Prima di tutto però dobbiamo ricordare che tutti i tipi di colonne possono contenere (se dichiarato nella loro definizione) il valore **NULL**, previsto dallo standard SQL per indicare un "non valore", cioè il fatto che una certa colonna può non avere valore su alcune righe della tabella.

Dati numerici

Vediamo quali sono i tipi di dati numerici:

BIT[(M)]
TINYINT[(M)] [UNSIGNED] [ZEROFILL]
SMALLINT[(M)] [UNSIGNED] [ZEROFILL]
MEDIUMINT[(M)] [UNSIGNED] [ZEROFILL]
INT[(M)] [UNSIGNED] [ZEROFILL]
BIGINT[(M)] [UNSIGNED] [ZEROFILL]
FLOAT[(M,D)] [UNSIGNED] [ZEROFILL]
DOUBLE[(M,D)] [UNSIGNED] [ZEROFILL]
DECIMAL[(M[,D])] [UNSIGNED] [ZEROFILL]

Le indicazioni comprese fra parentesi quadre sono opzionali. Come vedete, tutti i dati numerici escluso il BIT possono avere le opzioni UNSIGNED e ZEROFILL. Con la prima si specifica che il numero è senza segno, per cui non saranno consentiti valori negativi. Con la seconda si indica al server di memorizzare i numeri con degli zeri davanti nel caso in cui la lunghezza sia inferiore a quella massima prevista. Se usate ZEROFILL MySQL aggiungerà automaticamente UNSIGNED.

Il dato di tipo **BIT** è disponibile a partire da MySQL 5.0.3 per le tabelle MyISAM e dalla versione 5.0.5 per tabelle MEMORY, InnoDB e BDB. È un dato che contiene il numero di bit specificato con M (1 per default), che può andare da 1 a 64. Nelle versioni

precedenti era considerato sinonimo di TINYINT(1). Un valore di questo tipo può essere indicato ad es. con *b'111'*, che rappresenta in questo caso tre bit a 1 (corrispondenti al valore decimale 7).

I dati di tipo **TINYINT**, **SMALLINT**, **MEDIUMINT**, **INT** e **BIGINT** rappresentano numeri interi composti rispettivamente da 1, 2, 3, 4 e 8 bytes. Il TINYINT può contenere 256 valori, che vanno da -128 a +127 oppure da 0 a 255 nel caso di UNSIGNED. Allo stesso modo, SMALLINT può contenere 65536 valori, MEDIUMINT 16.777.216, INT oltre 4 miliardi, BIGINT circa 18 miliardi di miliardi.

In tutti i casi i valori massimi assoluti vanno dimezzati se non si usa UNSIGNED. Nel caso di BIGINT è però sconsigliato l'uso di UNSIGNED perchè può dare problemi con alcuni calcoli. L'indicazione del parametro M sugli interi non influisce sui valori memorizzabili, ma rappresenta la lunghezza minima visualizzabile per il dato. Se il valore occupa meno cifre, viene riempito a sinistra con degli spazi, o con degli zeri nel caso di ZEROFILL.

FLOAT e **DOUBLE** rappresentano i numeri in virgola mobile. *M* rappresenta il numero totale di cifre rappresentate e *D* il numero di cifre decimali.

FLOAT è a "precisione singola": i suoi limiti teorici vanno da -3.402823466E+38 a -1.175494351E-38 e da 1.175494351E-38 a 3.402823466E+38, oltre allo zero.

I valori DOUBLE sono invece a "precisione doppia": i limiti teorici sono da -1.7976931348623157E+308 a -2.2250738585072014E-308 e da 2.2250738585072014E-308 a 1.7976931348623157E+308, oltre allo zero.

Per entrambi i tipi di dato i limiti reali dipendono dall'hardware e dal sistema operativo. Se *M* e *D* non sono indicati i valori possono essere memorizzati fino ai limiti effettivi. Per questi dati l'uso di UNSIGNED disabilita i valori negativi, ma non ha effetto sui valori massimi positivi memorizzabili. La precisione dei numeri in virgola mobile è affidabile fino (circa) alla settima cifra decimale per i FLOAT e alla quindicesima per i DOUBLE. Una colonna FLOAT occupa 4 byte, una DOUBLE ne occupa 8.

I dati **DECIMAL** rappresentano infine numeri "esatti", con *M* cifre totali di cui *D* decimali. I valori di default sono 10 per *M* e 0 per *D*. I valori limite per questi dati sono gli stessi di DOUBLE. Il massimo di cifre consentite è 65 per *M* e 30 per *D*. A partire da MySQL 5.0.3 questi dati vengono compressi in forma binaria.

Esistono numerosi sinonimi per i dati numerici: **BOOL** e **BOOLEAN** equivalgono attualmente a TINYINT(1), sebbene sia prevista in futuro l'introduzione di un vero dato booleano per MySQL. **INTEGER** equivale a INT. **DOUBLE PRECISION** equivale a DOUBLE. **REAL** equivale a DOUBLE (a meno che tra le opzioni dell'SQL mode – v. lez. 4

– non sia presente *REAL_AS_FLOAT*). **FLOAT(p)** è un numero in virgola mobile la cui precisione in bit è indicata da *p*.

MySQL converte la dichiarazione in FLOAT o DOUBLE in base al valore di *p*: FLOAT da 0 a 24, DOUBLE da 25 a 53; in entrambi i casi la colonna risultante non avrà i valori *M* e *D*. Infine **DEC**, **NUMERIC** e **FIXED** sono sinonimi di DECIMAL.

Consultate la lezione 4 dove parla dell'SQL strict mode per verificare come vengono trattati da MySQL eventuali valori numerici non validi in fase di inserimento.

Date e tempo

Le colonne relative a date e tempo sono le seguenti:

DATE
DATETIME
TIMESTAMP[(M)]
TIME
YEAR[(2|4)]

Una colonna **DATE** può contenere date da '1000-01-01' (1° gennaio 1000) a '9999-12-31' (31 dicembre 9999). MySQL visualizza le date nel formato che vi abbiamo appena mostrato, ma vi consente di inserirle sotto forma di stringhe o numeri.

Una colonna **DATETIME** contiene una data e un'ora, con lo stesso range visto per DATE. La visualizzazione è nel formato 'AAAA-MM-GG HH:MM:SS', ma anche in questo caso possono essere usati formati diversi per l'inserimento.

Prima di MySQL 5.0.2 era sempre possibile inserire date o datetime a 0, oppure valorizzare a zero il giorno (o il giorno e mese) di una data. Era anche possibile indicare date non valide (ad es. '1999-04-31'). A partire da MySQL 5.0.2 questi comportamenti sono controllati da alcuni valori di SQL mode (v.lez.4):

- *ALLOW_INVALID_DATES* è necessario per consentire date non valide: in sua assenza, le date non valide in strict mode provocheranno un errore; senza strict mode verranno convertite a 0 con un warning;
- *NO_ZERO_DATE* non accetta date a 0 ('0000-00-00'): in strict mode verrà causato un errore a meno che non sia usata IGNORE; senza strict mode saranno comunque accettate con un warning;
- *NO_ZERO_IN_DATE* non accetta valori 0 per giorno e mese: in strict mode verrà generato errore, oppure inserita una data a 0 con IGNORE; senza strict mode saranno accettati con un warning

In un **TIMESTAMP** possono essere memorizzati i valori corrispondenti al timestamp

Unix, che vanno dalla mezzanotte del 1° gennaio 1970 ad un momento imprecisato dell'anno 2037.

Questo tipo di dato è utile per memorizzare automaticamente il momento dell'aggiornamento di una riga di tabella: infatti MySQL può impostare in automatico una colonna TIMESTAMP di una tabella nel momento in cui viene effettuata una INSERT o un UPDATE. La visualizzazione del timestamp avviene nello stesso formato del DATETIME; è possibile ottenerlo in formato numerico aggiungendo un *+0* alla colonna nella SELECT.

Fino a MySQL 4.0 le caratteristiche del timestamp erano diverse da quelle attuali. Innanzitutto veniva visualizzato in formato numerico, ed esisteva la possibilità di determinare il numero di cifre visualizzate indicando il valore di *M*. Da MySQL 4.1 in poi ciò non è più possibile. Inoltre la funzione di aggiornamento automatico era possibile solo per la prima colonna definita come TIMESTAMP in ogni tabella. Ora invece è possibile avere questo funzionamento anche per una colonna successiva alla prima.

Vediamo la possibile definizione di una colonna TIMESTAMP:

```
ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        ON UPDATE CURRENT_TIMESTAMP
```

Con questa dichiarazione, la colonna viene automaticamente inizializzata e successivamente aggiornata ad ogni modifica della riga con il timestamp del momento. Se omettete una delle due dichiarazioni, solo l'altra sarà attiva; ovviamente per la clausola DEFAULT potete anche fornire un valore costante.

Se omettete entrambe le dichiarazioni sulla prima colonna, questa sarà comunque automaticamente inizializzata e aggiornata. Se volete usare i valori automatici su una colonna successiva alla prima, dovete disabilitare questo comportamento per la prima colonna usando un valore esplicito di default (ad es. DEFAULT 0), e indicare le clausole viste sopra per la colonna che vi interessa. In questo caso l'omissione di entrambe non darà luogo all'inizializzazione e all'aggiornamento automatici.

Quando inserite un valore in un timestamp indicando NULL, se la colonna non supporta valori NULL verrà inserito il CURRENT_TIMESTAMP. Se usate l'SQL mode *MAXDB* tutte le colonne TIMESTAMP saranno convertite in DATETIME.

Una colonna **TIME** contiene un valore di tempo (ore, minuti e secondi) che va da '-838:59:59' a '838:59:59'. Anche qui la visualizzazione avviene nel formato indicato, ma è possibile usare formati diversi per l'inserimento.

Infine la colonna **YEAR** rappresenta, su quattro cifre, un anno compreso fra 1901 e

2155, oppure 0000. Su due cifre invece i valori vanno da 70 (1970) a 69 (2069).

I valori relativi al tempo possono essere inseriti sia come stringhe che come numeri, e MySQL vi consente di utilizzare, nel caso delle stringhe, molti caratteri diversi come separatori. L'importante però è che l'ordine dei valori sia sempre anno-mese-giorno-ore-minuti-secondi. Quando usate i separatori nelle stringhe potete anche omettere gli zeri non significativi (ad es. è consentito '2005-9-21' ma dovete usare '20050921').

Stringhe

Le colonne di tipo stringa, a partire da MySQL 4.1, possono avere un attributo CHARACTER SET che indica l'insieme di caratteri utilizzato per la colonna, e un attributo COLLATE che indica la collation relativa. Vediamo un esempio:

```
CREATE TABLE tabella
(
    c1 CHAR(20) CHARACTER SET utf8,
    c2 CHAR(20) CHARACTER SET latin1 COLLATE latin1_bin
);
```

In questa tabella abbiamo la colonna c1 definita col set di caratteri utf8 e la relativa collation di default; e la colonna c2 col set di caratteri latin1 e la relativa collation binaria. La lunghezza specificata è relativa al numero di caratteri (il numero di byte infatti può variare in base ai set di caratteri usati e al contenuto della colonna).

Vediamo i tipi di campi previsti:

[NATIONAL] CHAR(M) [BINARY | ASCII | UNICODE]
[NATIONAL] VARCHAR(M) [BINARY]
BINARY(M)
VARBINARY(M)
TINYBLOB
TINYTEXT
BLOB[(M)]
TEXT[(M)]
MEDIUMBLOB
MEDIUMTEXT
LONGBLOB
LONGTEXT
ENUM('valore1','valore2',...)
SET('valore1','valore2',...)

CHAR è una stringa di lunghezza fissa (*M*) riempita con spazi a destra al momento

della memorizzazione, che vengono eliminati in fase di lettura. La lunghezza prevista va da 0 a 255 caratteri. L'opzione NATIONAL indica che la stringa deve usare il set di caratteri di default. L'attributo BINARY indica che deve essere usata la collation binaria del set di caratteri utilizzato. ASCII assegna il character set *latin1*, UNICODE assegna *ucs2*.

CHAR BYTE equivale a CHAR BINARY. Notate che se una riga ha lunghezza variabile (cioè se almeno una colonna è definita a lunghezza variabile) qualsiasi campo CHAR di lunghezza superiore a 3 caratteri viene convertito in VARCHAR.

VARCHAR è una stringa a lunghezza variabile; le sue caratteristiche sono variate a partire da MySQL 5.0.3: in precedenza infatti la lunghezza massima era 255 e gli spazi vuoti a destra venivano eliminati in fase di memorizzazione; ora invece ciò non avviene più e la lunghezza massima dichiarabile è salita a 65535 caratteri. Gli attributi NATIONAL e BINARY hanno lo stesso significato visto in CHAR. Se definite una colonna VARCHAR con meno di 4 caratteri sarà trasformata in CHAR.

BINARY e **VARBINARY** corrispondono a CHAR e VARCHAR, ma memorizzano stringhe di byte invece che di caratteri. Non hanno quindi character set. I valori BINARY ricevono un riempimento a destra di byte 0x00 a partire da MySQL 5.0.15; in precedenza il riempimento era a spazi e veniva rimosso in fase di lettura. Nei valori VARBINARY, fino a MySQL 5.0.3 gli spazi finali venivano rimossi in lettura.

I formati di tipo BLOB e TEXT sono utilizzati rispettivamente per valori binari e di testo. La lunghezza massima è 255 caratteri per **TINYBLOB** e **TINYTEXT**, 65535 per **BLOB** e **TEXT**, 16.777.215 per **MEDIUMBLOB** e **MEDIUMTEXT**, 4 gigabyte per **LONGBLOB** e **LONGTEXT**.

Per queste ultime però bisogna tenere presenti i limiti del packet size nel protocollo client/server nonchÈ quelli della memoria. È possibile anche dichiarare una colonna BLOB o TEXT specificando una lunghezza in byte: in questo caso il server sceglierà il tipo più piccolo in grado di contenere i caratteri richiesti (ad es. con BLOB(100000) verrà creato un MEDIUMBLOB).

Se cercate di inserire un valore troppo lungo nei campi, con strict mode avrete un errore; senza strict mode il valore sarà troncato a destra e ci sarà un warning se i caratteri troncati non sono spazi.

Una colonna **ENUM** può contenere uno dei valori elencati nella definizione, oppure NULL o una stringa vuota, che viene assegnata quando si cerca di inserire un valore non valido. I valori possibili possono essere fino a 65535.

Una colonna **SET**, come la ENUM, prevede un insieme di valori possibili (fino a 64), ma

in questo caso la colonna può assumere anche più di un valore, oppure nessuno.

Dati geometrici

I dati geometrici sono stati introdotti con la versione 4.1 di MySQL, e si basano sulle specifiche dell'*Open GIS Consortium*. Dovreste conoscere il Modello Geometrico proposto da tale ente per poter utilizzare con proprietà questi dati. Sul **manuale di MySQL** troverete informazioni sul modello e rimandi ad altri siti web sull'argomento.

Qui vediamo quali sono i tipi di dati geometrici previsti da MySQL:

GEOMETRY
POINT
LINESTRING
POLYGON
MULTIPOINT
MULTILINESTRING
MULTIPOLYGON
GEOMETRYCOLLECTION

Il significato di ogni dato è piuttosto intuitivo: GEOMETRY può contenere un valore geometrico generico; POINT contiene un punto, LINESTRING una linea, POLYGON un poligono. GEOMETRYCOLLECTION rappresenta un insieme di dati geometrici di qualsiasi tipo, mentre gli altri tre sono insiemi di dati del tipo relativo.

Questi dati sono disponibili sulle tabelle MyISAM e, a partire da MySQL 5.0.16, anche su tabelle InnoDB e ARCHIVE.

Indici e chiavi

Effettuare una query su un database è in genere abbastanza semplice, grazie alla sintassi intuitiva del linguaggio SQL. Più complesso può essere, invece, ottimizzarne le performance. Per questo scopo esistono degli strumenti appositi: gli **indici**.

Un indice è una struttura dati ausiliaria che consente di recuperare più velocemente i dati di una tabella, evitandone la lettura dell'intero contenuto (*full table scan*), tramite una selezione più mirata.

Gli indici vanno usati consapevolmente, verificando quando sono effettivamente necessari e scegliendo con cura su quali campi della tabella applicarli. Un loro abuso, infatti, potrebbe avere addirittura l'effetto di ridurre le performance di interfacciamento con il database. Infatti, tali strutture dati vanno aggiornate ad ogni modifica apportata alla tabella, e quindi se da un lato gli indici agevolano le operazioni di lettura, dall'altro rendono più onerose tutte le altre.

Con MySQL si possono creare indici su qualunque tipo di dato, ma i loro dettagli applicativi dipendono dallo **Storage Engine** scelto per la tabella. Gli Storage Engine verranno trattati più avanti in questa guida, e per il momento è sufficiente sapere che si tratta dei gestori del salvataggio e reperimento dei dati su disco.

Nel seguito della lezione, si illustreranno le azioni collegate alla **creazione e gestione degli indici**. Tutti i comandi, espressi nella sintassi SQL, possono essere verificati tramite il client *mysql* avendo a disposizione un'istanza in funzione del DBMS ed un database già creato.

Tipi di indici

Esistono diversi tipi di indici:

- **PRIMARY KEY**: applicato ad uno o più campi di una tabella permette di distinguere univocamente ogni riga. Il campo sottoposto all'indice primary key non ammette duplicati né campi nulli;
- **UNIQUE**: simile alla primary key, con la differenza che tollera valori nulli, mentre i duplicati restano vietati;
- **COLUMN INDEX**: sono gli indici più comuni. Applicati ad un campo di una tabella, hanno puramente lo scopo di velocizzarne l'accesso permettendo valori duplicati e nulli. Come variante, possono esistere indici "multicolonna", che includono quindi più campi della tabella, oppure i cosiddetti **PREFIX INDEX** che, nei campi stringa, permettono di indicizzare non tutto il campo ma solo una porzione iniziale di caratteri, appunto il prefisso;

- **FULLTEXT**: sono indici che permettono di accelerare operazioni onerose, come la ricerca testuale su un intero campo.

Creare chiavi primarie

La chiave primaria viene solitamente creata in fase di definizione di una tabella. L'abbiamo già visto in una lezione precedente: si sceglie un campo indicato per tale ruolo e lo si designa ufficialmente come tale usando la keyword *PRIMARY KEY*.

```
CREATE TABLE `Persone` (
  `id` INT NOT NULL AUTO_INCREMENT,
  ...
  ...
  PRIMARY KEY (`id`))
```

Dovendo possedere caratteristiche di unicità, il campo ideale da scegliere come chiave primaria dovrebbe essere univocamente distintivo del record in cui si trova (possono andare bene dati come partite IVA, codici fiscali, numeri di documenti di identità, eccetera, in quanto univoci per definizione). Nel caso non ve ne siano, viene adottato – e non succede di rado – un numero intero progressivo, incrementato automaticamente dal DBMS ogni volta che si inserisce una nuova riga. L'esempio che vedremo più avanti utilizza questa opzione.

Analogamente, possono essere scelte più colonne per costituire una chiave primaria. In questo caso, i campi nel complesso costituiranno un'identificazione univoca per la riga.

Una chiave primaria può anche essere creata successivamente alla creazione della tabella:

```
CREATE PRIMARY KEY ON nome_tabella (elenco campi);
```

Creare indici

Al pari delle chiavi primarie, gli indici possono essere creati contestualmente alla tabella o aggiunti successivamente.

Nel primo caso, la dichiarazione avverrà all'interno del costruttore *CREATE TABLE*:

```
CREATE TABLE `Persone` (
  ...
  `eta` INT NOT NULL,
  ...
  INDEX eta_ind (`eta`) );
```

Nello stralcio di codice precedente, viene creato un indice sul campo *eta*. Ciò agevolerà selezioni di righe in base a questo valore. Si noti, inoltre, che l'indice possiede un nome che lo distingue, in questo caso *eta_ind*.

In alternativa, l'indice può essere creato successivamente alla definizione della tabella:

```
CREATE INDEX eta_ind ON Persone (`eta`) ;
```

L'**eliminazione di un indice** avviene tramite comando *DROP*:

```
DROP INDEX eta_ind ON Persone;
```

Altro aspetto interessante è che si possono creare, come accennato, dei **PREFIX INDEX**, basati su un prefisso di un campo stringa, ossia solo sui primi caratteri del valore del campo. Ciò può essere utile in campi testuali, dove realizzare un PREFIX INDEX può comunque agevolare le ricerche senza rendere troppo onerosa la gestione dell'indice:

```
CREATE TABLE `Persone` (
  ...
  `cognome` VARCHAR(20) NOT NULL,
  ...
  INDEX cognome_ind (cognome(6)) );
```

Un *PREFIX INDEX* viene dichiarato nella stessa maniera di un indice normale, l'unica differenza consiste nel numero tra parentesi tonde, che indica quanti byte (e non quanti caratteri) verranno catalogati.

FOREIGN KEYS e vincoli

Le *FOREIGN KEYS* sono costrutti che sfruttano gli indici per collegare due tabelle mediante l'associazione di campi. Applicare vincoli ai vari campi di diverse tabelle consente di mantenere la **consistenza dei dati**.

L'uso di questa tecnica è parte integrante dello Storage Engine **InnoDB**, che dalla versione 5.5 del DBMS è impostato di default nelle tabelle.

Sin dalle prime lezioni di questa guida, si è spiegato che in un database relazionale le tabelle sono collegate tra loro tramite relazioni logiche, facendo sì che righe di una tabella contengano riferimenti a valori chiave di un'altra tabella.

Tipicamente, in queste relazioni, esistono delle tabelle principali ed altre secondarie. Ad esempio, in un database che gestisce utenti ed i loro contatti, potremmo avere una

tabella principale per catalogare i dati personali ed una secondaria per contenere uno o più riferimenti telefonici per utente:

```
CREATE TABLE utenti (`id` INT NOT NULL AUTO_INCREMENT,
        nome VARCHAR(50),
        ...
        ...
        PRIMARY KEY (`id`)
) ;

CREATE TABLE contatti(`id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
             utente_id INT,
             telefono VARCHAR(20),
           INDEX utente_ind (utente_id),
           FOREIGN KEY (utente_id) REFERENCES utenti(`id`)
);
```

L'esempio è schematico ma mette in luce i principali aspetti di questo tipo di legame:

- le tabelle devono avere lo stesso Storage Engine, InnoDB, ed i campi coinvolti nella relazione devono essere di tipi simili;
- la clausola *FOREIGN KEY* deve essere contenuta all'interno della definizione della tabella secondaria;
- i campi coinvolti nella relazione devono essere sottoposti ad un indice di qualche tipo, sia nella tabella principale che in quella secondaria, in modo da renderli accessibili in maniera efficiente;
- non possono essere coinvolti indici di tipo *PREFIX*, quindi non si possono utilizzare campi *BLOB* o *TEXT*.

Il mantenimento dell'**integrità referenziale** (ciò che consente di mantenere la consistenza dei dati logicamente connessi, su tabelle distinte) segue due regole.

La prima vieta di inserire o modificare dati nella tabella secondaria che non abbiano collegamento con quelli della tabella principale. Ad esempio, nella tabella *contatti*, ogni numero di telefono inserito deve corrispondere ad un ID di un utente esistente nella tabella principale. Operazioni che violano questa regola verranno respinte da MySQL che addurrà come motivazione la violazione del vincolo imposto dalla foreign key.

La seconda regola dispone che, in caso di cancellazione o modifica di dati nella tabella principale, un apposito vincolo debba specificare quale azione dovrà essere applicata nella tabella secondaria.

Un vincolo di tale genere può essere specificato in fase di definizione della foreign key. Per gestire la cancellazione possiamo procedere come segue:

```
FOREIGN KEY (utente_id) REFERENCES utenti(`id`)
ON DELETE azione da attivare
```

Se invece vogliamo gestire il caso dell'aggiornamento:

```
FOREIGN KEY (utente_id) REFERENCES utenti(`id`)
ON UPDATE azione da attivare
```

Le azioni attivabili sono cinque:

- **CASCADE**: la cancellazione o la modifica di una riga nella tabella principale causerà, a cascata, la medesima modifica nella tabella secondaria;
- **SET NULL**: il campo oggetto della relazione nella tabella secondaria verrà impostato a *NULL*. In questo caso, è necessario che tale campo non sia stato qualificato come *NOT NULL* in fase di creazione;
- **RESTRICT**: impedisce che la modifica o la cancellazione nella tabella principale venga eseguita. Equivale a non specificare alcun vincolo, in pratica è l'azione di default;
- **NO ACTION**: in MySQL è un sinonimo di *RESTRICT*, quindi vale quanto detto al punto che precede;
- **SET DEFAULT**: nonostante il parser di MySQL la riconosca come valida, questa impostazione è vietata dal motore InnoDB.

Il CharSet

I **character set** (insiemi di caratteri) sono i diversi sistemi attraverso i quali i caratteri alfanumerici, i segni di punteggiatura e tutti i simboli visualizzabili su un computer vengono memorizzati in un valore binario.

In ogni set di caratteri, ad un valore binario corrisponde un carattere ben preciso. Di conseguenza, quando una stringa viene memorizzata utilizzando un certo set di caratteri, dovrà essere visualizzata attraverso quello stesso insieme, altrimenti alcuni caratteri potrebbero apparire diversi da come ce li aspettiamo.

L'esempio classico di questo inconveniente si verifica in genere con le lettere accentate e, a volte, con il simbolo dell'Euro, che ogni tanto capita di vedere non rappresentati correttamente, ad esempio su una pagina web: segno che non stiamo visualizzando quella pagina con il giusto insieme di caratteri.

MySQL, a partire dalla versione 4.1, ha introdotto un supporto molto avanzato alla gestione di diversi *character set*. Infatti ci consente di gestire i set di caratteri a livello di server, database, tabella e singola colonna, nonchè di client e di connessione.

Ad ogni set di caratteri sono associate una o più **collation**, che rappresentano i modi possibili di confrontare le stringhe di caratteri facenti parte di quel character set. Questo termine potrebbe essere tradotto con l'italiano *collazione*, al quale lo Zingarelli attribuisce, fra gli altri, il significato di "confronto", segnalandolo però come non più in uso.

Per fare un esempio quindi potremo dire che una determinata tabella utilizza il character set *latin 1* (quello maggiormente usato in Europa Occidentale) e la collation *latin1_general_cs*. Tale collation è multilingue (cioè non specifica per una lingua) e "case sensitive" come dimostra il "cs" finale, cioè tiene conto della differenza fra maiuscole e minuscole nell'ordinare o confrontare le stringhe.

In generale possiamo dire che il nome di ogni collation segue un determinato standard: inizia con il nome del character set a cui si riferisce, comprende di solito una specifica relativa ad una lingua, e termina con *cs* (case sensitive) o *ci* (case insensitive) a seconda che tenga o meno in considerazione la differenza tra maiuscole e minuscole, oppure con *bin* quando il valore binario dei caratteri è utilizzato direttamente per i confronti.

Quindi avremo, ad esempio, *latin1_swedish_ci* per la collation svedese case insensitive di latin1, e *latin1_german2_ci* per la collation, sempre case insensitive, basata sulle regole tedesche DIN-2, mentre *utf8_bin* è la collation binaria della codifica utf8

(Unicode). Normalmente l'esistenza di una collation dedicata ad una singola lingua si ha quando le regole generali del set di caratteri non soddisfano le esigenze di quella lingua.

Le collation dedicate al tedesco, ad esempio, servono a trattare nel giusto modo le vocali con dieresi (ad esempio Ü) e la lettera "sharp" (ß).

Le istruzioni SQL "SHOW CHARACTER SET" e "SHOW COLLATION" ci consentono di ottenere la lista, rispettivamente, dei set di caratteri e delle collation disponibili sul server. Dei primi viene mostrata, per ogni insieme, la collation di default. Delle seconde possiamo vedere a quale set di caratteri appartengono: è chiaro infatti, da ciò che abbiamo detto finora, che ogni collation è legata ad un singolo character set.

Come detto, MySQL gestisce character set e collation a diversi livelli: server, database, tabella, colonna. Parlando di dati memorizzati, ovviamente ciò che è rilevante è quale charset viene utilizzato per ogni colonna (di tipo CHAR, VARCHAR o TEXT). Tutti i valori di livello superiore, quindi, hanno il solo scopo di funzionare da default per il rispettivo livello inferiore.

A livello di server abbiamo le variabili *default_character_set* e *default_collation*, per le quali valgono le solite regole relative alle variabili di sistema: quindi possono essere inizializzate ad esempio da un file di opzioni, e anche modificate a runtime. In mancanza di inizializzazione il charset di default sarà *latin1* con la collation *latin1_swedish_ci*.

Vediamo ora alcuni esempi di definizioni SQL:

```
CREATE DATABASE db1 [CHARACTER SET utf8] [COLLATE utf8_general_ci];
CREATE TABLE tabella1 (
        colonna1 VARCHAR(5) CHARACTER SET latin1 COLLATE latin1_german_ci,
        colonna2 VARCHAR(5) CHARACTER SET latin1,
        colonna3 VARCHAR(10)
) [DEFAULT CHARACTER SET latin1 [COLLATE latin1_general_ci]];
```

Come sempre, le espressioni fra parentesi quadre vanno intese come facoltative. La prima istruzione crea un database i cui default sono character set utf8 e collation utf8_general_ci.

Non indicare la collation sarebbe stato indifferente, perchè utf8_general_ci è quella di default per utf8. Non indicare utf8 avrebbe significato utilizzare, come default per il database, i valori del server. Teniamo presente che i due valori viaggiano sempre accoppiati, nel senso che non è possibile trasmettere "verso il basso" il default della collation indipendentemente da quello del character set, per il motivo visto prima che

ogni collation è legata a un solo set di caratteri. Quindi, ad ogni livello, o si ereditano entrambi i valori dal livello superiore, oppure, una volta stabilito il charset, la collation, se non espressa esplicitamente, sarà il default per quel charset.

Proseguiamo coi nostri esempi: nella seconda istruzione viene creata una tabella con charset di default latin1 e collation latin1_general_ci; vedete quindi come i valori del database siano completamente ignorati per questa tabella. All'interno della tabella abbiamo colonna1, per la quale abbiamo dichiarato entrambi i valori; colonna2 che è dichiarata esplicitamente come charset latin1: non avendo indicato una collation, sarà usata latin1_swedish_ci, che è il default per latin1, e **non il default della tabella**. Per colonna3 invece non abbiamo indicato niente, per cui valgono i default della tabella.

Occupiamoci ora delle impostazioni altrettanto importanti relative ai client e alle connessioni. Le variabili di sistema *character_set_client*, *character_set_results* rappresentano, rispettivamente, il charset delle istruzioni in arrivo dal client e quello che sarà utilizzato per spedire le risposte; abbiamo poi *character_set_connection* e *collation_connection*, che sono utilizzate dal server per convertire le istruzioni ricevute fare confronti fra le costanti stringa.

Le più importanti sono le prime due, che devono permettere al client di dialogare correttamente col server: tali impostazioni infatti **dovranno riflettere l'effettivo set di caratteri utilizzato dal client**. I seguenti comandi si utilizzano per modificare queste impostazioni:

```
SET NAMES 'x';
SET CHARACTER SET 'x';
```

dove 'x' è un character set. La prima istruzione imposta a 'x' i valori di character_set_client, character_set_results e character_set_connection, mentre collation_connection sarà il default per il relativo charset. Con la seconda invece vengono impostati a 'x' i valori di character_set_client e character_set_results, mentre i valori della connessione saranno quelli di default del server.

Se utilizzate il client mysql sul prompt dei comandi di Windows, molto probabilmente avrete difficoltà nella visualizzazione e nell'immissione delle lettere accentate: questo è dovuto al fatto che in genere il Windows prompt utilizza il charset cp850 (per scoprirlo digitate il comando DOS 'chcp').

In questo caso quindi dovrete utilizzare l'istruzione *SET NAMES cp850* per operare in modo corretto. È tuttavia ovvio che nemmeno l'utilizzo del charset corretto può supplire alla mancanza di certi caratteri nel character set stesso: con il cp850 ad esempio non sarà possibile visualizzare nè inserire il simbolo dell'Euro.

Infine va ricordato che le proprietà di un database, relative a charset e collation, definite in fase di creazione, possono essere modificate successivamente, utilizzando il seguente comando:

```
ALTER DATABASE nome_db
[CHARACTER SET charset] [COLLATE collation]
```

InnoDB, MyISAM e gli Storage Engine

In una delle lezioni precedenti, quando si è parlato della creazione di tabelle, abbiamo incontrato la parola chiave `ENGINE`:

```
CREATE TABLE Persone
(...)
ENGINE=InnoDb
```

In questo caso il codice definirebbe la tabella denominata *Persone*, con **engine InnoDB**. Semplicisticamente, si potrebbe pensare agli Storage Engine come i "tipi" delle tabelle. In realtà, essi sono delle librerie (prodotte congiuntamente al DBMS o da enti terzi) che determinano il modo in cui i dati di quella tabella saranno salvati su disco, e ciò sarà determinante per valutare le prestazioni, l'affidabilità, le funzionalità offerte dalla tabella stessa, rendendola più o meno adatta a particolari utilizzi.

A partire dalla versione 5.5 di MySQL, **InnoDB è lo Storage Engine di default**, ossia quello assegnato automaticamente qualora, in fase di creazione della tabella, non si specifichi il parametro `ENGINE`.

Per sapere quali storage engine sono a disposizione della propria installazione del DBMS, possiamo eseguire, tramite il client testuale *mysql*, il seguente comando:

L'output mostrerà una tabella stilizzata che elenca tutti gli engine a disposizione e le principali funzionalità che li caratterizzano. In particolare, il campo *Support* sarà valorizzato con `YES`, `NO` o `DEFAULT`. I primi due valori specificano se lo storage engine può essere usato o no, l'ultimo indica se l'engine è quello di default.

In base alle proprie necessità, è possibile modificare lo Storage Engine di default:

```
SET storage_engine=MyISAM;
```

Il comando precedente fa sì che lo Storage Engine **MyISAM** sia impostato come opzione di default. Tramite `SHOW ENGINES` è possibile verificare se la modifica è stata attuata.

InnoDB

InnoDB è lo Storage Engine di default di MySQL. Lo scopo di InnoDB è quello di associare maggiore sicurezza (intesa soprattutto come consistenza ed integrità dei dati) a performance elevate.

Consistenza dei dati e velocità di elaborazione spesso possono contrastare, in quanto la prima proprietà necessita di applicare opportuni lock sui dati durante le modifiche, per evitare che altre richieste vi accedano; d'altro canto, per ottenere una maggiore velocità di elaborazione è richiesto che vengano servite contemporaneamente più richieste, agevolando la concorrenza.

Nell'ottica di perseguire tali finalità, InnoDB è stato arricchito di alcune funzionalità peculiari:

- **supporto alle transazioni**: per transazione (concetto che sarà approfondito più avanti nella guida) si intende la possibilità di un DBMS di svolgere più operazioni di modifica dei dati, facendo sì che i risultati diventino persistenti nel database solo in caso di successo di ogni singola operazione. In caso contrario, verranno annullate tutte le modifiche apportate;
- **`FOREIGN KEYS`**: conferiscono la possibilità di creare una relazione logica tra i dati di due tabelle, in modo da impedire modifiche all'una che renderebbero inconsistenti i dati dell'altra;
- **lock a livello di riga** piuttosto che di tabella: questa caratteristica permette di limitare le porzioni di dati sottoposte al lock durante le modifiche. Questo genere di lock può essere di due tipi: *shared* (permette alla transazione che mantiene il lock di leggere la tabella) o *esclusivo* (la transazione che attua il lock può anche modificare o cancellare i dati);
- **MVCC** (**M**ultiversion **C**oncurrency **C**ontrol) si tratta di una tecnica molto potente che permette di incrementare la concorrenza nelle interrogazioni, senza subire rallentamenti a causa dei lock applicati da altre transazioni. In pratica, permette di eseguire query su righe che, contemporaneamente, sono in fase di aggiornamento, leggendo i dati così come apparivano prima delle modifiche in corso;
- **clustered indexes**: sono un tipo di indici che molto spesso coincidono con la chiave primaria. In assenza di questa possono essere definiti autonomamente da MySQL. Il motivo dell'efficienza di tali indici consiste nella loro definizione all'interno dello stesso file che contiene i dati della riga.

La **dimensione di una tabella InnoDB** può raggiungere i **64 TB**, un valore inferiore a quello ammesso da altri Storage Engine.

MyIsam

Per molto tempo MyISAM è stato lo Storage Engine di default, finché non ha dovuto lasciare il passo alle ricche caratteristiche di InnoDB. Le tabelle che usano questo Storage Engine possono raggiungere dimensioni di **256 TB** e vengono salvate su disco divise in tre file differenti: uno di estensione *.frm* che contiene il formato della tabella, uno *.MYD* che contiene i dati, ed un ultimo, *.MYI*, che contiene gli indici.

Questo engine attualmente presenta performance ridotte a causa del lock eseguito a livello di tabella. Ciò è particolarmente evidente quando si tratta di tabelle soggette indifferentemente ad operazioni di lettura e scrittura. Al contrario, tale effetto negativo è ridotto se la tabella è read-only o comunque modificata molto raramente. Per queste ultime casistiche, MyISAM appare ancora un'opzione accettabile.

L'ampio uso fatto negli anni delle tabelle MyISAM ne ha dimostrato abbondantemente la loro affidabilità. Tuttavia, può capitare che esse risultino corrotte e, di conseguenza, non più leggibili. Ciò può accadere a seguito di circostanze particolari seppur piuttosto rare: guasti hardware, interruzioni improvvise del demone *mysqld*, spegnimento inatteso della macchina ospite. Tabelle MyISAM danneggiate possono essere controllate e riparate con i seguenti strumenti:

- i comandi `CHECK TABLE` e `REPAIR TABLE`, rispettivamente, per controllare e riparare la tabella. Il loro uso è possibile se il DBMS è in esecuzione;
- lo strumento *myisamchk*, a corredo di MySQL, da usare se il demone non è attivo.

Memory

Questo storage engine ha la caratteristica di non salvare i dati in maniera persistente su disco, ma di mantenerli in memoria. La conseguenza sarà di avere a disposizione tabelle molto veloci, ottime come supporto allo svolgimento di operazioni piuttosto lunghe. Al contrario, non è utilizzabile per il salvataggio duraturo dei dati. Il loro utilizzo si è ridotto nel tempo grazie ad un'altra caratteristica molto importante di InnoDB, che consiste nell'utilizzare come cache un buffer pool in memoria per agevolare operazioni di lettura che coinvolgono grandi quantità di dati. L'avvertenza doverosa nei confronti di Memory, che rimane comunque una tecnologia molto utile, è di non far assumere a questo tipo di tabelle dimensioni eccessive, che potrebbero penalizzare le prestazioni dell'intero DBMS.

Archive

Lo storage engine Archive risponde all'esigenza di conservare nel tempo grandi moli di dati, non più aggiornate ma archiviate solo a carattere "storico" o per eventuali analisi future (per esempio nel caso dei log di sistema). Tabelle di questo tipo permettono solo

operazioni di `SELECT` ed `INSERT` e non possono avere campi indicizzati, ma hanno il vantaggio di occupare meno spazio, in quanto **compresse** con *zlib*.

Ulteriori storage engine

Esistono molti altri storage engine, non sempre frequenti nell'uso ma particolarmente utili se impiegati negli ambiti per cui sono stati progettati:

- **CSV**: memorizza le informazioni come testo, in cui ogni riga contiene i campi separati da virgole. Non sono indicizzati e vengono utilizzati per l'esportazione o l'importazione di dati che, per le operazioni comuni, sono custoditi in tabelle gestite con altri Storage Engine;
- **Blackhole**: è paragonabile al dispositivo */dev/null* dei sistemi Unix-like. Molto utile nei test, permette di eseguire comandi per verificarne la correttezza. Non verranno inseriti dati e le query restituiranno sempre insiemi nulli;
- **Merge** e **Federated**: sono due storage engine separati rivolti entrambi alla tematica dell'**aggregazione**. Il primo permette di utilizzare tabelle diverse ma accomunate dalla medesima struttura, come se fossero una sola. Il secondo offre accesso unitario a più server MySQL: utile per gestire più istanze del DBMS come una sola entità logica;
- **Example**: è uno storage engine di esempio. Ha il solo scopo di mostrare la struttura che deve avere uno storage engine. Utile come modello per sviluppatori;
- **NDB Cluster**: è uno storage engine dedicato alla gestione di **cluster di database**, e costituisce il motore di gestione delle tabelle di dati utilizzate in MySQL Cluster.

INSERT: inserimento dei dati

L'inserimento dei dati in una tabella avviene tramite l'istruzione **INSERT**. Ovviamente dovremo avere il permesso di INSERT sulla tabella.

Vediamo quali sono le diverse sintassi che possiamo utilizzare:

INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
[INTO] *nome_tabella* [(*nome_colonna*,...)]
VALUES ({*espressione* | DEFAULT},...),(...),...
[ON DUPLICATE KEY UPDATE *nome_colonna=espressione*, ...]

Con questa sintassi possiamo inserire una o più righe nella tabella. Prima della clausola VALUES è possibile indicare i nomi delle colonne interessate dalla INSERT: a questi nomi corrisponderanno i valori compresi in ogni parentesi dopo VALUES. Per inserire più righe useremo più coppie di parentesi tonde dopo VALUES. Se non indichiamo la lista delle colonne, dovremo fornire un valore per ogni colonna della tabella, nell'ordine in cui sono definite.

INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
[INTO] *nome_tabella*
SET *nome_colonna*={*espressione* | DEFAULT}, ...
[ON DUPLICATE KEY UPDATE *nome_colonna=espressione*, ...]

In questo caso usiamo la clausola SET per assegnare esplicitamente un valore ad ogni colonna indicata. Con questa sintassi è possibile inserire una sola riga.

INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE]
[INTO] *nome_tabella* [(*nome_colonna*,...)]
SELECT ...
[ON DUPLICATE KEY UPDATE *nome_colonna=espressione*, ...]

Qui utilizziamo direttamente una SELECT per fornire i valori alla nuova tabella. Con questo sistema si possono inserire più righe. Come nel primo caso, è possibile elencare le colonne interessate; in caso contrario la SELECT dovrà fornire valori per tutte le colonne.

Vediamo ora la funzione delle varie clausole utilizzabili:

LOW PRIORITY: l'inserimento non viene effettuato fino a quando esistono client che leggono sulla tabella interessata; questo può portare ad attese anche lunghe.

DELAYED: anche in questo caso l'inserimento viene ritardato fino a quando la tabella non è libera. La differenza rispetto al caso precedente è che al client viene dato immediatamente l'ok, e le righe da inserire vengono mantenute in un buffer gestito dal server fino al momento della effettiva scrittura. Ovviamente le righe non saranno visibili con una SELECT fino a quando non saranno inserite sulla tabella.

HIGH PRIORITY: annulla l'effetto di una eventuale opzione –low-priority-updates che fosse attiva sul server.

IGNORE: permette di gestire eventuali errori che si verificano in fase di inserimento (chiavi duplicate o valori non validi); invece di generare errori bloccanti, i record con chiavi doppie vengono semplicemente scartati, mentre i valori non validi vengono "aggiustati" al valore più prossimo.

ON DUPLICATE KEY UPDATE: nel caso in cui si verifichi una chiave doppia, l'istruzione specificata viene eseguita sulla riga preesistente. Con questa opzione non è possibile usare DELAYED.

I **valori** da inserire nella tabella con le prime due sintassi possono essere indicati da costanti o espressioni, oppure richiamando esplicitamente il DEFAULT. Il default viene usato anche per le colonne non specificate. Tuttavia, in **strict mode**, è obbligatorio specificare valori per tutte le colonne che non hanno un default esplicito; in caso contrario si avrà un errore.

Le colonne di tipo AUTO_INCREMENT vengono valorizzate automaticamente indicando NULL (oppure tralasciandole). Per conoscere il valore generato si può usare, dopo l'inserimento, la funzione LAST_INSERT_ID(), che restituisce l'ultimo valore creato nel corso della connessione attuale.

Oltre alla INSERT, MySQL offre l'istruzione **REPLACE**, che è un'estensione allo standard SQL e che consente di sostituire le righe preesistenti con le righe inserite qualora si verifichi una situazione di chiave doppia. In pratica, usando REPLACE, qualora non sia possibile inserire una riga perchè una PRIMARY KEY o un indice UNIQUE esistono già sulla tabella, MySQL cancella la riga vecchia ed inserisce la nuova. Questo comportamento è opposto a quello di INSERT IGNORE, con il quale è la nuova riga ad essere scartata.

Per effettuare una REPLACE dovremo avere i permessi di INSERT e DELETE; le sintassi

sono pressochè identiche a quelle della INSERT; vediamole:

REPLACE [LOW_PRIORITY | DELAYED]
[INTO] *nome_tabella* [(*nome_colonna*,...)]
VALUES ({*espressione* | DEFAULT},...),(...),...
oppure
REPLACE [LOW_PRIORITY | DELAYED]
[INTO] *nome_tabella*
SET *nome_colonna*={*espressione* | DEFAULT}, ...
oppure
REPLACE [LOW_PRIORITY | DELAYED]
[INTO] *nome_tabella* [(*nome_colonna*,...)]
SELECT ...

Un altro modo di inserire dati in una tabella è quello che ci consente di importare un file di testo: **LOAD DATA INFILE**. Vediamone la sintassi:

LOAD DATA [LOW_PRIORITY | CONCURRENT] [LOCAL] INFILE '*nome_file*.txt'
[REPLACE | IGNORE]
INTO TABLE *nome_tabella*
[FIELDS
[TERMINATED BY '*stringa*']
[[OPTIONALLY] ENCLOSED BY '*char*']
[ESCAPED BY '*char*']
]
[LINES
[STARTING BY '*stringa*']
[TERMINATED BY '*stringa*']
]
[IGNORE *numero* LINES]
[(*nome_colonna_o_variabile*,...)]
[SET *nome_colonna = espressione*,...)]

Le opzioni **LOW_PRIORITY** e **IGNORE** funzionano come per una INSERT. L'opzione **REPLACE** permette all'istruzione di funzionare come una REPLACE.

L'opzione **LOCAL** specifica che il file da leggere si trova sulla macchina del client. Se non è presente, si assume che il file si trovi sulla macchina del server. In quest'ultimo caso è necessario il privilegio FILE per eseguire l'operazione.

Se si usa **FIELDS** è obbligatorio indicare almeno una delle opzioni che la compongono:

- TERMINATED BY indica quale stringa separa i campi;

- ENCLOSED BY indica i caratteri usati per racchiudere i valori;
- ESCAPED BY specifica il carattere di escape usato per i caratteri speciali (cioè quelli utilizzati nelle altre opzioni di FIELDS e LINES)

LINES può indicare le opzioni per le righe: STARTING BY indica una stringa che sarà omessa in ogni riga (può anche non trovarsi all'inizio della riga, nel qual caso sarà omesso tutto ciò che si trova prima).

TERMINATED BY indica il carattere di fine riga. I default per FIELDS e LINES sono i seguenti:

FIELDS TERMINATED BY 't' ENCLOSED BY '' ESCAPED BY '\' LINES TERMINATED BY 'n' STARTING BY ''

Quindi: campi suddivisi da tabulazioni e nessun carattere a racchiuderli; righe senza prefisso e che terminano con il carattere di newline; backslash come carattere di escape (per tabulazioni e newline).

IGNORE *n* LINES si usa per saltare le prime *n* righe del file di input (ad esempio perchè contengono intestazioni).

Può essere indicata una lista di nomi di colonna o variabili alle quali assegnare i valori letti dal file. Se questa lista non viene fornita, si presume che il file contenga in ogni riga un valore per ogni colonna della tabella; se si forniscono variabili, i valori non finiranno direttamente nella tabella, ma potranno essere utilizzati nella clausola **SET** per effettuare elaborazioni; con tale clausola è anche possibile assegnare valori ad altre colonne indipendentemente dai dati presenti nel file (ad esempio un timestamp).

UPDATE e DELETE: modifica e cancellazione dei dati

Dopo aver visto come inserire e cercare i dati, vediamo ora come modificarli o eliminarli. Per gli aggiornamenti si usa l'istruzione **UPDATE**, di cui vediamo la sintassi:

UPDATE [LOW_PRIORITY] [IGNORE] *nome_tabella*
SET *nome_colonna=espressione* [, *nome_colonna2=espressione2* ...]
[WHERE *condizioni*]
[ORDER BY ...]
[LIMIT *numero_righe*]

Il funzionamento è abbastanza intuitivo:

- dopo UPDATE indichiamo quale tabella è interessata
- con SET specifichiamo quali colonne modificare e quali valori assegnare
- con WHERE stabiliamo le condizioni che determinano quali righe saranno interessate dalle modifiche (se non specifichiamo una WHERE tutte le righe saranno modificate)

Inoltre possiamo usare ORDER BY per decidere in che ordine effettuare gli aggiornamenti sulle righe, e LIMIT per stabilire un numero massimo di righe che saranno modificate. Evidentemente l'uso di ORDER BY difficilmente ha senso se non accoppiato con LIMIT.

L'UPDATE restituisce il numero di righe modificate; attenzione però: se tentate di assegnare ad una riga valori uguali a quelli che ha già, MySQL se ne accorge e non effettua l'aggiornamento. Ai fini della LIMIT la riga viene comunque conteggiata.

È possibile anche usare LOW_PRIORITY, come già visto per le INSERT, per ritardare l'esecuzione dell'aggiornamento ad un momento nel quale la tabella non è impegnata da altri client. Con la clausola IGNORE invece indichiamo al server di ignorare gli errori generati dall'aggiornamento. Eventuali modifiche che causassero chiavi doppie non saranno, in questo caso, effettuate.

In una UPDATE è possibile fare riferimento ad una colonna per utilizzare il suo valore precedente all'aggiornamento; ad esempio:

UPDATE vendite SET venduto=venduto+1 WHERE idVenditore=5;

In questo caso il valore della colonna *venduto* viene incrementato di 1.

L'operazione di UPDATE può essere effettuata anche su più tabelle. In questo caso indicheremo i nomi delle tabelle interessate con la stessa sintassi già vista per le join. Con gli update multi-tabella però non è possibile usare le clausole ORDER BY e LIMIT.

Per effettuare una UPDATE è necessario avere il privilegio UPDATE sulle tabelle da modificare più il privilegio SELECT su eventuali altre tabelle a cui viene fatto accesso in sola lettura.

Passiamo ora alla **DELETE**, con la quale cancelliamo una o più righe da una o più tabelle (per farlo dobbiamo avere il privilegio DELETE).
Questa è la sintassi per la DELETE su una sola tabella:

DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM *nome_tabella*
[WHERE *condizioni*]
[ORDER BY ...]
[LIMIT *numero_righe*]

Le opzioni LOW_PRIORITY e IGNORE hanno lo stesso significato visto per la UPDATE. La clausola QUICK è utile solo per tabelle MyISAM: velocizza l'operazione ma non effettua l'ottimizzazione degli indici. È utile se i valori degli indici cancellati saranno sostituiti da valori simili.

Anche ORDER BY e LIMIT funzionano come nella UPDATE: permettono di stabilire l'ordine delle cancellazioni e di limitare il numero di righe cancellate. Con la WHERE stabiliamo le condizioni in base alle quali le righe verranno eliminate. Se non la indichiamo, tutte le righe saranno eliminate.

La cancellazione di righe da una tabella può portare alla presenza di spazio inutilizzato nella tabella stessa: se si effettuano molte DELETE su una tabella sarà bene effettuare periodicamente una OPTIMIZE TABLE oppure eseguire l'utility *myisamchk* (vedere lezione 24).

Vediamo ora come effettuare una DELETE su più tabelle: per questa operazione esistono due possibili sintassi

DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
nome_tabella[.*] [, *nome_tabella*[.*] ...]
FROM *tabelle*
[WHERE *condizioni*]
oppure

DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
FROM *nome_tabella*[.*] [, *nome_tabella*[.*] ...]
USING *tabelle*
[WHERE *condizioni*]

In questo caso può capitare che abbiamo la necessità di cancellare righe da una o più tabelle leggendo i dati anche da altre tabelle, senza cancellare niente da queste ultime. Le tabelle che subiranno le cancellazioni sono elencate dopo DELETE nella prima sintassi, e dopo FROM nella seconda. La join sulle tabelle da cui leggere i dati viene invece espressa con la clausola FROM per il primo caso, e con USING nel secondo.

Vediamo due esempi equivalenti:

DELETE t1, t2 FROM t1, t2, t3 WHERE t1.id=t2.id AND t2.id=t3.id; DELETE FROM t1, t2 USING t1, t2, t3 WHERE t1.id=t2.id AND t2.id=t3.id;

In entrambi i casi verranno cancellate le righe corrispondenti da t1 e t2, ma solo quando esiste in t3 un ulteriore valore corrispondente. Come avrete notato, anche per la DELETE multi-tabella non è possibile usare le opzioni ORDER BY e LIMIT.

Quando vogliamo eliminare per intero il contenuto di una tabella possiamo utilizzare l'istruzione **TRUNCATE** che è più veloce:

TRUNCATE [TABLE] *nome_tabella*

Nella maggior parte dei casi, con la TRUNCATE la tabella viene eliminata e ricreata. Questo porta, fra l'altro, alla reinizializzazione dei valori AUTO_INCREMENT.

SELECT: interrogare MySQL

SELECT è sicuramente l'istruzione più utilizzata in SQL, in quanto è quella che ci permette di svolgere il lavoro fondamentale che deve fare un database, cioè recuperare i dati memorizzati.

Prima di analizzarne la sintassi è importante capire bene un concetto essenziale: ogni SELECT produce un risultato che è costituito, da un punto di vista logico, da **una tabella**, esattamente come quelle da cui va a leggere. Il risultato infatti è composto, come ogni tabella, da righe e colonne.

Le colonne sono quelle che indichiamo nella SELECT, mentre le righe sono selezionate dalle tabelle originali in base alle condizioni espresse nella clausola WHERE. Vediamo due esempi che rappresentano i casi opposti:

```
SELECT * FROM nome_tabella

SELECT COUNT(*) FROM nome_tabella
```

Questi due esempi sono apparentemente molto simili, ma in realtà producono effetti diametralmente opposti riguardo alla tabella risultato (o *resultset*).

Nel primo caso otterremo tutte le righe e tutte le colonne della tabella (in pratica una copia); infatti l'uso dell'**asterisco nella SELECT** significa *'voglio tutte le colonne'*, mentre l'assenza della `WHERE` fa sì che tutte le righe vengano restituite.

Nel secondo caso invece il resultset è formato da una sola riga e una sola colonna: infatti la colonna è costituita dall'espressione COUNT(*) (che significa 'voglio sapere quante sono le righe'), mentre il fatto di avere utilizzato una **funzione di colonna** (la stessa COUNT) fa sì che tutte le righe siano "concentrate" in una sola. Le funzioni di colonna svolgono proprio questo compito: quello di elaborare `n` righe (in questo caso tutte le righe della tabella, visto che di nuovo manca la WHERE) e riassumerle in un valore unico. Un altro esempio per chiarire meglio:

```
SELECT COUNT(*), MAX(colonna1) FROM nome_tabella WHERE colonna2 = valore
```

In questo caso avremo sempre una riga, visto che anche qui abbiamo usato funzioni di colonna. Le colonne però saranno due, perchè abbiamo chiesto due valori: il numero di righe e il valore massimo di `colonna1`. La presenza della clausola WHERE fa sì che vengano incluse nel conteggio solo le righe in cui il valore di `colonna2` è uguale a quello specificato.

Diamo ora un'occhiata alla sintassi della SELECT (alcune clausole sono state omesse per semplicità):

```
SELECT

    [ALL | DISTINCT | DISTINCTROW ]

    espressione, ... [INTO OUTFILE 'nome_file' opzioni | INTO DUMPFILE 'nome_file']

    [FROM tabelle

        [WHERE condizioni]

        [GROUP BY {nome_colonna | espressione | posizione}

        [ASC | DESC], ... [WITH ROLLUP]]

        [HAVING condizioni]

        [ORDER BY {nome_colonna | espressione | posizione}

        [ASC | DESC] , ...]

        [LIMIT [offset,] numero_righe]

    ]
```

Come vedete, la struttura fondamentale di una SELECT è la seguente:

- **SELECT**, seguita da una o più espressioni che saranno le colonne della tabella risultato;
- **FROM**, seguita dai nomi di una o più tabelle dalle quali devono essere estratti i dati;
- **WHERE**, che specifica le condizioni in base alle quali ogni riga sarà estratta oppure no dalle tabelle;
- **GROUP BY**, che specifica le colonne sui cui valori devono essere raggruppate le righe nel risultato: tutte le righe con valori uguali verranno ridotte a una;
- **HAVING**, che specifica ulteriori condizioni da applicare alle righe *dopo* il raggruppamento effettuato da `GROUP BY`;
- **ORDER BY**, che specifica in quale ordine figureranno le righe del resultset;
- **LIMIT**, che stabilisce il massimo numero di righe da estrarre

È possibile omettere tutta la parte da FROM in poi per effettuare query molto semplici che non fanno riferimento ad alcuna tabella, ad esempio:

che estrae la data e l'ora attuali.

Le *espressioni* che formeranno le colonne in output possono riferirsi a colonne delle tabelle referenziate, ma possono anche essere ottenute con funzioni applicate alle colonne, o con espressioni matematiche, o con funzioni che restituiscono valori indipendenti (come nel caso di NOW()).

Abbiamo già visto che se usiamo le cosiddette *funzioni di colonna* otterremo un valore unico che rappresenta *n* righe. Di conseguenza non possiamo avere un normale nome di colonna (valore scalare) di fianco ad una funzione di colonna:

```
SELECT colonna1, max(colonna2) FROM nome_tabella
```

Questa query produce un errore, perchè `colonna1` restituirebbe un valore per ogni riga della tabella, mentre `max(colonna2)` restituisce un valore unico. Di conseguenza non è possibile determinare quante righe deve avere la tabella risultato. L'unico modo per avere una `SELECT` in cui funzioni di colonna stanno accanto a valori scalari è di applicare una GROUP BY su questi ultimi: in questo modo le funzioni di colonna verranno applicate non a tutti i valori della tabella risultato, ma singolarmente ai gruppi di valori che fanno parte di ogni gruppo di righe.

Un esempio può chiarire meglio il concetto:

```
SELECT categoria, max(stipendio) FROM dipendenti GROUP BY categoria
```

Da questa query otterremo una riga per ogni diverso valore di `categoria`, ed in ogni riga il valore di `max(stipendio)` sarà riferito alle righe che hanno quel determinato valore di `categoria`. Così potremo sapere qual è lo stipendio massimo degli impiegati, dei quadri e dei dirigenti. In sostanza possiamo affermare che le funzioni di colonna vengono applicate ai gruppi di righe, e solo se non prevediamo raggruppamenti verranno applicate all'intera tabella risultato.

La clausola **FROM** indica la tabella o le tabelle da cui i dati saranno estratti. Le query più semplici estraggono dati da una sola tabella, ma è molto frequente aver bisogno di combinare più tabelle. In questo caso si effettua una JOIN.

Poiché ogni tabella appartiene a un database, la forma completa è `nome_database.nome_tabella`. Se non indichiamo un nome di database si sottintende l'uso del database corrente. Inoltre ad ogni tabella possiamo associare un alias a cui fare riferimento nel resto della query, attraverso la clausola **AS** (ad es. `FROM ordini AS o`).

La clausola **WHERE** determina le condizioni che saranno applicate ad ogni riga della

tabella di input per decidere se tale riga farà parte del risultato.

La **GROUP BY**, come abbiamo già visto, si usa per raggruppare i risultati sui valori di una o più colonne. La tabella risultato verrà ordinata in base a questi valori. Se aggiungiamo la clausola `WITH ROLLUP`, otterremo delle righe extra con i totali sui dati numerici ad ogni rottura di valore dei campi raggruppati.

La clausola **HAVING** svolge una funzione di selezione delle righe, esattamente come `WHERE`. La differenza è che WHERE viene applicata sulle righe delle tabelle originali, mentre `HAVING` viene applicata sulle righe della tabella risultato dopo i raggruppamenti richiesti dalle `GROUP BY`. In pratica, è utile per effettuare test sui valori restituiti dalle funzioni di colonna.

Con **ORDER BY** indichiamo su quali valori ordinare l'output. Se aggiungiamo DESC i valori saranno ordinati in modo discendente, mentre il default è ascendente. Se non usiamo la ORDER BY l'ordine dei risultati sarà indefinito (a meno che non usiamo GROUP BY).

La clausola **LIMIT**, infine, limita il numero di righe che saranno restituite. Se usata con un solo parametro, indica il numero massimo di righe restituite a partire dalla prima. Se usata con due parametri, il primo indica la riga di partenza (la prima riga equivale a 0), mentre il secondo è il numero massimo di righe.

SELECT DISTINCT

La clausola **DISTINCT** permette di escludere dal risultato le righe duplicate, ovvero quelle identiche ad altre righe. Se ci sono due o più righe di risultato uguali, con `DISTINCT` (o `DISTINCTROW`, che è sinonimo) ne vedremo una sola.

ALL è l'opposto di DISTINCT, cioè estrae tutto, ed è il valore applicato per default.

Ottenere un file della query

INTO OUTFILE si usa per scrivere la tabella risultato su un file di output che verrà creato sul server (non si può usare un nome di file già esistente). Le opzioni relative sono le stesse `FIELDS` e `LINES` già viste per `LOAD DATA INFILE`, di cui `SELECT INTO OUTFILE` è complementare.

Per usare questa clausola **è necessario il privilegio FILE**. Se invece di INTO OUTFILE si usa INTO DUMPFILE, il file di output conterrà una sola riga, senza delimitatori di colonne o di righe e senza escape di caratteri.

JOIN, creare query relazionali

Capita spesso la necessità di selezionare dati prelevandoli fra più tabelle, evidentemente correlate tra loro. A tale scopo si utilizzano le **join**.

Esistono diverse tipologie di join. Fondamentalmente sono tre: la **inner join**, la **outer join** e la **cross join**.

La *cross join* è concettualmente la più semplice, ma raramente trova applicazione in quanto è difficile che abbia un senso logico: si tratta del "prodotto cartesiano" di due tabelle. In pratica, ogni riga della prima tabella viene combinata con tutte le righe della seconda. Ipotizzando di avere una tabella di 5 righe e una di 6, il risultato sarà una tabella di 30 righe.

La *inner join* si effettua andando a cercare righe corrispondenti nelle due tabelle, basandosi sul valore di determinate colonne.

Immaginiamo, in un esempio classico, di avere una *tabella ordini* e una *tabella clienti*. Diciamo che la prima contiene le colonne *idOrdine, idCliente, articolo, quantità* mentre la seconda contiene *idCliente, nome, cognome*.

Evidentemente il campo '**idCliente**' della tabella ordini è una chiave esterna sulla tabella clienti, che ci permette di recuperare i dati anagrafici del cliente che ha effettuato l'ordine (abbiamo limitato al minimo, per semplicità, il numero dei campi contenuti nelle due tabelle). In questo caso quindi potremo fare le join basandoci sulla corrispondenza dei valori dei campi '**idCliente**' nelle due tabelle (naturalmente non è necessario che le colonne abbiano lo stesso nome).

Le righe estratte con una inner join saranno **solo** quelle che hanno il valore di una tabella corrispondente a quello nell'altra tabella.

Le *outer join*, come le inner join, vengono effettuate in base alla corrispondenza di alcuni valori sulle tabelle. La differenza è che, nel caso delle outer join, è possibile estrarre anche le righe di una tabella che **non** hanno corrispondenti nell'altra.

Vediamo alcuni esempi:

SELECT * FROM ordini AS o, clienti AS c WHERE o.idCliente = c.idCliente AND idOrdine > 1000; SELECT * FROM ordini AS o JOIN clienti AS c on o.idCliente = c.idCliente WHERE idOrdine > 1000;

Queste due query sono equivalenti e rappresentano una inner join: estraggono i dati relativi ad ordine e cliente per quegli ordini che hanno un identificativo maggiore di

1000. La prima è una join implicita: infatti non l'abbiamo dichiarata esplicitamente e abbiamo messo la condizione di join nella clausola WHERE. Quando elenchiamo più tabelle nella FROM senza dichiarare esplicitamente la JOIN stiamo facendo una inner join (oppure una cross join se non indichiamo condizioni di join nella WHERE).

Nella seconda, al posto di 'JOIN' avremmo potuto scrivere per esteso 'INNER JOIN'; in alcune vecchie versioni di MySQL ciò è obbligatorio.

SELECT * FROM ordini as o LEFT JOIN clienti as c ON o.idCliente = c.idCliente WHERE idOrdine > 1000;

In questo caso abbiamo effettuato una **left outer join**: la query ottiene gli stessi risultati delle due precedenti, ma in più restituirà le eventuali righe della tabella ordini il cui valore di idCliente non ha corrispondenti sulla tabella clienti. In queste righe della tabella risultato, **i campi dell'altra tabella saranno valorizzati a NULL**. Quindi potremmo eseguire una query che estrae solo le righe della prima tabella senza corrispondente, così:

SELECT * FROM ordini as o LEFT JOIN clienti as c ON o.idCliente = c.idCliente WHERE idOrdine > 1000 AND c.idCliente IS NULL

Le outer join si dividono in **left outer join**, **right outer join** e **full outer join**.

Con le prime otterremo le righe senza corrispondente che si trovano nella tabella di sinistra (cioè quella dichiarata per prima nella query). Le right outer join restituiscono invece le righe della seconda tabella che non hanno corrispondente nella prima. Con le full outer join infine si ottengono le righe senza corrispondente da entrambe le tabelle.

Nella sintassi di MySQL, la parola OUTER è facoltativa: scrivere LEFT JOIN o LEFT OUTER JOIN è equivalente. Allo stesso modo avremmo potuto scrivere RIGHT JOIN o RIGHT OUTER JOIN per una right join.
In MySQL non è invece possibile effettuare le full outer join.

Quando, come nel nostro esempio, le colonne su cui si basa la join hanno lo stesso nome nelle due tabelle, è possibile utilizzare una sintassi abbreviata per effettuare la join: la clausola **USING**. Vediamo la join precedente con questa clausola:

SELECT * FROM ordini LEFT JOIN clienti USING (idCliente) WHERE idOrdine > 1000;

Naturalmente la join può essere basata anche su più colonne. In questo caso elencheremo più condizioni, separate da virgole, nella ON, oppure elencheremo i nomi delle colonne, sempre separati da virgole, nella clausola USING.

C'è una ulteriore possibilità di abbreviare la sintassi, quando la join è basata su **tutte** le

colonne che nelle due tabelle hanno lo stesso nome: si tratta della clausola **NATURAL**.

Anche questa è applicabile al nostro esempio:

SELECT * FROM ordini NATURAL LEFT JOIN clienti WHERE idOrdine > 1000;

Le clausole USING e NATURAL possono essere utilizzate sia con le inner join che con le outer join.

Join fra più tabelle

Finora abbiamo visto esempi di join fra due tabelle, ma è possibile effettuarne anche fra più di due. In questo caso l'operazione sarà logicamente suddivisa in più join, ciascuna delle quali viene effettuata fra due tabelle; il risultato di ogni join diventa una delle due tabelle coinvolte nella join successiva. L'ordine con cui vengono effettuate le diverse join dipende dall'ordine in cui sono elencate le tabelle e (a partire da MySQL 5.0.1) dall'uso di eventuali parentesi:

FROM *t1* JOIN *t2* ON t1.col1 = t2.col2 LEFT JOIN *t3* ON t2.col3 = t3.col3

In questo caso viene effettuata prima la join fra t1 e t2; di seguito, il risultato di questa join viene utilizzato per la left join con t3.

FROM *t1* JOIN (*t2* LEFT JOIN *t3* ON t2.col3 = t3.col3) ON t1.col1 = t2.col2

La presenza delle parentesi fa sì che venga effettuata prima la left join fra t2 e t3, e di seguito il risultato venga utilizzato per la inner join con t1.

Una nota importante: se si mischiano join implicite con join esplicite, a partire da MySQL 5.0.12 queste ultime prendono la precedenza anche in assenza di parentesi. Questo può far sì che alcune query che in precedenza funzionavano possano causare errori, soprattutto se non scritte in maniera ortodossa.

Operatori e funzioni

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);

Gestire date e orari

In una lezione precedente abbiamo visto i cinque tipi di dato che vengono usati in MySQL per gestire le informazioni temporali: `DATE`, `TIME`, `DATETIME`, `TIMESTAMP` e `YEAR`. In generale, comunque, MySQL esprime il formato della data come *AAAA-MM-GG* dove *AAAA* rappresenta l'anno scritto a quattro cifre, mentre *MM* e *GG* simboleggiano, rispettivamente, il mese ed il giorno. Il tempo viene trattato nel formato *OO:MM:SS* in cui i due punti separano, nell'ordine, le ore dai minuti e questi dai secondi.

In questa lezione vedremo, suddivise per tipologie, le principali funzioni che possono essere sfruttate all'interno dei comandi SQL per gestire informazioni temporali. Un elenco completo delle funzioni a disposizione è comunque disponibile sul **sito ufficiale**.

Impostare data e ora

La prima cosa che ci interessa imparare è inserire dati di uno dei tipi sopra elencati. Lo si può fare come segue:

```
insert into utenti (data) values ('2015-03-01');

insert into utenti (orario) values ('17:34');

insert into utenti (dataora) values ('2015-03-02 17:34');
```

In alternativa, per impostare le informazioni temporali attuali, si possono usare le seguenti funzioni:

Funzione	Descrizione
NOW()	restituisce data e ora attuali. Ammette i sinonimi `CURRENT_TIMESTAMP()` e `CURRENT_TIMESTAMP`
CURDATE()	restituisce data attuale. Ammette i sinonimi `CURRENT_DATE()` e `CURRENT_DATE`
CURTIME()	restituisce orario attuale. Ammette i sinonimi `CURRENT_TIME()` e `CURRENT_TIME`

Recuperare le informazioni

Una volta impostate, le informazioni data/ora possono essere lette, in tutto o in parte,

prelevandone solo alcuni elementi. Per queste operazioni, esistono apposite funzioni:

- **giorni, mesi e anni**:
  molte funzioni permettono di recuperare queste informazioni da una data. Le più importanti sono `YEAR()`, `MONTH()` e `DAY()` che, rispettivamente, restituiscono anno, mese e giorno;
- **ore, minuti e secondi**:
  questi dati possono essere estrapolati da informazioni orarie usando le funzioni `HOUR()`, `MINUTE()` e `SECOND()`;
- **giorno nella settimana o nell'anno**:
  può essere utile sapere a che giorno della settimana corrisponde una certa data. Tale informazione può essere ottenuta con `DAYOFWEEK` ed il risultato sarà un numero da 1 a 7, dove 1 corrisponderà alla domenica, 2 a lunedì e così via fino al 7, che corrisponde a sabato. Con la funzione `DAYOFYEAR` si otterrà un numero compreso tra 1 e 366 che indicherà il giorno dell'anno corrispondente alla data presa in considerazione.

Formattare date e orari

L'uso di un formato unico per esprimere date e orari è utile per gestire i dati dal punto di vista del programmatore, ma per mostrare l'output agli utenti è necessario formattare opportunamente questi dati. Lo si può fare con due funzioni: `DATE_FORMAT()` e `TIME_FORMAT()`.

La funzione `DATE_FORMAT()` permette di esprimere il formato di una data usando una stringa costituita da appositi metacaratteri:

Metacarattere	Descrizione
%d	giorno del mese
%m	mese espresso in numero. La variante `%M` esprime il mese in parole
%Y	l'anno su quattro cifre. La variante `%y` esprime l'anno su due cifre

Un elenco completo dei codici da usare nell'espressione del formato è disponibile nella **documentazione ufficiale**.

Ecco alcuni esempi che mostrano vari modi per esprimere il 1° marzo 2015, direttamente sperimentabili nella console del comando *mysql*:

```
> SELECT DATE_FORMAT('2015-03-01','%d/%m/%Y');

01/03/2015

> SELECT DATE_FORMAT('2015-03-01','%d/%m/%y');

01/03/15
```

```
> SELECT DATE_FORMAT('2015-03-01','%d %M %Y');

01 March 2015
```

La funzione `TIME_FORMAT()` svolge lo stesso compito di `DATE_FORMAT()`, ma è riferita agli orari. I codici utilizzabili nell'espressione sono disponibili al medesimo **link** riportato in precedenza, ma si possono usare solo i formati orientati alla gestione dell'orario.

I metacaratteri più comunemente usati sono: `%H` per indicare le ore (da 0 a 24, in alternativa `%h` le mostra da 0 a 12), `%i` per i minuti e `%S` o `%s` per i secondi.

Alcuni esempi:

```
> SELECT TIME_FORMAT('17:25:34','%H-%i');

17-25

> SELECT TIME_FORMAT('17:25:34','%h:%i %p');

05:25 PM

> SELECT TIME_FORMAT('17:25:34','sono le %H e %i minuti');

sono le 17 e 25 minuti
```

Calcoli con date e orari

Spesso può essere utile saper svolgere operazioni con le date e gli orari, sommandoli o sottraendoli tra loro o con periodi di tempo costanti.

Per **sommare** un periodo di tempo ad una data o un orario si possono usare le funzioni `ADDDATE` e `ADDTIME`. La prima calcola la data derivante dalla somma tra il primo argomento, ed un intervallo di tempo espresso come `INTERVAL espressione unità`. Ad esempio:

```
> SELECT ADDDATE('2015-03-01',INTERVAL 5 DAY);

   2015-03-06

> SELECT ADDDATE('2015-03-01', 5);

   2015-03-06
```

Come si vede, per sommare alla data cinque giorni si possono usare due espressioni diverse, `INTERVAL 5 DAY` o semplicemente il numero 5. Esiste una funzione sinonimo che si può usare, `DATE_ADD`, la quale però accetta solo la prima forma.

Discorso analogo vale per `ADDTIME`. Ecco direttamente qualche esempio:

```
> SELECT ADDTIME('17:25','05:05');

   22:30:00
```

```
> SELECT ADDTIME('17:25','00:05:05');

   17:30:05
```

Nel caso di `ADDTIME`, si può indicare direttamente il lasso di tempo da sommare.

Un periodo può essere anche **sottratto da una data**. Esiste per questo motivo `DATE_SUB` il cui funzionamento è speculare a `DATE_ADD`:

```
    > SELECT DATE_SUB('2015-03-01',INTERVAL 5 DAY);

2015-02-24
```

Infine, possiamo calcolare il periodo che intercorre tra due date con `DATEDIFF`. Questa funzione accetta due argomenti ed il risultato sarà un numero positivo se la prima data è successiva alla seconda, negativo altrimenti:

```
> SELECT DATEDIFF('2015-03-01','2015-02-10');

   19

> SELECT DATEDIFF('2015-01-01','2015-02-10');

   -40
```

Ricerche full-text

MySQL offre l'opportunità di effettuare particolari ricerche, dette **full-text**, su campi di testo anche piuttosto estesi. La caratteristica speciale di queste ricerche è che non si limitano a rintracciare le occorrenze di una parola o di una frase – al pari di quello che potrebbe fare l'operatore `LIKE` – ma individuano le righe in cui il campo o i campi sottoposti alla ricerca mostrano un'attinenza particolare con il pattern indicato.

È una caratteristica che assomiglia poco ad un comune filtro e molto di più a quelle forme di valutazione utili nei motori di ricerca o nei CMS (Content Management System), usate in moltissimi blog.

Gli indici FULLTEXT e la funzione MATCH

Tali ricerche possono essere eseguite su uno o più campi ma richiedono la presenza di un **`FULLTEXT INDEX`** che li coinvolga tutti.

La definizione seguente crea un tabella con un indice `FULLTEXT` applicato ad un campo solo:

```
CREATE TABLE Blog

(

id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,

title VARCHAR(50),

body TEXT,

FULLTEXT (body)

)
```

Come si vede, è sufficiente indicare la parola chiave `FULLTEXT` ed elencare, tra parentesi tonde, i campi coinvolti.

È bene comunque ricordare che tali indici possono essere creati solo su campi di testo (`VARCHAR`, `TEXT`, `CHAR`) e con gli Storage Engine MyISAM ed InnoDB. In particolare, su quest'ultimo motore sono permessi solo a partire dalla versione 5.6 del DBMS.

La **ricerca** potrà essere effettuata usando il costrutto `MATCH ... AGAINST`:

```
SELECT * FROM Blog WHERE MATCH (body) AGAINST ('database');
```

Questa query effettuerà una ricerca sul campo `body` della tabella `Blog` usando il termine `database` come pattern.

Nella clausola `MATCH` possono essere inclusi più campi, ma è fondamentale che tutti facciano parte dell'indice `FULLTEXT` creato. È altrettanto essenziale che, se l'indice include più campi, tutti vengano nominati nella clausola `MATCH`.

La rilevanza

Il risultato della funzione `MATCH` è un valore in virgola mobile, non negativo, che rappresenta la rilevanza di ogni record recuperato alla luce della ricerca effettuata. Ciò rappresenta la principale differenza rispetto ai filtri più comuni e meno complessi: il risultato non ci dice solot se una stringa appartiene al campo o meno, ma cerca anche di capire se esiste un'attinenza tra il pattern ricercato ed i contenuti.

Per vedere i valori che la rilevanza può assumere, si esegua una query di questo tipo:

```
> select id,MATCH(body) AGAINST ('archivio fisico') from Blog;
   1    0
   3    0.6695559024810791
   2    0.6801062822341919
   4    0
   5    0
```

La tabella `Blog` contiene alcuni post campione che trattano di database. Abbiamo cercato nel campo `body` il testo "archivio fisico" e sono stati segnalati, in base alla rilevanza, due record:

- il record con `id` pari a 2, contenente l'espressione "...l'archivio fisico...";
- il record con `id` pari a 3, contenente la stringa "...l'archivio a livello fisico...".

Entrambi vengono ritenuti interessanti ai fini della ricerca pertanto i livelli di rilevanza calcolati sono significativi. Il voto ottenuto dal post con `id` pari a 2 è leggermente superiore in quanto contiene un'espressione più calzante con il pattern.

Per ottenere solo i post rilevanti possiamo eseguire la query successiva che include l'uso di `MATCH` all'interno della clausola `WHERE`.

```
> select id, MATCH(body) AGAINST ('archivio fisico') from Blog where MATCH(body)
AGAINST ('archivio fisico');

  2 0.6801062822341919

  3 0.6695559024810791
```

Nonostante il costrutto `MATCH...AGAINST` appaia due volte volte, l'ottimizzatore del DBMS riesce a percepire che si tratta della medesima espressione e la calcola una sola volta.

Inoltre, il fatto che i risultati vengano forniti in ordine decrescente di rilevanza non è casuale: questo è infatti il comportamento di default.

Modificatori di ricerca

All'interno della clausola `AGAINST`, subito dopo il pattern di ricerca, possono essere inclusi dei modificatori.

Quello di default è **IN NATURAL LANGUAGE MODE**. In pratica, scrivere `AGAINST ('archivio fisico')` oppure `AGAINST ('archivio fisico' IN NATURAL LANGUAGE MODE)` produce il medesimo effetto.

Questo modificatore effettua una ricerca considerando che il pattern proposto viene scritto in linguaggio naturale, quello spontaneo per l'essere umano, implementando quindi un'attività da tipico motore di ricerca.

In casi in cui il pattern di ricerca sia troppo corto, la rilevanza ottenuta potrebbe essere bassa. Si può pertanto richiedere una ricerca con "espansione delle query" utilizzando il modificatore **`WITH QUERY EXPANSION`** o `IN NATURAL LANGUAGE MODE WITH QUERY EXPANSION`.

Questa modalità di ricerca effettuerà il procedimento due volte: prima cerca il pattern così come fornito; la seconda volta, invece, associa al pattern i contenuti più rilevanti individuati nella prima ricerca. Ciò, tipicamente, produrrà un maggior numero di documenti rilevanti riscontrati.

Un'ultima tipologia di ricerca è detta **booleana** e introdotta dal modificatore **IN BOOLEAN MODE**.

Utilizza simboli + e – per indicare, all'interno del pattern, se la parola deve essere contenuta (+) o no (–).

Ad esempio, la clausola `AGAINST('+cane -gatto' IN BOOLEAN MODE)` specifica che il pattern è inteso per rilevare i post che contengono il termine 'cane' ma non 'gatto'. A

livello di logica booleana, il simbolo + corrisponde ad un AND, il - ad un NOT, mentre l'assenza di simbolo corrisponde ad un OR.

Il motore InnoDB accetta tali simboli solo se precedono il termine, ma si tratta di una restrizione in quanto, secondo le regole generali di MySQL, tali simboli possono anche seguire la parola cui si riferiscono.

Funzioni di aggregazione e UNION

Oltre a leggere i valori "riga per riga", in molti casi è utile sottoporre le tabelle a valutazioni che li coinvolgono in gruppo. A questo scopo si utilizzano le cosiddette **funzioni di aggregazione**, tre la quali troviamo:

- `COUNT`: restituisce il numero di elementi;
- `COUNT(DISTINCT)`: variante di `COUNT` che considera i valori duplicati solo una volta;
- `MAX`: restituisce il valore massimo dell'insieme;
- `MIN`: restituisce il valore minimo dell'insieme;
- `AVG`: restituisce il valore medio dell'insieme;
- `SUM`: somma i valori.

Esistono altre funzioni di questo genere. Se ne può trovare un elenco esaustivo nell'**apposita pagina della documentazione ufficiale**. Esiste anche un nutrito gruppo di funzioni a carattere statistico.

Funzioni di aggregazione: esempi

Prendiamo ad esempio una tabella che, nel client mysql, si mostra come in figura:

```
mysql> select * from spese;
+---------+-------------+
| importo | descrizione |
+---------+-------------+
|      23 | alimentari  |
|      28 | cartoleria  |
|     100 | alimentari  |
|       4 | cartoleria  |
|      65 | alimentari  |
|       4 | cartoleria  |
|      65 | alimentari  |
|      88 | strumenti   |
|     150 | strumenti   |
+---------+-------------+
9 rows in set (0.00 sec)
```

Il campo *importo* rappresenta simbolicamente un costo sostenuto mentre *descrizione*

ne rappresenta la causale.

Gli esempi qui di seguito riportati utilizzano le funzioni di aggregazione sul predetto insieme, e mostrano i risultati ottenuti:

```
> SELECT COUNT(importo) FROM spese;
   9
> SELECT COUNT(DISTINCT importo) FROM spese;
   7
> SELECT SUM(importo) FROM spese;
   527
> SELECT AVG(importo) FROM spese;
   58.5556
> SELECT MIN(importo) FROM spese;
   4
> SELECT MAX(importo) FROM spese;
   150
```

Si noti che i valori 4 e 65 sono ripetuti nell'insieme: questo giustifica la differenza di risultato tra `COUNT` e `COUNT(DISTINCT)`.

È importante notare il rapporto che le funzioni di aggregazione hanno con operatori molto importanti di SQL, quali `WHERE` e `GROUP BY`. Il primo influenza la quantità di elementi che verranno sottoposti alla funzione di interazione. In una query come la seguente verrà prima eseguito il filtro sulle righe e, solo successivamente, sarà applicata la funzione di aggregazione sui record risultanti:

```
> SELECT SUM(importo) FROM spese WHERE descrizione='cartoleria';
   36
```

Per quanto riguarda l'uso del costrutto `GROUP BY` con funzioni di aggregazione, partiamo da un esempio che mostra un risultato ingannevole. Supponiamo di svolgere la seguente query:

```
> SELECT descrizione, MIN(importo) FROM spese;
   alimentari | 4
```

Il valore minimo individuato è 4 (e ciò è corretto) ma l'associazione sulla stessa riga del termine *alimentari* potrebbe – ingannevolmente – indurre a pensare che questa sia la descrizione associata al valore. Controllando la figura precedente si vede, invece, che

la descrizione del valore 4 è *cartoleria*. Cos'è successo quindi? La query ha prodotto il valore minimo corrispondente all'invocazione di `MIN`, e non sapendo quale elemento del campo descrizione associare ha prelevato il primo valore che ha trovato.

Per avere dei risultati corretti e che non traggano in inganno come nell'esempio precedente è necessario che la proiezione, ossia l'elenco dei campi indicati dopo il `SELECT`, contenga funzioni di aggregazione ed eventuali campi semplici solo se elencati in una clausola `GROUP BY`.

La seguente query:

```
> SELECT descrizione, MIN(importo) FROM spese GROUP BY descrizione;
   alimentari  |           23
   cartoleria  |           4
   strumenti   |           88
```

mostra un risultato veritiero. Saranno raggruppati i record in base alla descrizione e per ogni gruppo verrà individuato il minimo.

La clausola UNION

La clausola `UNION` viene usata per unire i risultati di due query.

Affinchè il tutto possa funzionare è necessario che le proiezioni delle due `SELECT` coinvolte siano composte dallo stesso numero di campi, e che i tipi di dato degli stessi siano compatibili con quelli corrispondenti nell'altra query.

Ad esempio, immaginiamo di avere due tabelle con la stessa struttura, di nome *dati* e *dati_archivio*, dove quest'ultima, come spesso può capitare, contiene vecchi record un tempo inseriti nella prima ed ormai conservati solo per valore storico:

```
SELECT descrizione, importo, data_operazione FROM dati
UNION
SELECT descrizione, importo, data_operazione FROM dati_archiviati
```

Il risultato sarà un unico set di record reperiti da entrambe le tabelle.

Al posto di `UNION` si può usare **`UNION ALL`**, che mostrerà anche i valori duplicati, mentre il comportamento di default di `UNION` non lo fa.

Le subquery

All'epoca della loro introduzione con la versione 4.1 di MySQL, le subquery sono state accolte dagli sviluppatori come una delle innovazioni più attese. A lungo infatti è stato sottolineato come la mancanza di alcune funzionalità penalizzasse notevolmente MySQL nel confronto con altri RDBMS, e l'assenza delle subquery era sicuramente fra quelle che più si notavano.

Una subquery non è altro che una `SELECT` all'interno di un'altra istruzione. Le subquery possono essere nidificate anche a profondità notevoli.

Abbiamo già visto che ogni SELECT restituisce logicamente una tabella formata da righe e colonne. Nel caso delle subquery è necessario fare una distinzione: esse infatti possono restituire un **valore singolo (scalare)**, una **singola riga**, una **singola colonna**, oppure una normale tabella. Le diverse tipologie di subquery possono trovare posto in diversi punti dell'istruzione.

La subquery come operando scalare.

Il caso più semplice di subquery è quella che restituisce un singolo valore. La si può usare in qualsiasi punto sia possibile utilizzare un valore di colonna. L'uso più frequente lo troviamo come operatore di confronto:

SELECT *colonna1* FROM t1
WHERE *colonna1* = (SELECT MAX(*colonna2*) FROM t2);

Questa query estrae i valori di colonna1 nella tabella t1 che sono uguali al valore massimo di colonna2 nella tabella t2.

Subquery che restituiscono colonne.

Quando una subquery restituisce una colonna, può essere usata per fare confronti attraverso gli operatori **ANY**, **SOME**, **IN** e **ALL**:

SELECT s1 FROM t1 WHERE s1 > ANY (SELECT s1 FROM t2); SELECT s1 FROM t1 WHERE s1 IN (SELECT s1 FROM t2);

La prima query significa "seleziona da t1 i valori di s1 che sono maggiori di **almeno 1** dei valori di s1 su t2". La seconda invece seleziona i valori di s1 che sono **uguali ad almeno 1** dei valori di s1 su t2. "IN" è sinonimo di "= ANY". "SOME" è invece equivalente in tutto e per tutto ad "ANY".

SELECT s1 FROM t1 WHERE s1 > ALL (SELECT s1 FROM t2);

Il significato qui è "seleziona da t1 i valori di s1 che sono maggiori di **tutti** i valori di s1 su t2".
La clausola "NOT IN" equivale a "<> ALL".

Subquery che restituiscono righe.

Quando una subquery restituisce una singola riga, può essere usata per fare confronti attraverso i **costruttori di righe**:

SELECT *colonna1*,*colonna2* FROM t1
WHERE (*colonna1*,*colonna2*) IN
(SELECT *colonna1*,*colonna2* FROM t2);

Questa query estrae le righe di t1 in cui i valori di colonna1 e colonna2 sono ripetuti in una riga di t2. L'espressione "(colonna1,colonna2)" è, appunto, un costruttore di riga, che poteva essere espresso anche come "ROW(colonna1,colonna2)".

Subquery correlate.

Le subquery correlate sono quelle che contengono un riferimento ad una tabella che fa parte della query esterna:

SELECT * FROM t1 WHERE *colonna1* = ANY
(SELECT *colonna1* FROM t2 WHERE t2.*colonna2* = t1.*colonna2*);

In questa subquery, la clausola WHERE contiene un riferimento alla tabella t1, che tuttavia non è nominata nella clausola FROM della subquery stessa: la troviamo infatti nella FROM della query esterna.

Query di questo tipo richiedono che la subquery venga rieseguita ad ogni riga estratta dalla query esterna, e di conseguenza non sono molto performanti. Meglio quindi evitarle quando possibile: spesso infatti una subquery correlata è trasformabile in una join.

Le subquery correlate vengono usate a volte con le clausole **EXISTS** e **NOT EXISTS**; la clausola EXISTS è vera quando la subquery restituisce almeno una riga, mentre è falsa nel caso opposto. Ovviamente NOT EXISTS funziona al contrario.

Un esempio:

SELECT DISTINCT tipoNegozio FROM negozi
WHERE EXISTS (SELECT * FROM negozi_citta
WHERE negozi_citta.tipoNegozio = negozi.tipoNegozio);

Ipotizzando che la tabella negozi_citta contenga i tipi di negozio presenti nelle varie

città, questa query estrae i tipi di negozio che sono presenti in almeno una città.

Subquery nella FROM.

E’ possibile utilizzare una subquery anche nella clausola FROM, con questa sintassi:

SELECT ... FROM (subquery) [AS] *nome* ...

Notate che è obbligatorio assegnare un nome alla subquery, per poterla referenziare nelle altre parti della query. Ad esempio:

```
SELECT sq.*, t2.c1
FROM (SELECT c1, c2, c3 FROM t1 WHERE c1 > 5) AS sq
LEFT JOIN t2 ON sq.c1 = t2.c1;
```

In questo caso l’output della subquery viene chiamato “sq” ed il riferimento è usato sia nella SELECT sia nella condizione di join.

Transazioni e lock

Una delle classiche problematiche che un DBMS deve gestire è l'accesso simultaneo ai dati da parte di diversi utenti, sia in lettura che in aggiornamento.

Una situazione tipica di questo genere è il caso in cui, ad esempio, due utenti leggono lo stesso dato con l'intenzione di aggiornarlo: evidentemente uno dei due lo farà per primo, e a quel punto il secondo utente, quando tenterà a sua volta un aggiornamento, troverà una situazione variata rispetto al momento in cui aveva letto i dati, col rischio di creare situazioni incongruenti.

Un'altra classica situazione che pone dei problemi è quella in cui un'applicazione deve effettuare più aggiornamenti logicamente collegati fra loro, tanto da richiedere che tutti gli aggiornamenti siano annullati qualora uno solo di essi dovesse fallire.

Le soluzioni per questi problemi sono, nella forma più semplice, i lock sulle tabelle, e in quella più avanzata le transazioni. Queste ultime sono disponibili su tabelle InnoDB – lo Storage Engine di default a partire dalla versione 5.5 del DBMS – oltre che su NDB, un meccanismo dedicato ai Cluster che vedremo in seguito. Al contrario, non possono essere utilizzate su tabelle MyISAM.

Cominciamo con l'analisi dei **lock**, che possiamo considerare dei vincoli di "uso esclusivo" che un utente può ottenere su determinate tabelle per il tempo necessario a svolgere le operazioni che gli sono necessarie. Con i lock si possono simulare (parzialmente) transazioni, o in alcuni casi semplicemente velocizzare le operazioni di scrittura, qualora vi siano grosse moli di dati da inserire. L'uso dei lock è consigliato solo con le tabelle di tipo MyISAM, che non supportano le transazioni.

Un **lock** può essere richiesto in lettura o in scrittura: nel primo caso l'utente ha la garanzia che nessuno farà aggiornamenti sulla tabella bloccata fino a quando non sarà rilasciato il lock, ma agli altri utenti viene comunque lasciata la possibilità di leggere sulla stessa tabella. In questo caso però nemmeno l'utente che ha ottenuto il lock può fare aggiornamenti.

Il lock in scrittura invece impedisce agli altri utenti qualsiasi tipo di accesso alla tabella,

e consente all'utente che l'ha ottenuto operazioni di lettura e scrittura.

Vediamo la sintassi delle operazioni di lock, ricordando che esse richiedono il privilegio LOCK TABLES nonché quello di SELECT sulle tabelle interessate:

```
LOCK TABLES
    tabella [AS alias] {READ [LOCAL] | [LOW_PRIORITY] WRITE}
    [, tabella [AS alias] {READ [LOCAL] | [LOW_PRIORITY] WRITE}] ...
```

Innanzitutto notiamo che è possibile effettuare il lock su più tabelle con un'unica istruzione LOCK TABLES; in realtà, più che di una possibilità si tratta di un obbligo.

Infatti ogni istruzione LOCK TABLES causa il rilascio dei lock ottenuti in precedenza: di conseguenza, se avete bisogno di ottenere lock su più tabelle, siete obbligati a farlo con un'unica istruzione.

Ad ogni tabella di cui chiediamo il lock è possibile associare un alias, esattamente come nelle query: anche in questo caso siamo vincolati ad usare questo sistema qualora le query che ci accingiamo ad effettuare utilizzino gli alias. In pratica, dopo avere ottenuto un lock, le nostre query possono utilizzare **solo** le tabelle su cui abbiamo i lock: non è possibile quindi, in presenza di lock attivi, accedere ad altre tabelle; inoltre, a queste tabelle dovremo accedere utilizzando **gli stessi alias** definiti in fase di lock. Qualora una tabella sia presente più volte in una query, avremo evidentemente bisogno di più di un alias: di conseguenza dovremo ottenere un lock **per ogni alias**, sebbene la tabella sia la stessa.

La clausola LOCAL associata ad un READ lock consente ad altri utenti di effettuare inserimenti che non vadano in conflitto con le nostre letture. La clausola LOW_PRIORITY associata ad un WRITE lock fa sì che la richiesta dia la precedenza alle richieste di lock in lettura (normalmente invece un lock in scrittura ha priorità più alta).

I lock ottenuti vengono rilasciati con l'istruzione:

UNLOCK TABLES

In realtà abbiamo già visto che anche una nuova richiesta di lock causa il rilascio dei precedenti; inoltre i lock vengono rilasciati automaticamente alla chiusura della connessione, qualora non sia stato fatto esplicitamente.

Le transazioni

Passiamo ora alle transazioni, con particolare riferimento alle tabelle InnoDB.

L'uso delle transazioni permette di "consolidare" le modifiche alla base dati solo in un

momento ben preciso: dal momento in cui avviamo una transazione, gli aggiornamenti rimangono sospesi (e invisibili ad altri utenti) fino a quando non li confermiamo (**commit**); in alternativa alla conferma è possibile annullarli (**rollback**).

Innanzitutto va segnalato che MySQL gira di default in **autocommit mode**: questo significa che tutti gli aggiornamenti vengono automaticamente consolidati nel momento in cui sono eseguiti. Se siamo in autocommit, per iniziare una transazione dobbiamo usare l'istruzione **START TRANSACTION**; da questo punto in poi tutti gli aggiornamenti rimarranno sospesi. La transazione può essere chiusa con l'istruzione **COMMIT**, che consolida le modifiche, oppure con **ROLLBACK**, che annulla tutti gli aggiornamenti effettuati nel corso della transazione. Possiamo utilizzare anche **COMMIT AND CHAIN** o **ROLLBACK AND CHAIN**, che provocano l'immediata apertura di una nuova transazione, oppure **COMMIT RELEASE** o **ROLLBACK RELEASE**, che oltre a chiudere la transazione chiudono anche la connessione al server.

Con l'istruzione **SET AUTOCOMMIT=0** possiamo disattivare l'autocommit: in questo caso non è più necessario avviare le transazioni con START TRANSACTION, e tutti gli aggiornamenti rimarranno sospesi fino all'uso di COMMIT o ROLLBACK.
All'interno di una transazione è anche possibile stabilire dei **savepoint**, cioè degli stati intermedi ai quali possiamo ritornare con una ROLLBACK, invece di annullare interamente la transazione.

Vediamo un esempio:

START TRANSACTION
...istruzioni di aggiornamento (1)...
SAVEPOINT sp1;
...istruzioni di aggiornamento (2)...
ROLLBACK TO SAVEPOINT sp1;
...istruzioni di aggiornamento (3)...
COMMIT

In questo caso, dopo avere avviato la transazione abbiamo eseguito un primo blocco di aggiornamenti, seguito dalla creazione del savepoint col nome 'sp1'; in seguito abbiamo eseguito un secondo blocco di aggiornamenti; l'istruzione *ROLLBACK TO SAVEPOINT sp1* fa sì che "ritorniamo" alla situazione esistente quando abbiamo creato il savepoint: in pratica solo il secondo blocco di aggiornamenti viene annullato, e la transazione rimane aperta; una semplice ROLLBACK invece avrebbe annullato tutto e chiuso la transazione.

La COMMIT effettuata dopo il terzo blocco fa sì che vengano consolidati gli aggiornamenti effettuati nel primo e nel terzo blocco.

È bene ricordare che un utilizzo corretto delle transazioni è possibile solo utilizzando lo stesso tipo di tabelle all'interno di ogni transazione. È altamente sconsigliato ovviamente l'utilizzo di tabelle MyISAM nelle transazioni, in quanto su di esse non è possibile effettuare il ROLLBACK e gli aggiornamenti relativi sono immediatamente effettivi: in caso di ROLLBACK quindi si genererebbero proprio quelle inconsistenze che l'uso delle transazioni mira ad evitare.

Ricordiamo anche che alcuni tipi di operazioni non sono annullabili: in generale tutte quelle che creano, eliminano, o alterano la struttura di database e tabelle. È bene quindi evitare di includere in una transazione tali operazioni, che fra l'altro, nella maggior parte dei casi, causano una implicita COMMIT.

In alcuni casi è utile utilizzare due clausole particolari quando si effettua una SELECT:

SELECT ……. FOR UPDATE; SELECT ……. LOCK IN SHARE MODE;

La clausola FOR UPDATE stabilisce un lock su tutte le righe lette che impedirà ad altri utenti di leggere le stesse righe fino al termine della nostra transazione; evidentemente si utilizza quando leggiamo un dato con l'intenzione di aggiornarlo. La clausola LOCK IN SHARE MODE invece stabilisce un lock che impedisce solo gli aggiornamenti, garantendoci che il contenuto della riga rimarrà invariato per la durata della nostra transazione.

Isolation level

Un aspetto importante relativamente alle transazioni è il livello di isolamento al quale vengono effettuate. I livelli possibili sono quattro, e li elenchiamo in ordine crescente:

- **READ UNCOMMITTED**: a questo livello sono visibili gli aggiornamenti effettuati da altri utenti **anche se non consolidati**: è un comportamento non propriamente transazionale, che può dare ovviamente seri problemi di consistenza dei dati; va utilizzato solo quando non abbiamo preoccupazioni di questo tipo e abbiamo bisogno di velocizzare le letture
- **READ COMMITTED**: a questo livello gli aggiornamenti diventano visibili solo dopo il consolidamento
- **REPETEABLE READ**: in questo caso perchè un aggiornamento diventi visibile deve essere non solo consolidato, ma anche la transazione che legge deve essere terminata; in pratica, la stessa lettura ripetuta all'interno di una transazione darà sempre lo stesso risultato; è la modalità di default
- **SERIALIZABLE**: come nel caso precedente, ma in più, la semplice lettura di un dato provoca il blocco degli aggiornamenti fino al termine della transazione; in sostanza è come se ogni SELECT venisse effettuata con la clausola LOCK IN SHARE MODE

Il livello di isolamento utilizzato può essere determinato dall'opzione di avvio del server –transaction-isolation (vedere lez. 4, facendo attenzione alla diversa sintassi delle opzioni); per sapere qual è il livello in uso possiamo usare l'istruzione **SELECT @@tx_isolation**; inoltre possiamo modificarlo con la seguente istruzione:

SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL { READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE }

Se omettiamo le clausole GLOBAL e SESSION la modifica è valida solo per la transazione successiva; con SESSION impostiamo il valore per l'intera connessione, mentre con GLOBAL modifichiamo il valore per il server: tale valore verrà quindi adottato su tutte le connessioni aperte successivamente (**non** su quelle già aperte); in quest'ultimo caso è necessario il privilegio SUPER.

Per concludere, abbiamo già detto che l'uso di LOCK TABLES è consigliato solo su storage engine non transazionali. Ricordiamo anche che se lo usiamo su storage engine transazionali un'istruzione LOCK TABLES causerà una implicita COMMIT di una eventuale transazione aperta. All'opposto, avviare una transazione provoca l'implicita esecuzione di una UNLOCK TABLES.

Le viste (views)

Le viste sono comunemente considerate un modo per mostrare i dati di un database con una struttura diversa da quella che hanno effettivamente sulla base dati.

Un uso possibile delle viste è quello di concedere ad un utente l'accesso ad una tabella mostrandogli solo alcune colonne: tali colonne saranno inserite nella vista, sulla quale l'utente avrà i permessi di accesso, mentre gli saranno negati quelli sulla tabella sottostante.

Altre possibili applicazioni riguardano la possibilità di leggere dati da più tabelle contemporaneamente attraverso JOIN o UNION, oppure di comprendere dati che non sono fisicamente presenti sulla tabella in quanto calcolati a partire da altri dati.

Le viste non possono avere lo stesso nome di una tabella facente parte dello stesso database. Ecco la sintassi da usare per la creazione di una vista:

```
CREATE
    [OR REPLACE]
    [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
    [DEFINER = { utente | CURRENT_USER }]
    [SQL SECURITY { DEFINER | INVOKER }]
    VIEW nome [(lista_colonne)]
    AS istruzione_select
    [WITH [CASCADED | LOCAL] CHECK OPTION]
```

La clausola **OR REPLACE** consente di sostituire una vista con lo stesso nome eventualmente già esistente. Le clausole **DEFINER** e **SQL SECURITY**, invece, le approfondiremo meglio nelle lezioni successive.

La lista delle colonne, opzionale, viene usata per specificare i nomi delle colonne dalle quali è composta la vista. È possibile ometterla e adottare come nomi di colonne quelli restituiti dalla SELECT.

L'istruzione **SELECT** è quella che definisce quali dati sono contenuti nella vista. Essa può essere anche molto complessa, fare riferimento a più tabelle, contenere subquery, ma è sottoposta ad alcuni limiti: non può utilizzare variabili, non può contenere subquery nella FROM, non può fare riferimento a tabelle temporanee ma solo a tabelle già esistenti. Non possono esistere viste temporanee.

Una caratteristica essenziale delle viste è quella di essere o meno **aggiornabili**: una vista aggiornabile infatti consente di modificare i dati della tabella sottostante, cosa che non è possibile in caso contrario. Tuttavia una vista, per essere aggiornabile, deve soddisfare determinate condizioni: in pratica, deve esistere una relazione uno a uno fra

le righe della vista e quelle della tabella sottostante.

Quindi la SELECT che genera la vista non può contenere funzioni aggregate, DISTINCT, GROUP BY, HAVING, UNION, subquery nella lista delle colonne selezionate, JOIN (con qualche eccezione, vedere oltre); inoltre non può avere un'altra vista non aggiornabile nella clausola FROM, o una subquery nella WHERE che fa riferimento a una tabella nella FROM.

Le viste possono essere generate con due algoritmi: MERGE o TEMPTABLE. Con quest'ultimo, la SELECT relativa alla vista viene utilizzata per creare una tabella temporanea, sulla quale viene poi eseguita l'istruzione richiesta; con MERGE invece la SELECT viene "mescolata" con l'istruzione che richiama la vista stessa. Una vista non è mai aggiornabile se creata con TEMPTABLE.

La clausola **ALGORITHM** che abbiamo visto nella CREATE può essere usata per specificare quale algoritmo usare: tuttavia per poter usare MERGE è necessario che la vista soddisfi alcuni dei criteri che abbiamo visto in precedenza per le viste aggiornabili. Se utilizziamo TEMPTABLE per una vista che potrebbe essere costruita con MERGE, avremo la conseguenza di non poterla usare per aggiornamenti, ma il vantaggio di una maggior velocità nel rilascio della tabella sottostante. Se non indichiamo la clausola ALGORITHM, il default sarà UNDEFINED.

Quando una vista è aggiornabile possiamo anche utilizzarla per **inserire** dati, purché contenga tutte le colonne della tabella che non hanno un valore di default, e nessuna colonna derivata da espressioni.

In alcuni casi, come eccezione a quanto detto prima, può essere aggiornabile una vista che contiene una INNER JOIN, a patto che gli aggiornamenti vengano fatti sui campi di una sola tabella. Sulla stessa vista è possibile fare delle INSERT, anche in questo caso su una sola tabella.

La clausola **WITH CHECK OPTION** si usa nei casi di viste aggiornabili, per impedire inserimenti o aggiornamenti di righe della tabella che non soddisfano la condizione prevista nella clausola WHERE della vista. Qualora la vista faccia riferimento ad altre viste, l'ulteriore specificazione di **CASCADED** fa sì che il controllo venga propagato alle condizioni delle viste interne; se invece viene specificato **LOCAL** il controllo viene effettuato solo al primo livello, cioè sulla vista attuale. Il default è CASCADED.

Per modificare una vista si utilizza una istruzione ALTER che è praticamente identica alla CREATE OR REPLACE:

```
ALTER
    [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
```

```
[DEFINER = { utente | CURRENT_USER }]
[SQL SECURITY { DEFINER | INVOKER }]
VIEW nome [(lista_colonne)]
AS istruzione_select
[WITH [CASCADED | LOCAL] CHECK OPTION]
```

L'eliminazione della vista si ottiene con l'istruzione DROP VIEW:

DROP VIEW [IF EXISTS] *nome*

Infine è possibile visualizzare la definizione della vista attraverso la seguente istruzione:

SHOW CREATE VIEW *nome*

I permessi

Per creare una vista occorre il privilegio CREATE VIEW (nonchè il privilegio SUPER se si vuole definire un altro utente come DEFINER); per eliminarla il privilegio DROP. Entrambi i privilegi sono necessari per modificare una vista (sia con ALTER VIEW che con CREATE OR REPLACE).

Infine il privilegio SHOW_VIEW consente di visualizzare la definizione della vista. Ovviamente sono poi necessari i permessi richiesti per accedere ai dati selezionati; tali permessi sono richiesti al creatore o all'esecutore della vista a seconda del parametro SQL SECURITY.

Stored Procedures e Stored Functions

Le **stored procedures** (procedure memorizzate) sono un'altra delle caratteristiche la cui assenza è stata a lungo sottolineata dai detrattori di MySQL: con la versione 5.0 si è finalmente posto rimedio a questa assenza.

Una stored procedure è un insieme di istruzioni SQL che vengono memorizzate nel server con un nome che le identifica; tale nome consente in seguito di rieseguire l'insieme di istruzioni facendo semplicemente riferimento ad esso. Vediamo come creare una stored procedure:

CREATE PROCEDURE *nome* ([*parametro*[,...]])
[SQL SECURITY { DEFINER | INVOKER }] *corpo*
//parametri:
[IN | OUT | INOUT] *nomeParametro tipo*

Come abbiamo detto, ogni procedura è identificata da un nome. Inoltre la procedura è attribuita ad uno specifico database (a partire da MySQL 5.0.1), esattamente come una tabella. Di conseguenza la procedura viene assegnata ad un database al momento della creazione, ed i nomi referenziati al suo interno si riferiranno allo stesso database, a meno che non siano qualificati con un nome di database specifico. In fase di creazione, quindi, se indichiamo il nome della procedura senza specificare il database questa sarà assegnata al db attualmente in uso.

Ogni procedura può avere uno o più parametri, ciascuno dei quali è formato da un nome, un tipo di dato e l'indicazione se trattasi di parametro di input, di output o entrambi. Se manca l'indicazione, il parametro è considerato di input.

La clausola SQL SECURITY stabilisce se, al momento dell'esecuzione, la procedura utilizzerà i permessi dell'utente che la sta eseguendo o quelli dell'utente che l'ha creata (il default è DEFINER).

Vediamo adesso un esempio concreto di stored procedure:

CREATE PROCEDURE *nomeProc* (IN *param1* INT, OUT *param2* INT)

SELECT COUNT(*) INTO *param2* FROM *tabella*
WHERE *campo1* = *param1*;

Questa istruzione crea una procedura chiamata 'nomeProc' nel database in uso; la procedura usa un parametro in input e uno in output, entrambi interi, ed effettua il conteggio delle righe in tabella in cui il valore di campo1 corrisponde al primo parametro; il risultato della query viene memorizzato nel secondo parametro attraverso la clausola INTO.

Ecco come chiamare la procedura e visualizzare il risultato:

CALL *nomeProc* (5, @a);
SELECT @a;

Con l'istruzione CALL effettuiamo la chiamata della procedura (immaginando che il database attualmente in uso sia lo stesso a cui la procedura è associata), passando il valore 5 come parametro di input e la variabile @a come parametro di output, nel quale verrà memorizzato il risultato. La SELECT successiva visualizza il valore di tale variabile dopo l'esecuzione.

Nell'esempio appena visto la stored procedure conteneva una semplice SELECT; è possibile invece creare procedure che contengono sintassi complesse comprendenti più istruzioni: in pratica, dei veri e propri script, con la possibilità di controllare il flusso attraverso vari costrutti (IF, CASE, LOOP, WHILE, REPEAT, LEAVE, ITERATE). Inoltre è possibile utilizzare i **cursori** per gestire i resultset.

Ecco un esempio di procedura di una certa complessità:

```
DELIMITER //
CREATE PROCEDURE procedura1 (param1 INT, param2 CHAR(3),
        OUT param3 INT)
BEGIN
        DECLARE finito INT default 0;
        DECLARE a INT;
        DECLARE b CHAR(50);
        DECLARE cur1 CURSOR FOR SELECT id,nome
                FROM clienti WHERE cat = param2;
        DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'
                SET finito = 1;
        OPEN cur1;
        SET param3 = 0;
        FETCH cur1 INTO a,b;
        ciclo: WHILE NOT finito DO
                IF param3 < param1 THEN
                        SET param3 = param3 + 1;
                        FETCH cur1 INTO a,b;
                ELSE
```

```
                        LEAVE ciclo;
                END IF;
        END WHILE ciclo;
END; //
DELIMITER ;
```

Analizziamo questo codice.

Per prima cosa notiamo il comando **DELIMITER**, che serve al client mysql per modificare il normale delimitatore delle istruzioni, che sarebbe il punto e virgola. Infatti, siccome la stored procedure contiene più istruzioni, al suo interno il punto e virgola viene utilizzato più volte. Di conseguenza, se vogliamo riuscire a memorizzare la procedura, dobbiamo comunicare al server che il delimitatore è un altro; in caso contrario, al primo punto e virgola penserebbe che la nostra CREATE sia terminata. Decidiamo quindi che il nuovo delimitatore sia un doppio slash.

Dichiariamo la procedura col nome *procedura1* e tre parametri, di cui i primi due di input e l'ultimo di output. Il codice da eseguire deve essere racchiuso tra le clausole **BEGIN** ed **END**; all'interno troviamo tre istruzioni **DECLARE**che dichiarano altrettante **variabili**, quindi la dichiarazione di un **cursore**, infine la dichiarazione di un **HANDLER** per l'SQLSTATE 02000, di cui vedremo tra poco l'utilità.

La prima vera operazione è l'apertura del cursore (**OPEN**), che esegue la query associata utilizzando uno dei parametri di input. Di seguito, dopo avere inizializzato il parametro di output (che servirà da contatore delle righe lette), eseguiamo la prima **FETCH**, cioè la lettura della prima riga di risultato della query, i cui valori sono assegnati alle variabili a e b. Di seguito c'è un ciclo **WHILE** (al quale abbiamo assegnato proprio il nome 'ciclo') che viene eseguito fintanto che il valore di 'finito' è falso, cioè è uguale a zero come da inizializzazione. Tale valore verrà modificato nel momento in cui non ci sono più righe da leggere, come specificato con l'handler (02000 è infatti lo stato che indica tale situazione).

All'interno del ciclo si verifica se la variabile 'param3' ha già raggiunto il valore di 'param1', che era l'altro parametro in input: in questo caso l'istruzione **LEAVE** consente di abbandonare il ciclo stesso; in caso contrario si incrementa la variabile e si esegue una nuova lettura del cursore, sempre riportando i risultati nelle variabili a e b (l'esempio è solo dimostrativo, in quanto non utilizziamo mai questi valori).

Il ciclo quindi termina quando sono finite le righe del cursore oppure quando ne abbiamo lette tante quante indicate dal primo parametro, a seconda di quale dei due eventi si verifica prima. Al termine dell'esecuzione il parametro di output conterrà il numero di righe lette. L'istruzione DELIMITER viene utilizzata di nuovo per ripristinare il delimitatore punto e virgola, una volta che la procedura è stata memorizzata.

Vediamo quindi quali elementi possiamo trovare in una stored procedure:

- **Variabili**: sono quelle dichiarate con l'istruzione DECLARE, più i parametri ricevuti dalla procedura
- **Condizioni**: corrispondono a codici errore di MySQL oppure valori di SQLSTATE ai quali possiamo dare un nome per poi gestirli con un HANDLER
- **Cursori**: si usano per eseguire query che restituiscono un resultset quando vogliamo scorrere tale resultset all'interno della procedura
- **Handler**: istruzioni da eseguire al verificarsi di particolari condizioni; tali condizioni possono essere, oltre a quelle definite in precedenza, direttamente un SQLSTATE, oppure un codice errore o altre condizioni: SQLWARNING, NOT FOUND, SQLEXCEPTION. La dichiarazione deve anche specificare se, al verificarsi della condizione, la procedura deve continuare o terminare (CONTINUE o EXIT)

Tutti questi elementi sono creati con l'istruzione DECLARE, e devono trovarsi all'inizio del codice (dopo BEGIN) nell'ordine in cui li abbiamo elencati.

Vediamo ora cosa possiamo trovare nel codice vero e proprio:

- **Variabili**: possono essere manipolate con l'istruzione **SET** o ricevere i valori delle query con **INTO**
- **Cursori**: si aprono con **OPEN**, si chiudono con **CLOSE** (MySQL li chiude in automatico al termine della procedura se non lo facciamo esplicitamente), e si scorrono con **FETCH**, che provvede a leggere una riga e mettere i risultati in una o più variabili
- **Controllo di flusso**: sono costrutti che ci permettono di inserire una logica nella procedura. Abbiamo **IF** per le condizioni, **LOOP**, **REPEAT** e **WHILE** per i cicli, **ITERATE** e **LEAVE** per la gestione dall'interno dei cicli stessi

Vediamo un po' di codice di esempio:

```
DECLARE a INT default 0;
DECLARE cond1 CONDITION FOR 1045;
DECLARE cond2 CONDITION FOR SQLSTATE '02000';
DECLARE cur1 CURSOR FOR query;
DECLARE EXIT HANDLER FOR cond1 SET variabile = valore;
DECLARE CONTINUE HANDLER FOR cond2 SET variabile = valore;
DECLARE CONTINUE HANDLER FOR SQLWARNING;
OPEN cur1;
FETCH cur1 INTO variabile1 variabile2;
CLOSE cur1;
IF condizione THEN istruzioni
    [ELSEIF condizione THEN istruzioni] ...
    [ELSE istruzioni]
```

```
END IF;
[nome:] LOOP
    istruzioni
END LOOP [nome];
[nome:] REPEAT
    istruzioni
    UNTIL condizione
END REPEAT [nome];
[nome] WHILE condizione DO
    istruzioni
END WHILE [nome];
ITERATE nomeCiclo;
LEAVE nomeCiclo;
```

Nella prima riga definiamo una variabile; nella seconda diamo un nome al codice errore 1045; nella terza all'SQLSTATE '02000'; nella quarta definiamo un cursore. Di seguito definiamo gli handler per le due condizioni definite in precedenza, specificando che nel caso della prima interromperemo la procedura, mentre nel secondo la continueremo; in entrambi i casi impostiamo il valore di una variabile. Definiamo poi un handler anche per la condizione di SQLWARNING. Le tre istruzioni successive effettuano apertura, lettura e chiusura del cursore.

Abbiamo poi la sintassi della IF, seguita da quelle relative ai cicli; possiamo notare che il ciclo LOOP non ha condizione di uscita (dovremo quindi uscirne con un LEAVE o attraverso un handler), mentre il ciclo REPEAT ha una condizione di uscita (cioè termina quando la condizione è vera), ma viene comunque eseguito almeno una volta; il ciclo WHILE ha una condizione di permanenza (termina quando diventa falsa), e può anche non essere eseguito mai.

Con LEAVE abbandoniamo un ciclo, mentre con ITERATE passiamo all'esecuzione successiva (saltiamo cioè, per una esecuzione, la parte di codice interna al ciclo successiva alla ITERATE). I nomi dei cicli sono facoltativi, ma diventano indispensabili qualora dobbiamo referenziarli con LEAVE o ITERATE.

Una volta creata, la stored procedure può essere eliminata con **DROP PROCEDURE**, o modificata con **ALTER PROCEDURE**. Non è tuttavia possibile modificare il codice della procedura: per fare questo dovremo eliminarla e ricrearla.

ALTER PROCEDURE *nomeProcedura* SQL SECURITY { DEFINER | INVOKER };
DROP PROCEDURE *nomeProcedura*;

I permessi necessari

Per creare una stored procedure dobbiamo avere il privilegio CREATE ROUTINE, nonchè il privilegio SUPER in caso siano abilitati i log binari di MySQL. Per modificarla o

eliminarla ci serve il privilegio ALTER ROUTINE (che viene assegnato automaticamente al creatore della procedura). Per eseguirla infine è necessario il privilegio EXECUTE (anche questo assegnato in automatico all'autore) nonchè il permesso di accesso al database che la contiene.

I controlli sulle istruzioni contenute dipendono dal valore impostato per il parametro SQL SECURITY: nel caso sia DEFINER (il default), sarà l'utente che ha definito la procedura che deve avere i permessi necessari; se invece il valore del parametro è INVOKER i permessi sono richiesti a chi esegue la routine.

Le stored functions

Le stored functions sono simili alle stored procedures, ma hanno uno scopo più semplice, cioè quello di definire vere e proprie funzioni, come quelle già fornite da MySQL. Esse restituiscono un valore, e non possono quindi restituire resultset, al contrario delle stored procedures. Nelle versioni di MySQL precedenti alla 5.0 esistevano le "user-defined functions", che venivano memorizzate esternamente al server. Ora queste funzioni sono ancora supportate, ma è sicuramente consigliabile utilizzare le nuove stored functions.

Vediamo come crearle:

CREATE FUNCTION *nome* ([*parametro*[,...]])
RETURNS *tipo*
[SQL SECURITY { DEFINER | INVOKER }] *corpo*
//parametri:
nomeParametro tipo

Rispetto alle stored procedures, vediamo che si aggiunge la clausola RETURNS che specifica che tipo di dato la funzione restituisce. Inoltre nella lista dei parametri non esiste distinzione fra input e output, in quanto i parametri sono solo in input.

Esistono poi le istruzioni **ALTER FUNCTION** e **CREATE FUNCTION** che sono analoghe alle corrispondenti relative alle procedure. Anche le considerazioni sulla sicurezza sono le stesse relative alle stored procedures. Il codice contenuto può essere lo stesso, con la differenza che, come abbiamo già detto, non sono previsti parametri in output e non è possibile restituire un resultset. é invece **obbligatorio** restituire un dato del tipo indicato nella clausola RETURNS, attraverso l'istruzione **RETURN *valore***.

Per informazioni più approfondite su stored procedures e stored functions vi rimandiamo al **manuale ufficiale** di MySQL.

Trigger

I trigger sono oggetti associati a tabelle, che vengono attivati nel momento in cui un determinato evento si verifica relativamente a quella tabella. Sono stati introdotti a partire da MySQL 5.0.2.

Quando definiamo un trigger, stabiliamo per quale evento deve essere attivato (inserimento di righe, modifiche o cancellazioni) e se deve essere eseguito prima o dopo tale evento; avremo quindi i seguenti tipi di trigger:

- **BEFORE INSERT**
- **BEFORE UPDATE**
- **BEFORE DELETE**
- **AFTER INSERT**
- **AFTER UPDATE**
- **AFTER DELETE**

Il trigger stabilirà un'istruzione (o una serie di istruzioni) che saranno eseguite per ogni riga interessata dall'evento.

Ecco la sintassi per la creazione di un trigger:

```
CREATE
    [DEFINER = { utente | CURRENT_USER }]
    TRIGGER nome tipo
    ON tabella FOR EACH ROW
    [{ FOLLOWS | PRECEDES} nome_di_altro_trigger]
    istruzioni
```

Il trigger è associato ad una tabella, ma fa parte di un database, per cui il suo nome deve essere univoco all'interno del db stesso.

A partire dalla versione 5.7.2 di MySQL, è possibile definire più trigger, relativi ad una tabella, che vengano attivati con le medesime tempistiche per uno stesso evento. Di default, il loro ordine di attivazione è identico a quello di creazione ma può essere modificato sfruttando le parole chiave `FOLLOWS` e `PRECEDES`. Queste devono essere

accompagnate dal nome del trigger che, rispettivamente, sarà seguito o preceduto da quello che stiamo definendo.

Nelle versioni precedenti alla 5.7.2, al contrario, era impossibile avere per una stessa tabella, ad esempio, più trigger `BEFORE INSERT`, mentre se ne potevano avere due con tempistica `BEFORE` oppure due relativi all'`INSERT` ma con tempistiche diverse.

È importante tener presente che, riferendosi ad inserimenti o a cancellazioni delle righe, non si intende necessariamente una istruzione INSERT o DELETE, ma qualsiasi operazione dalla quale scaturisca l'evento interessato. Ad esempio, l'inserimento di dati può avvenire anche tramite una istruzione LOAD DATA.

Le istruzioni da eseguire all'attivazione del trigger possono essere una o più di una. In quest'ultimo caso si usa la sintassi per le istruzioni composte del tipo BEGIN ... END come già visto nella precedente lezione sulle stored procedures. Da notare che solo a partire da MySQL 5.0.10 il codice dei trigger può contenere riferimenti diretti alle tabelle.

La clausola **DEFINER** (introdotta in MySQL 5.0.17) specifica se come creatore del trigger deve essere considerato l'utente attuale (default) o un altro utente specificato nella forma *nome*@*host*. Questo sarà l'utente di cui saranno controllati i permessi al momento dell'esecuzione del trigger.

Una volta creato, il trigger può essere eliminato con l'istruzione DROP TRIGGER:

DROP TRIGGER [*database*.]*nome*

Ovviamente il nome del database, se omesso, viene considerato uguale al database in uso. È importante notare che, prima di MySQL 5.0.10, questa istruzione richiedeva che il nome del trigger fosse qualificato non col nome del database, ma con quello della tabella a cui era associato. Ne consegue che, in caso di upgrade da una versione precedente la 5.0.10, è necessario eliminare i trigger prima dell'aggiornamento e ricrearli in seguito; in caso contrario le istruzioni DROP TRIGGER non funzionerebbero dopo l'upgrade.

Vediamo un esempio pratico di trigger:

```
delimiter //
CREATE TRIGGER upd_check BEFORE UPDATE ON account
FOR EACH ROW
BEGIN
        IF NEW.amount < 0 THEN
                SET NEW.amount = 0;
        ELSEIF NEW.amount > 100 THEN
                SET NEW.amount = 100;
```

```
        END IF;
END;//
delimiter ;
```

Questo codice si attiva prima di ogni update sulla tabella account: su ognuna delle righe da modificare viene controllato il valore che sta per essere assegnato al campo amount, per verificare che sia compreso fra 0 e 100; in caso contrario viene riportato entro tali limiti. Come potete vedere, quindi, attraverso il trigger siamo in grado di modificare il valore che sta per essere aggiornato sulla tabella.

Il qualificatore **NEW** indica proprio che il nome di colonna che stiamo utilizzando si riferisce al nuovo valore della riga che sta per essere aggiornata. NEW si può utilizzare in caso di INSERT e UPDATE. Analogamente è disponibile il qualificatore **OLD** che fa riferimento ai valori precedenti la modifica, e si può utilizzare in caso di UPDATE e DELETE. La modifica attraverso l'istruzione **SET** è possibile solo per i valori NEW e solo nei trigger di tipo BEFORE.

Per lavorare sui trigger è attualmente necessario il privilegio SUPER.

Cifratura e decifratura dei dati

MySQL ha a disposizione una serie di funzioni che si occupano di **compressione, crittografia e codifica delle informazioni**. Il loro ruolo è determinante soprattutto sotto l'aspetto della sicurezza dei dati e della conservazione di password, qualora il DBMS costituisca la base di un sistema di autenticazione.

In questa lezione verranno presentate le funzioni appartenenti a questa categoria, suddivise per tipologie, mettendo in evidenza le indicazioni d'uso. Le funzioni, come si vedrà, possono essere sperimentate all'interno del client *mysql*, utilizzando solo il costrutto `SELECT`.

Funzioni hash

Molto comune è l'utilizzo di funzioni hash, che attuano una forma di crittografia "a senso unico". In pratica esse consentono di ricavare un stringa di lunghezza fissa ed apparentemente incomprensibile (detta **codice hash**), a partire da una stringa qualsiasi, facendo però in modo che l'operazione inversa sia impossibile. In altre parole, non dovrebbe essere possibile utilizzare il codice hash per risalire alla stringa originale. Le funzioni hash possono tornare utili per memorizzare password di utenti nel database, in maniera che non siano leggibili in modo diretto nel remoto caso in cui qualche malintenzionato ottenga l'accesso al DB.

L'operazione viene svolta tramite appositi algoritmi, e le funzioni che li applicano sono le seguenti:

- **MD5**: calcola una checksum da 128 bit, a partire da una stringa qualsiasi, restituendo un codice esadecimale di 32 caratteri:

```
> SELECT MD5('cane');

   0165801c0cce07c7a8751949845c93d2
```

- **SHA1**: applica l'algoritmo SHA (Secure Hash Algorithm) a 160 bit, che permette di ottenere un codice esadecimale di 40 caratteri:

```
> SELECT SHA1('cane');

   95df5d1bd9fbc6c52626039f90ab8d1b57406680
```

È possibile usare anche la funzione `SHA()`, ma si tratta di un sinonimo di SHA1;

- **SHA2**: implementa un algoritmo più sicuro rispetto a MD5 e SHA1 che può essere applicato nelle versioni SHA-224, SHA-256, SHA-384 e SHA-512. SHA2 richiede due parametri in input: il primo è la stringa da cifrare, il secondo è la lunghezza in bit desiderata per il risultato (i valori accettabili sono 224, 256, 384 e 512 mentre se si indica lo 0 verrà utilizzato ugualmente 256):

```
> SELECT SHA2('cane',256);

   df39220189ca8225b3ef4bb8f3654b2217c11b056cb764a806586f3b95530052
```

Funzioni per la compressione

Esistono alcune funzioni utili per **comprimere e decomprimere informazioni**, il cui utilizzo è subordinato all'installazione del modulo *zlib*.

Si tratta di:

- `COMPRESS`: comprime le informazioni fornendo come risultato una stringa binaria;
- `UNCOMPRESS`: decomprime delle informazioni compresse con `COMPRESS`;
- `UNCOMPRESSED_LENGTH`: fornisce la lunghezza della stringa prima di essere compressa.

Qualora fosse necessario immagazzinare nel database il risultato di una compressione, non si dovrebbero usare campi `CHAR` o `VARCHAR`, bensì `BLOB` o `VARBINARY`, più adatti per stringhe binarie.

Funzioni per la crittografia

Per la crittografia viene utilizzato l'algoritmo **AES** (**A**dvanced **E**ncryption **S**tandard), molto conosciuto nel mondo informatico. Le funzioni a disposizione sono `AES_ENCRYPT` per crittografare e `AES_DECRYPT` per decrittare.

La crittografia che viene applicata è simmetrica: si utilizza una chiave (generalmente una stringa) per codificare i dati, e la stessa password dovrà essere fornita per leggerli in chiaro.

Ad esempio, la seguente invocazione:

```
> SELECT AES_ENCRYPT('questo è il mio segreto','melarossa');
```

cifrerà la frase "*questo è il mio segreto*" utilizzando la string*melarossa* come password. La stessa chiave segreta deve essere fornita al momento di invocare `AES_DECRYPT`.

Verificare la forza di una password

Proprio per il ruolo che MySQL svolge nella conservazione delle credenziali di utenti, nella versione 5.6.6 del DBMS è stata introdotta una nuova funzione, chiamata **`VALIDATE_PASSWORD_STRENGTH`**, che riceve in input una password, scritta in chiaro, e restituisce un intero compreso tra 0 e 100 che ne indica la forza: 0 è il livello di debolezza assoluta, 100 il massimo della forza.

Il funzionamento di `VALIDATE_PASSWORD_STRENGTH` si basa su criteri che valutano le vulnerabilità della parola chiave scelta come quantità e tipo di caratteri utilizzati. Il livello di sufficienza per la forza di una password dovrebbe superare il valore 50 e possibilmente il 75.

Funzioni deprecate

Nell'ambito delle funzioni di codifica ne esistono diverse, tra l'altro molto usate in passato, che attualmente sono deprecate e pertanto dovrebbero essere evitate perchè rivelatesi **insicure**, e quindi probabilmente non più supportate in futuro.

Eccone un elenco:

- `DES_ENCRYPT` e `DES_DECRYPT`: funzioni crittografiche basate sull'algoritmo DES. Sono state decprecate nella versione 5.7.6 e dovrebbero essere sostituite con le menzionate `AES_ENCRYPT` e `AES_DECRYPT`;
- `ENCRYPT`: funzione crittografica basata sulla chiamata di sistema Unix `crypt()`. Come le precedenti, dovrebbe essere rimpiazzata da `AES_ENCRYPT`;
- `PASSWORD` e `OLD_PASSWORD`: fornivano una versione hash di una password. Possono essere rimpiazzate con `MD5`, `SHA1` o `SHA2`;
- `ENCODE` e `DECODE`: rispettivamente si occupavano della crittografia e decrittografia di una stringa tramite una password. Anche queste dovrebbero essere sostituite con `AES_ENCRYPT` e `AES_DECRYPT`.

GRANT e REVOKE, gestire i permessi

GRANT e REVOKE

I permessi possono essere gestiti in due modi: attraverso le istruzioni SQL GRANT e REVOKE, oppure con le normali istruzioni SQL (INSERT, UPDATE ecc.) sulle tabelle interessate.

La differenza da tenere presente è che nel primo caso le modifiche sono immediatamente effettive, mentre nel secondo caso è necessario usare il comando FLUSH PRIVILEGES per costringere MySQL a ricaricare in memoria le tabelle dei permessi.

Vediamo alcuni esempi di sintassi per GRANT e REVOKE:

GRANT SELECT ON acquisti.* TO luca@localhost IDENTIFIED BY ‘password’ WITH GRANT OPTION

Questa istruzione assegna il privilegio SELECT all’utente luca@localhost sul database acquisti. Se l’utente non esisteva in precedenza, la riga relativa viene aggiunta alla tabella user e ‘password’ sarà la sua password. Se l’utente esisteva già, la password viene sostituita.

Il permesso relativo alle SELECT sarà regisrato sulla tabella db, essendo assegnato a livello di database. Inoltre all’utente viene assegnato il permesso GRANT, grazie al quale sarà in grado di assegnare ad altri utenti i propri permessi. Attenzione: con questa opzione l’utente potrà assegnare ad altri **tutti** i propri permessi: non solo quelli ricevuti con questa istruzione, ma anche quelli che aveva già e quelli che riceverà in futuro.

GRANT ALL ON acquisti.ordini TO paolo@localhost

Questa istruzione assegna tutti i permessi sulla tabella ordini del database acquisti all’utente paolo@localhost. Non è stata specificata una password, per cui se l’utente esisteva già questa non sarà modificata. Se invece l’utente non esisteva, viene creato senza password.

Attenzione però: l'opzione sql-mode del server (v. lez. 4) prevede il valore NO_AUTO_CREATE_USER fra quelli possibili. Se questo valore è attivo, non sarà possibile creare implicitamente un utente senza password con una GRANT. In questo caso l'istruzione fallirebbe, a meno che ovviamente l'utente non fosse già esistente. In questo esempio abbiamo visto che non viene assegnato il permesso GRANT, ma se l'utente lo possedeva già sarà comunque in grado di riassegnare questi permessi.

REVOKE SELECT on acquisti.* FROM luca@localhost REVOKE ALL PRIVILEGES, GRANT OPTION FROM paolo@localhost

Con la prima istruzione togliamo il privilegio SELECT sul db acquisti all'utente luca@localhost.
Con la seconda togliamo tutti i privilegi sulle tabelle più quello di GRANT a paolo@localhost. In questo caso l'utente rimarrà privo di privilegi, ma la sua utenza non viene comunque eliminata dalla tabella user.

È importante ricordare che quando si elimina un database o una tabella, tutti i permessi esistenti **rimangono attivi**. Ovviamente questo è influente nel caso in cui venissero ricreati oggetti con lo stesso nome.

Vediamo ora alcune altre istruzioni relative alla gestione dei permessi:

CREATE USER alberto@localhost;
CREATE USER fabio@localhost IDENTIFIED BY 'password';
DROP USER alberto@localhost;
SET PASSWORD = PASSWORD('pw');
SET PASSWORD FOR paolo@localhost = PASSWORD('pw');

La prima istruzione crea un utente senza password (in questo caso funziona anche se NO_AUTO_CREATE_USER è attivo); la seconda crea un utente con password. Tali utenti non devono essere già esistenti. La terza istruzione elimina un utente. La quarta imposta la password 'pw' per l'utente collegato; l'ultima imposta la password 'pw' per l'utente paolo@localhost.

Le prime tre istruzioni richiedono il permesso CREATE USER; la quarta è possibile per chiunque sia collegato come utente non anonimo; la quinta richiede il permesso UPDATE sul database mysql.

Vediamo ora quali sono i principali permessi che possono essere assegnati ad un utente relativamente alle tabelle e ai database, e le istruzioni che autorizzano:

Permesso	**Istruzioni**
ALL	tutte esclusa GRANT

ALTER	ALTER TABLE
CREATE	CREATE TABLE
CREATE TEMPORARY TABLES	CREATE TEMPORARY TABLE
CREATE VIEW	CREATE VIEW
DELETE	DELETE
DROP	DROP TABLE
INDEX	CREATE INDEX, DROP INDEX
INSERT	INSERT
LOCK TABLES	LOCK TABLES
SELECT	SELECT
SHOW VIEW	SHOW CREATE VIEW
UPDATE	UPDATE
USAGE	nessuna
GRANT OPTION	GRANT, REVOKE

Ci sono alcuni permessi di tipo amministrativo che non ha senso riferire ad un database: tali permessi si trovano infatti solo sulla tabella user. Vediamone alcuni:

Permesso	**Istruzioni**
CREATE USER	CREATE USER, DROP USER, RENAME USER, REVOKE ALL PRIVILEGES
FILE	SELECT … INTO OUTFILE, LOAD DATA INFILE
PROCESS	SHOW FULL PROCESSLIST
RELOAD	FLUSH
SHOW DATABASES	SHOW DATABASES
SHUTDOWN	mysqladmin shutdown
SUPER	KILL, SET GLOBAL

SHOW DATABASES può essere utilizzata anche dagli utenti che non possiedono il permesso relativo (a meno che il server non sia stato avviato con l'opzione –skip-show-database).

Tuttavia questi non vedranno tutti i database presenti sul server, ma solo quelli per i quali possiedono diritti. In generale, comunque, gli utenti non amministrativi non dovrebbero mai possedere i permessi di quest'ultimo gruppo.

L'ultimo argomento da segnalare riguardo al sistema dei permessi è quello molto importante relativo alla **memorizzazione delle password**. Le password di ogni utente sono memorizzate nella colonna password della tabella user: un campo di 41 caratteri di cui il primo è un asterisco mentre i successivi 40 sono il risultato dell'algoritmo di hashing sulla password che è stata impostata con una GRANT o una CREATE USER, oppure con la funzione PASSWORD().

L'algoritmo di cifratura è monodirezionale, per cui non è possibile risalire alla password partendo dalla stringa criptata. Quando l'utente tenta di collegarsi e digita la password, il client esegue la cifratura ed invia la password criptata al server, che la confronta con quella memorizzata sul database.

Un elemento rilevante però è che l'algoritmo di cifratura attualmente utilizzato è stato **introdotto con la versione 4.1 di MySQL**: le versioni precedenti utilizzavano un algoritmo più semplice che produceva una stringa criptata di soli 16 caratteri. Questa situazione ha creato un problema di compatibilità che si verifica quando un client di tipo 'vecchio' (cioè di versione pre-4.1) tenta di collegarsi ad un server di tipo 'nuovo': i vecchi client infatti non sono in grado di supportare il nuovo algoritmo di hashing della password.

La conseguenza di ciò è che un client vecchio non può collegarsi ad un server nuovo se la password dell'utente che cerca di collegarsi è memorizzata col nuovo sistema. La situazione più tipica in cui ciò si verifica è l'utilizzo della vecchia **estensione mysql** del linguaggio PHP, molto utilizzato con MySQL. Tale estensione infatti è un client pre-4.1, e quindi non supporta il nuovo algoritmo di cifratura.

La soluzione è che un server di tipo nuovo può memorizzare password sia di un tipo che dell'altro: essendo evidente la differenza fra i due tipi di cifratura, il server è in grado di distinguerli facilmente e quindi può consentire l'accesso sia agli utenti con la vecchia password che a quelli con la nuova.

L'opzione del server *–old-passwords* è quella che determina quali tipi di password vengono creati dalle istruzioni GRANT, SET PASSWORD e dalla funzione PASSWORD(). Se il server viene avviato con questa opzione, le password generate saranno di tipo vecchio. Se invece l'opzione non è attiva, verranno generate password di tipo nuovo.

In sostanza quindi, per utilizzare un client di tipo vecchio su un server post-4.1, è necessario utilizzare un utente la cui password è cifrata col vecchio algoritmo.

Riassumendo:

- Sul server possono coesistere utenti con la password vecchia e utenti con la password nuova. I client di tipo nuovo possono accedere a tutte le utenze, quelli di tipo vecchio solo alle utenze con la password vecchia.
- Se il server è avviato senza opzione –old-passwords, le password vengono generate col formato nuovo. Ciò significa che i nuovi utenti creati non saranno accessibili dai vecchi client. Inoltre c'è il rischio che venga modificata la password di un utente che si collega con un client vecchio: la nuova password sarà cifrata con il nuovo algoritmo e l'utente non sarà più in grado di collegarsi. Per generare password di tipo vecchio è possibile utilizzare la funzione

OLD_PASSWORD('password').

- Se il server è avviato con l'opzione –old-passwords, tutte le password verranno generate nel formato vecchio. In questo modo non verranno mai generate password lunghe, e anche quelle preesistenti, se modificate, torneranno al vecchio formato. In questo caso si mantiene la compatibilità con tutti i client, ma si perde la maggior sicurezza derivante dal nuovo algoritmo di cifratura

Evidentemente toccherà all'amministratore del database stabilire, in base alle esigenze dei propri utenti, se utilizzare o meno l'opzione `--old-passwords`.

Gestire gli utenti

Il sistema dei permessi di MySQL è un sistema piuttosto avanzato ma non standard, basato sul contenuto del database *mysql* che troverete preinstallato sul vostro server dopo l'esecuzione dello script *mysql_install_db* (se avete seguito il nostro tutorial per l'installazione nella lezione 2 lo script sarà già stato eseguito automaticamente).

Il primo concetto essenziale da tener presente è che **l'identificazione dell'utente** non avviene semplicemente attraverso il suo nome utente, ma dalla combinazione di questo con la macchina da cui l'utente si collega. Quindi due utenti che si collegano con lo stesso nome ma da due indirizzi diversi, per MySQL sono **due utenti diversi**.

La prima ad essere consultata quando un utente cerca di connettersi al server è la tabella **user**. Il server cerca infatti di riconoscere l'utente in base ai valori contenuti nelle colonne 'Host' e 'User' di tale tabella: una volta che l'utente è stato "riconosciuto", gli sarà consentito l'accesso se digiterà la password specificata sulla tabella per quella combinazione di utente e host.

La colonna 'Host' della tabella user può contenere nomi host (ad es. *www.mysql.com*) oppure indirizzi ip. È possibile utilizzare caratteri "wild card" che rappresentano un carattere (_) o "n" caratteri (%). Quindi ad esempio il valore '%.mysql.com' sarà considerato valido per qualsiasi dominio di terzo livello appartenente a mysql.com.

Nel caso venga utilizzato l'indirizzo ip, sarà possibile anche indicare una maschera per specificare quanti bit devono essere utilizzati per trovare la corrispondenza nell'indirizzo; è da notare però che il numero di bit deve essere necessariamente multiplo di 8 (8, 16, 24, 32). Ad esempio il valore '151.42.62.0/255.255.255.0' significa che la riga è valida per gli IP da 151.42.62.0 a 151.42.62.255.

Nella colonna 'User' sono contenuti evidentemente i nomi degli utenti che si possono collegare. Qui non è consentito l'utilizzo delle wildcard; è consentito però lasciare vuoto il valore: in questo caso tutti gli utenti che si collegheranno dal nome host (o indirizzo IP) corrispondente saranno riconosciuti, qualsiasi nome utilizzino.

In questo caso però saranno considerati da MySQL come 'utenti anonimi': questo significa che il nome utente col quale si sono presentati non sarà valido quando verranno fatti i controlli relativi ai permessi per le varie operazioni.

È evidente che con questo sistema può capitare che un utente, quando cerca di collegarsi, possa essere riconosciuto in base al contenuto di più righe della tabella user: vediamo infatti un esempio:

```
+-----------+----------+-
| Host      | User     |
+-----------+----------+-
| %         | paolo    |
| %         | luca     |
| localhost | paolo    |
| localhost |          |
+-----------+----------+-
```

In questa situazione, se l'utente paolo si collega da localhost, sia la prima riga che la terza sono in grado di riconoscerlo, in quanto il valore '%' vale per qualsiasi host. Allo stesso modo, se si presenta luca (sempre da localhost), sarà riconosciuto dalla seconda riga ma anche dalla quarta, in quanto il nome utente vuoto corrisponde a qualsiasi utente.

Tuttavia MySQL determina **una sola riga** da utilizzare per riconoscere l'utente (e quindi assegnargli poi i relativi privilegi).

Tale riga viene scelta riconoscendo l'utente **prima** in base al nome host, e fra i nomi host viene privilegiato quello più specifico (quindi senza wildcard); di seguito viene cercato il nome dell'utente, e anche in questo caso il nome esatto ha la precedenza sull'utente anonimo. La prima riga che corrisponde al nome utente sulla base di questi criteri è quella che verrà utilizzata per l'autenticazione. Quindi, nel nostro esempio precedente, quando paolo si collega da localhost verrà riconosciuto dalla terza riga (nome host più specifico), mentre la prima riga sarà usata quando si collega da altri indirizzi.

Per quanto riguarda luca invece la situazione può facilmente trarre in inganno, perchè quando si collega da localhost sarà riconosciuto in base alla quarta riga e non alla seconda: infatti il riconoscimento del nome host ha la precedenza su quello del nome utente, e quindi la prima riga che soddisfa le credenziali di **luca@localhost** è quella con utente anonimo e localhost: di conseguenza la connessione di luca avverrà come **utente anonimo** e non con il suo nome utente!

Naturalmente, una volta riconosciuto l'utente, questi dovrà fornire la password corretta, che il server confronterà con quella contenuta nell'omonima colonna della tabella user.

Ci soffermeremo fra poco sul modo in cui la password viene memorizzata; per il momento vogliamo citare il caso in cui la colonna password della tabella sia vuota. Infatti si potrebbe pensare che in questo caso all'utente venga sempre concesso l'accesso, ma **non è così**: infatti l'assenza di password significa che l'utente **non deve** digitare una password; in caso contrario l'accesso gli verrà sempre negato!

Possiamo utilizzare due funzioni di MySQL per sapere con quale utente ci siamo presentati, e con quale siamo stati autenticati:

```
SELECT USER();
-> mario@localhost
SELECT CURRENT_USER();
-> @localhost
```

Con **USER** otteniamo il nome utente e host con i quali abbiamo richiesto l'accesso, mentre **CURRENT_USER** ci indica quale utente ha usato MySQL per autenticarci. Nell'esempio l'utente si è presentato come 'mario', ma è stato autenticato come utente anonimo.

Una volta ottenuto l'accesso al server, l'utente deve avere i **permessi** necessari per lavorare sui vari database. La tabella user contiene numerose colonne relative ai permessi ('Select_priv', 'Insert_priv', 'Update_priv' ecc.) ciascuna delle quali può contenere il valore 'Y' o 'N'. La presenza di 'Y' significa che l'utente è autorizzato a compiere quell'operazione: il fatto di avere il permesso sulla tabella user implica che il permesso è valido per tutti i database del server. In sostanza, un utente 'normale' non dovrebbe avere permessi di questo tipo sulla tabella user.

La tabella **db** contiene invece i permessi relativi ai singoli database. La chiave di questa tabella è formata da User, Host e Db: quindi ogni riga specifica quali permessi ha un determinato utente su un determinato database. Anche in questo caso possiamo trovare le wildcard per host e db; la colonna db può anche essere vuota per indicare tutti i database. La colonna user invece, se vuota, vale solo per gli utenti anonimi.

Un caso particolare è quando il campo Host della tabella db è vuoto: in questo caso infatti entra in gioco la tabella **host**, che specifica i permessi sui vari database per i diversi host. Qui la chiave Host e Db, e ancora una volta è possibile avere le wildcard su entrambe le colonne. Il valore vuoto corrisponde alla wildcard '%'.

Quando viene usata la tabella host, i permessi per l'utente derivano dall'**intersezione** dei permessi trovati sulla tabella db con quelli concessi dalla tabella host: questo significa che l'operazione richiesta è consentita solo se il valore relativo è 'Y' su entrambe le tabelle, nelle righe corrispondenti. Vediamo un esempio:

Tabella db

Host	Db	User	Select_priv	Insert_priv
192.168.0.%	acquisti	paolo	Y	N
	vend%	paolo	Y	N

Tabella Host

Host	Db	Select_priv	Insert_priv
192.168.0.11	%	N	N
192.168.0.%	%	Y	Y

Ovviamente per semplicità abbiamo indicato solo due delle colonne relative ai privilegi.

Supponendo che l'utente paolo sia collegato dalla macchina 192.168.0.11 e che non abbia nessun permesso sulla tabella user, ipotizziamo che voglia lavorare sul database acquisti: in questo caso verrà riconosciuto dalla prima riga della tabella db, che gli consentirà di fare le select ma non le insert.

Quando invece paolo cercherà di lavorare sul database vendite, verrà riconosciuto dalla seconda riga, che però non riporta un indirizzo host, per cui il server andrà a verificare i permessi sulla tabella host. Su questa tabella la riga che gli compete è la prima, in quanto coincide esattamente con l'indirizzo della sua macchina ed è valida per qualsiasi database.

Questa riga però gli nega i permessi sia sulla insert che sulla select, per cui paolo, nonostante la tabella db lo autorizzi alle select, **non è in grado di effettuare alcuna operazione** sul database vendite. Se si fosse collegato da un'altra macchina con indirizzo 192.168.0.%, sarebbe riuscito almeno a fare le select (non le insert, che gli vengono comunque negate dalla tabella db).

Anche per queste tabelle quindi vale il principio che i valori di host, database e user vengono riconosciuti preferenzialmente in base alla maggiore specificità del contenuto dei campi, quando ci sono più righe che coincidono con la richiesta. Per questo paolo non può lavorare sul database vendite quando si collega da 192.168.0.11: la seconda riga della tabella host gli consentirebbe di farlo, ma la prima ha un indirizzo più specifico e quindi prevale.

Esistono poi le tabelle **tables_priv** e **columns_priv**, che contengono privilegi ancora più specifici in quanto relativi ad una singola tabella (nel primo caso) e addirittura ad una singola colonna (nel secondo caso).

Queste tabelle vengono consultate quando i privilegi assegnati dalle tabelle user, db e host non sono sufficienti a garantire il permesso di effettuare l'operazione richiesta. Entrambe contengono nella chiave i campi Host, Db e User, oltre ad una colonna Table_name che identifica la tabella interessata; la tabella columns_priv contiene un'ulteriore colonna Column_name che specifica a quale colonna si riferiscono i permessi. In queste tabelle solo il campo host può contenere wildcard od essere vuoto (in questo caso la riga vale per tutti gli indirizzi), mentre il database deve sempre

essere specificato. Inoltre queste tabelle non contengono una colonna per ogni permesso come le precedenti, bensì le colonne *Table_priv* e *Column_priv* che contengono tutti i permessi assegnati dalla riga.

Riassumendo: quando il server MySQL deve decidere se un determinato utente ha il permesso di compiere l'operazione che sta richiedendo, il controllo viene fatto per passi successivi: prima si controlla la tabella user, poi la combinazione di db e host, infine tables_priv e columns_priv.

Da notare che ad ogni passaggio i permessi trovati vengono sommati a quelli precedenti: infatti una singola istruzione può richiedere più permessi, e un utente potrebbe averli "sparsi" sui vari livelli. Ovviamente quando tutti i permessi necessari sono stati trovati la ricerca si interrompe e l'istruzione viene eseguita; se invece si arriva in fondo senza averli trovati tutti l'istruzione verrà negata.

Dump, backup e recovery

Il backup e il recovery dei dati sono, da sempre, attività fondamentali per garantire la sicurezza dei dati di un DBMS. In particolare, attraverso i backup effettuiamo salvataggi del contenuto del database ad un determinato momento, mentre il recovery è l'operazione con la quale ripristiniamo il contenuto del database a seguito di un danneggiamento.

Per poter effettuare dei recovery completi in caso di problemi, è necessario eseguire il server MySQL con i log attivati (opzione –log-bin del server); in caso contrario, tutto ciò che sarà possibile, se ci saranno problemi, sarà ripristinare i dati all'ultimo backup, perdendo le modifiche successive.

Sono comunque piuttosto rari i casi in cui si rende necessario ricorrere all'uso dei backup per recuperare i dati: in genere si tratta di guasti all'hardware o all'alimentazione del sistema, oppure crash del sistema operativo o del filesystem. Anche in questi casi tuttavia è possibile che MySQL sia in grado di riavviarsi correttamente.

Backup dei dati

La prima misura di sicurezza da adottare nei confronti dei nostri dati è quindi quella di effettuare backup periodici. Il modo migliore per fare ciò è usare il programma **mysqldump**, che ci consente di salvare i nostri dati in maniera molto semplice.

Attenzione! In tutti gli esempi di codice di questa lezione che lanciano programmi, alle opzioni indicate vanno aggiunte quelle relative ad utente e password per l'accesso al server – ved. lezione 3

mysqldump –single-transaction –all-databases > *nome_file*

Con questo comando otteniamo, nel file *nome_file*, una lista di istruzioni INSERT che ci permettono di ripristinare l'intero contenuto del database. Il file è in formato testo, quindi possiamo aprirlo e verificarne il contenuto. L'opzione –single-transaction è utile se il database contiene tabelle InnoDB, in quanto permette di ottenere una vista consistente di tali tabelle, cioè non influenzate da eventuali aggiornamenti effettuati durante l'operazione di backup. Per quanto riguarda invece le tabelle non transazionali (MyISAM) non è possibile avere questa garanzia; bisogna quindi essere certi che le tabelle non vengano modificate durante il backup.

Se vogliamo effettuare il backup di un **singolo database** invece che dell'intero server possiamo farlo omettendo l'opzione –all-databases e indicando al suo posto il nome del database che ci interessa.

Nel momento in cui vogliamo ricaricare sul server il contenuto del backup effettuato, è sufficiente far leggere il file di backup al client mysql:

mysql < *nome_file*
mysql *nome_db* < *nome_file*

La seconda sintassi va usata nel caso in cui il backup contenga i dati di un singolo database; in questo caso infatti nel backup non ci sono riferimenti al database utilizzato.
Immaginiamo ora di volerci garantire la possibilità di effettuare, in caso di crash, un recovery completo dei dati fino al momento del guasto. Innanzitutto dovremo quindi preoccuparci di abilitare i file di log. Per fare questo dobbiamo aggiungere al file di configurazione, nella sezione *mysqld*, l'opzione **log-bin**:

log-bin=*percorso*/*basename*

Il server creerà quindi i file di log nella directory indicata, e come nome di file utilizzerà il *basename* aggiungendovi, come estensione, un numero progressivo di 6 cifre che viene incrementato ad ogni nuovo file di log. È anche possibile omettere l'indicazione del percorso e del basename, utilizzando semplicemente l'opzione *log-bin*; in questo caso i log verranno creati nella directory dei dati di MySQL, e come nome dei file verrà usato '*nome_computer*-bin'. L'ideale tuttavia sarebbe che i log si trovassero su un'unità disco diversa da quella dei dati, in modo che un'eventuale guasto dell'hardware non comprometta la loro disponibilità.

Ad ogni riavvio del server MySQL crea un nuovo file di log, incrementando il progressivo; inoltre è possibile forzare questa operazione attraverso l'istruzione **FLUSH LOGS**.
Per una migliore gestione dei backup in questo caso potremo utilizzare un formato più esteso del comando di esecuzione di mysqldump quando effettuiamo il salvataggio completo dei dati:

mysqldump –single-transaction –flush-logs –master-data=2 –all-databases –delete-master-logs > *nome_file*

Come vedete abbiamo aggiunto due opzioni: con **flush-logs** MySQL crea un nuovo file di log, che sarà il primo da utilizzare in caso di ripristino dei dati a partire da questo backup; con **master-data=2** sul file di backup viene scritto (in forma di commento) il nome del file di log appena creato; infine l'opzione **delete-master-logs** cancella i log precedenti, che non servono più.

In caso di disastro, quindi, dovremo innanzitutto ripristinare il backup principale come visto in precedenza; poi ci occuperemo dei file di log, attraverso il programma

mysqlbinlog:

mysqlbinlog *nome_file_log nome_file_log* | mysql
mysqlbinlog *nome_file_log nome_file_log* > *nome_file*

Con questo comando diamo in input due file di log al programma: nel primo caso gli indichiamo di indirizzare l'output sul client mysql in modo da rieseguire le istruzioni memorizzate nei log (ricordiamoci di aggiungere i dati per la connessione!), e quindi ricostruire la situazione della base dati. È importante, qualora i file di log da elaborare siano più di uno, rieseguirli tutti con una unica istruzione. Nel secondo caso invece scriviamo l'output su un file di testo in modo da poterlo poi visualizzare ed esaminare.

Nel leggere i log binari abbiamo la possibilità di delimitare, in due modi, le istruzioni da prendere in considerazione: possiamo usare un timestamp di inizio e di fine, oppure le posizioni sul file.

Vediamo come:

mysqlbinlog –start-date="2006-01-25 9:55:00" –stop-date="2006-01-25 10:00:00" *nome_file*
mysqlbinlog –start-position="2345" –stop-position="4567" *nome_file*

Nel primo caso diciamo a mysqlbinlog di leggere solo le istruzioni comprese fra le 9.55 e le 10.00 della data indicata; con il secondo invece indichiamo le posizioni nel file di log a cui fare riferimento. In entrambi i casi possiamo usare anche una sola delle due opzioni (inizio o fine). Per sapere a quali posizioni fare riferimento, possiamo esaminare l'output del programma sul file di testo, che riporta per ogni istruzione memorizzata la posizione di inizio e di fine.

Manutenzione delle tabelle

Ci occupiamo ora della manutenzione delle tabelle MyISAM, per le quali abbiamo a disposizione il programma **myisamchk**. Tuttavia questo programma andrebbe utilizzato a server non attivo, in quanto può causare problemi qualora tenti di accedere alle tabelle in contemporanea al server MySQL: il nostro consiglio è quindi quello di utilizzare, al suo posto, le istruzioni SQL che svolgono le stesse funzioni:

CHECK TABLE *tabella* [*opzione*]
REPAIR TABLE *tabella* [*opzione*]
OPTIMIZE TABLE *tabella*

CHECK TABLE si usa per verificare lo stato di una tabella e vedere se ci sono dei problemi.
I possibili valori di *opzione* sono: QUICK, FAST, MEDIUM, EXTENDED, CHANGED; tali

opzioni sono in ordine crescente di complessità, e garantiscono un controllo via via più accurato ma anche più lento. L'output di CHECK TABLE dovrebbe, normalmente, segnalare che la tabella è OK; in caso contrario è necessario ripararla con **REPAIR TABLE**.
Anche qui abbiamo la possibilità di indicare più opzioni (QUICK, EXTENDED, USE_FRM). Il consiglio è di iniziare con la prima e provare le successive solo in caso di insuccesso. Anche in questo caso avremo un output che ci comunica l'esito dell'operazione.

L'istruzione **OPTIMIZE TABLE** serve, infine, per ottimizzare l'occupazione di spazio di una tabella: è bene eseguirla in particolare quando sono state effettuate molte cancellazioni o molti aggiornamenti su una tabella.

Un'altra alternativa alle istruzioni SQL per la manutenzione delle tabelle è il programma client **mysqlcheck**, che a differenza di myisamchk può essere eseguito tranquillamente anche a server avviato.

Ottimizzare il database

L'ottimizzazione nell'uso di un database è un argomento estremamente complesso, in quanto è condizionato da una notevole quantità di variabili. Esistono concetti che sono applicabili in generale alle basi di dati relazionali, e altri che sono specificamente relativi ad un certo RDBMS, in dipendenza delle sue caratteristiche. Naturalmente finchè le nostre applicazioni sono di dimensioni limitate, sia come quantità di dati, sia come numero di utenti che accedono al database, difficilmente noteremo problematiche di questo genere.

Se invece ci troviamo a gestire applicazioni che devono supportare numerosi accessi simultanei (ad esempio siti web che riscuotono un notevole successo) oppure basi di dati che assumono una notevole consistenza (nell'ordine almeno delle centinaia di migliaia di righe, il che può succedere anche per applicazioni con pochi utenti) potrà capitarci di avere un degrado più o meno forte nelle prestazioni, che di solito può essere risolto (o perlomeno limitato) ottimizzando alcuni aspetti dell'applicazione o della base dati, o in alcuni casi della configurazione del server.

In questa lezione faremo alcuni accenni all'ottimizzazione di MySQL, alla quale è dedicato **un intero capitolo** del manuale.

Il primo livello di ottimizzazione al quale possiamo guardare è quello relativo **al server e alla sua configurazione**. Come abbiamo già visto nella **lezione 4**, sono molto numerose le variabili che influiscono sul funzionamento di MySQL. L'amministratore del server ha la possibilità di impostarne i valori attraverso i file di configurazione, oppure con opzioni al momento dell'avvio, o ancora modificandole a server attivo.

L'istruzione SQL **SHOW VARIABLES** ci consente di visualizzare i valori di tutte le variabili in uso sul server (sebbene non tutte abbiano un'influenza diretta sulle prestazioni). Ovviamente sarebbe molto lungo spiegare il significato di tutte le variabili; inoltre è necessario tempo anche per abituarsi a valutare l'impatto di ciascuna di esse sul funzionamento del server. Ci limiteremo quindi a dire che le prime da prendere in considerazione per quanto riguarda l'ottimizzazione sono **key_buffer_size** e **table_cache**: la prima rappresenta la quantità di spazio di memoria che viene utilizzata

da MySQL per tenere in memoria i valori degli indici delle tabelle MyISAM, in modo da limitare gli accessi al disco (può essere impostato intorno al 25% del totale della memoria per una macchina su cui MySQL è l'applicazione principale); la seconda invece indica il numero di tabelle che il server può mantenere aperte contemporaneamente. Raggiunto questo numero, MySQL dovrà chiudere una tabella ogni volta che ha la necessità di aprirne un'altra.

Un accorgimento che può consentire di risparmiare tempo su tutte le istruzioni inviate al server è quello di **utilizzare un sistema semplice di permessi**: in sostanza, evitare completamente di attribuire permessi a livello di tabella o di colonna, e limitarsi a dare permessi sui database. Infatti, se le tabelle *tables_priv* e *columns_priv* del database mysql non contengono dati, MySQL non dovrà andare ogni volta a verificare i permessi su di esse.

Il secondo livello di ottimizzazione riguarda **la struttura delle basi di dati**, cioè il modo in cui vengono progettate le tabelle. Vediamo qualche suggerimento:

- le tabelle MyISAM sulle quali vengono effettuati frequenti aggiornamenti sono più veloci se non hanno righe a lunghezza variabile; naturalmente dovete tenere presente che usare righe a lunghezza fissa può avere la controindicazione di sprecare spazio, per cui bisogna fare una valutazione su quale dei due aspetti è prioritario;
- le tabelle MyISAM possono rivelarsi piuttosto lente nel caso in cui abbiano frequenti aggiornamenti e siano lette da query lente; in questo caso è bene considerare la possibilità di cambiare storage engine (vedere **lez. 12**);
- cercate di limitare al minimo l'occupazione di spazio, perchè questo consente al server di leggere maggiori quantità di dati con un accesso al disco: di conseguenza valutate sempre qual è il campo più piccolo adattabile ai vostri dati e non utilizzatene uno più grande (ad esempio, per valori interi, un campo MEDIUMINT occupa 3 byte mentre un INT ne occupa 4: quindi se non vi servono più di 16 milioni di valori usare un MEDIUMINT invece che un INT comporta un risparmio del 25%); inoltre cercate di dichiarare sempre le colonne NOT NULL, in modo da risparmiare lo spazio necessario all'indicatore dei valori NULL: quanto meno, dichiarate che una colonna può essere NULL solo se ne avete realmente bisogno;
- la chiave primaria di una tabella dovrebbe essere più corta possibile, per rendere più immediata l'identificazione di una riga
- gli indici (vedere **lez. 10**) sono il fattore forse più importante nell'ottimizzazione di una tabella: sono infatti indispensabili per avere letture veloci; in particolare, le colonne che fanno riferimento ad altre tabelle (chiavi esterne) e quelle utilizzate per le ricerche dalle query dovrebbero essere sempre indicizzate; tuttavia

bisogna considerare che la presenza di indici velocizza la lettura ma rallenta la scrittura (gli indici infatti vanno tenuti aggiornati), per cui è importante trovare il giusto equilibrio fra le due esigenze
- se dovete indicizzare campi di testo, sarebbe bene limitare il numero di caratteri inclusi nell'indice; se ad esempio avete un campo di 50 caratteri, ma già i primi 10 sono sufficienti ad avere un range di valori ben distinti fra loro, indicizzare solo questi 10 comporterà un rilevante risparmio sulle dimensioni dell'indice;
- quando dovete memorizzare dati binari (ad esempio immagini), è consigliabile salvarli su disco e non sul database, limitandosi ad inserire in tabella un riferimento al filesystem: questo dovrebbe consentire una maggiore velocità
- i dati che fanno parte di una tabella dovrebbero essere in **terza forma normale**, ci sono casi in cui può essere conveniente accettare ridondanze, quando questo comporta significativi miglioramenti nelle prestazioni

Un terzo livello di ottimizzazione, non meno importante degli altri, è quello che riguarda **l'accesso ai dati**: una query infatti può essere più o meno veloce (a volte con differenze anche notevoli), in base alla strategia scelta da MySQL per eseguirla.

Anche l'ottimizzazione delle query è un argomento piuttosto complesso. Se notate che alcune query sono piuttosto lente, un primo strumento utilizzabile per valutarle è la funzione BENCHMARK, usata dal client mysql:

SELECT BENCHMARK(100000,'*query*');

Questa istruzione ci permette di eseguire una query un numero arbitrario di volte: indicheremo tale numero come primo parametro e la query che vogliamo vedere come secondo parametro. Come risultato dell'istruzione non otterremo niente, ma il client MySQL ci mostra dopo ogni istruzione il tempo che ha impiegato ad eseguirla: in questo modo potremo valutare l'impatto di eventuali modifiche sulla struttura della query. In genere è necessario usare numeri piuttosto grandi (almeno 100.000, ma spesso anche maggiori) per avere tempi valutabili nell'ordine dei centesimi di secondo; ovviamente questo dipende dalla complessità della query e dalla velocità del processore che utilizziamo.

Il secondo passo da fare per valutare l'efficienza di una query è l'utilizzo della **EXPLAIN**, che ci permette di visualizzare i criteri utilizzati da MySQL per la sua esecuzione:

EXPLAIN [EXTENDED] SELECT ...

A **questa pagina** del manuale potete trovare una dettagliata spiegazione di come interpretare l'output di questa istruzione.
In generale, per avere query più veloci dovremo far sì che tutte le tabelle interessate

vengano lette attraverso gli indici, e non attraverso uno scorrimento completo (table scan); in alcune situazioni può capitare che MySQL non utilizzi un indice che pure esiste: a volte infatti considera più veloce scorrere la tabella, ad esempio perchè ritiene che la colonna indicizzata non abbia una quantità sufficiente di valori diversi. Può capitare però che questo avvenga perchè il server non ha statistiche aggiornate sul contenuto della tabella: possiamo allora aggiornarle eseguendo una **ANALYZE TABLE *nome_tabella***.

Un modo di "suggerire" a MySQL di utilizzare un indice è quello di aggiungere la clausola **FORCE INDEX *nome_indice*** di seguito al nome della tabella nella SELECT. Ricordiamo anche che quando confrontiamo in una query due campi indicizzati è bene che i due indici siano dello stesso tipo e della stessa lunghezza: questo permette al server di massimizzare le prestazioni.

Accorgimenti si possono utilizzare anche per velocizzare le operazioni di **inserimento** dei dati, in particolare quando dobbiamo inserire più righe alla volta.

Ad esempio:

- eseguire una sola INSERT per più righe, utilizzando VALUES multipli (vedere **lez. 13**);
- se si devono caricare dati da file di testo, utilizzare l'istruzione LOAD DATA INFILE (il guadagno di velocità è notevole);
- se un'elaborazione deve effettuare più di cinque inserimenti, precederli con un LOCK sulla tabella interessata (oppure includerli in una transazione se si usa uno storage engine transazionale); se il numero di inserimenti è molto elevato, sbloccare le tabelle ogni migliaio di righe inserite per consentire ad altri client di accedervi senza costringerli ad attese troppo lunghe.

Oltre a questi accorgimenti, dobbiamo tenere presente che quando effettuiamo numerosi aggiornamenti o cancellazioni su una tabella, lo spazio su disco occupato da questa tabella può diventare male organizzato (in particolare per le tabelle con righe a lunghezza variabile): è bene quindi eseguire periodicamente un'ottimizzazione della tabella attraverso l'istruzione **OPTIMIZE TABLE *nome_tabella***.

Con questo abbiamo concluso questa rapida carrellata sull'ottimizzazione: come abbiamo detto all'inizio, però, l'argomento è ben più complesso, per cui vi invitiamo ad approfondirlo sul manuale di MySQL.

Configurazioni Cluster

Fino ad ora sono state trattate singole installazioni del DBMS. In molti contesti professionali può essere utile predisporre più server MySQL che collaborino tra loro. Gli scopi di queste configurazioni possono essere molteplici: **alta disponibilità** del servizio, **disaster recovery**, bilanciamento del carico, **parallelizzazione** del lavoro.

In questa lezione, si affronterà la tematica vedendo da vicino le casistiche più comuni: la **Replication** e **MySQL Cluster**.

Replication

La Replication consiste nella duplicazione dello stesso database su server MySQL differenti. Ciò rappresenta la base di qualunque strategia di disaster recovery, evitando perdite di dati in caso di guasti hardware o software nonché di attacchi informatici o errori umani.

La forma più comune (ma non l'unica) di replication è la cosiddetta **master/slave**. Sfrutta due istanze del DBMS: una – il *master* – gestisce il database principale, l'altra – lo *slave* – ne contiene una copia.

La replication avverrà grazie ad un log binario che il master manterrà costantemente, registrandovi tutte le modifiche apportate ai dati. Il server slave verrà autorizzato a consultare questi log così da poter applicare le stesse modifiche al proprio database, la replica. Punto di partenza affinchè il tutto funzioni è che il contenuto dei due database – il master e lo slave – all'inizio sia assolutamente identico.

Nel seguito verrà mostrato un esempio di configurazione che può essere sperimentata anche su due macchine virtuali, purchè siano connesse in Rete e raggiungibili l'una dall'altra.

Per l'attivazione del master, sarà necessario modificare il file *my.cnf*. In una configurazione standard tali opzioni sono già incluse ma non attive in quanto commentate; pertanto, prima di riscriverle da zero, conviene verificarne la presenza. Le modifiche verteranno sull'assegnazione di un identificativo al server (*server-id*), l'attivazione di un log binario (*log_bin*), la disponibilità del server sull'indirizzo IP da noi scelto (*bind-address*) ed il nome del database da replicare (*binlog_do_db*):

```
server-id = 1
bind-address            = 192.168.100.2
log_bin                 = /var/log/mysql/mysql-bin.log
binlog_do_db            = database_da_replicare
```

Successivamente, si deve accedere al server tramite client *mysql* e creare un utente che verrà usato dagli slave per contattare il master:

```
> GRANT REPLICATION SLAVE ON *.* TO 'slavehost'@'%' IDENTIFIED BY 'secret';
> FLUSH PRIVILEGES;
```

Le righe precedenti hanno impostato un utente *slavehost* che può accedere da ogni indirizzo di rete tramite la password *secret*. Anche in questo caso tali impostazioni possono essere modificate a piacimento seguendo la documentazione relativa al comando `GRANT`.

A questo punto si deve riavviare il server MySQL per rendere effettive le modifiche.

Tramite il client *mysql*, digitiamo quindi il comando seguente:

```
> SHOW MASTER STATUS;
```

Se abbiamo svolto correttamente i pochi passi fin qui indicati, dovrebbe essere mostrata una tabella con alcuni dati: *file*, *position* e *Binlog_do_db*. Questi al momento non ci dicono molto (a parte forse il terzo, il nome del database da replicare) ma è bene prenderne nota in quanto serviranno tra poco per avviare la replica negli slave.

Per la **configurazione slave**, modifichiamo nel file *my.cnf* la configurazione di alcuni parametri:

```
server-id = 2
relay-log               = /var/log/mysql/mysql-relay-bin.log
log_bin                 = /var/log/mysql/mysql-bin.log
binlog_do_db            = database_da_replicare
```

Le opzioni sono le medesime modificate nel master, ad eccezione di *relay-log*, necessaria solo nello slave.

Riavviare anche in questo caso MySQL e, se non ci sono errori, si può aprire il client *mysql* e digitare il seguente comando:

```
> CHANGE MASTER TO MASTER_HOST='192.168.100.2',MASTER_USER='slave', MASTER_PASSWORD='sec

> START SLAVE;
```

I dati riportati nel comando sono quelli impostati nel master, e nello specifico:

- `MASTER_HOST`, `MASTER_USER` e `MASTER_PASSWORD`: rispettivamente indirizzo IP del

master, utente e password dell'account per la replication creato nel master;
- `MASTER_LOG_FILE` e `MASTER_LOG_POS`: sono i primi due dati letti nell'output del comando `SHOW MASTER STATUS` eseguito sul master.

Anche nello slave si può verificare l'attivazione delle funzionalità di replica con:

```
> SHOW SLAVE STATUS;
```

Svolti questi passi, potremo testare la coppia di server DBMS che lavorano in replica. La prova si potrà effettuare semplicemente modificando, inserendo o cancellando una riga nel database master e verificando se l'aggiornamento è stato svolto nello slave.

Ulteriori informazioni possono essere ricavate dalla documentazione ufficiale. Si consideri, comunque, che nella versione 5.5 del DBMS sono state eliminate tre opzioni di configurazione indipendenti, *master_host*, *master_user* e *master_password*, che venivano usate nello slave. Nonostante siano ripportate in molti tutorial on line, esse non vengono più accettate dalle versioni più moderne di MySQL. Sono state sostituite dall'uso di `CHANGE MASTER TO`, che abbiamo visto precedentemente.

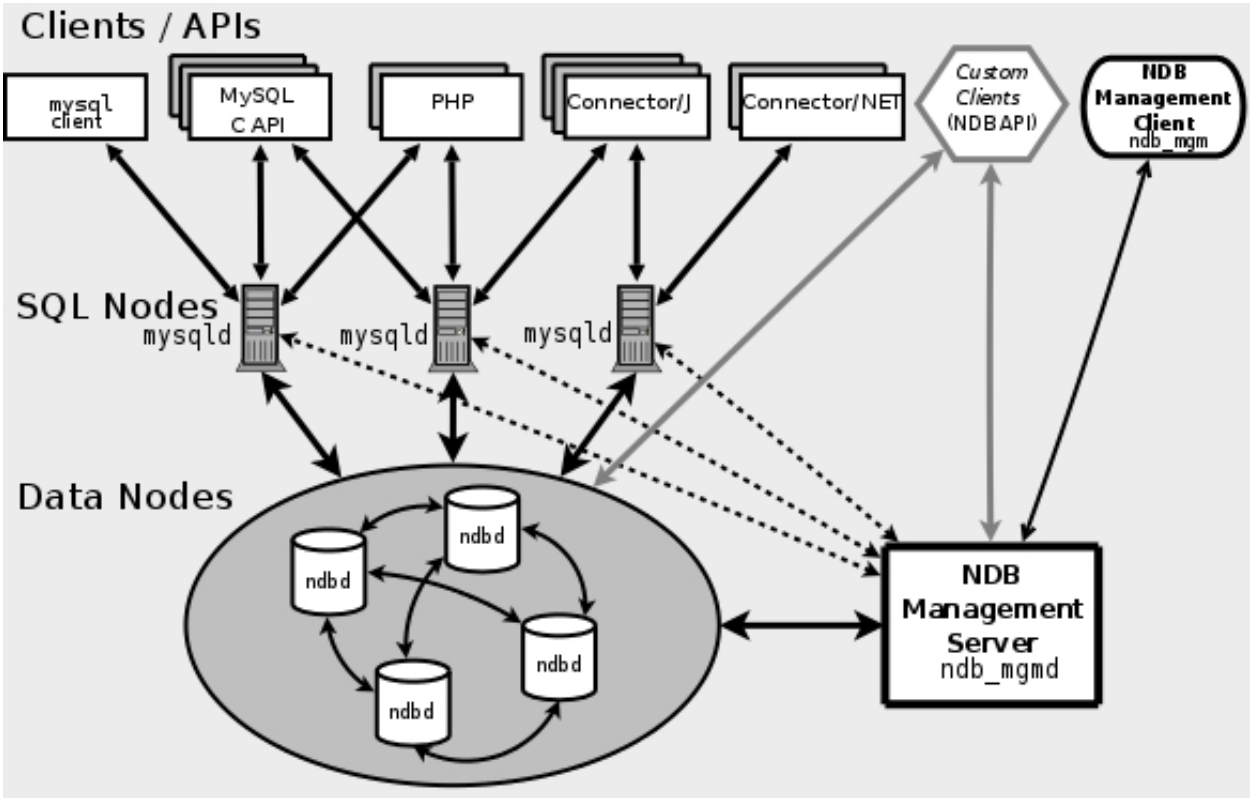
MySQL Cluster

MySQL Cluster permette di creare database transazionali mediante un'architettura distribuita in grado di:

- consentire la **scalabilità orizzontale**: aggiungendo nodi, la struttura risulta potenziata di conseguenza;
- sfruttare il cosiddetto commodity hardware, sistemi informatici di recupero e non molto costosi;
- fornire una **disponibilità del servizio "a cinque 9"**, come si indica in gergo un livello del 99,999%;
- gestire automaticamente i carichi di lavoro;
- garantire **maggiore sicurezza** grazie ad un'architettura distribuita priva di nodi vulnerabili.

Il progetto ha a disposizione un apposito Storage Engine, **NDB**, che oltre ad offrire caratteristiche avanzate simili a quelle di InnoDB (transazioni, granularità di lock a livello di riga, indici di vario genere) è l'unico pensato per supportare i Cluster.

L'architettura di un cluster MySQL può essere molto articolata, come mostra la figura seguente, tratta dalla documentazione ufficiale del DBMS:



Come si vede, la composizione è disposta su più strati:

- al centro vi è una sequenza di nodi SQL in cui viene eseguito il demone *mysqld*. Le applicazioni client vi faranno accesso come ad una normale installazione del DBMS. Si presti attenzione però che non si tratta del demone canonico *mysqld*, bensì di una versione modificata. Le due versioni sono incompatibili;
- il salvataggio di massa dei dati viene svolto sui Data Nodes. Qui NDB si occupa dell'archiviazione su disco;
- lo strato client è composto da tutte le applicazioni in grado di dialogare con MySQL, che interagiranno con l'architettura cluster in modo trasparente;
- il centro di gestione è NDB Management Server, che legge la configurazione del cluster e veicola la comunicazione tra i nodi fornendo così indicazioni utili al salvataggio e recupero dei dati. Mantiene inoltre dei log sull'attività del cluster. Lo stato dell'installazione può essere consultato tramite un NDB Management Client.

Per quanto riguarda l'installazione, possiamo scaricare MySQL Cluster tramite un'**apposita pagina del sito ufficiale**.

A partire dalla versione 7.3 è stato creato un **Auto-Installer**, un'applicazione fruibile tramite browser, che permette un'installazione guidata del cluster.

È utilizzabile pertanto su ogni sistema operativo e attraverso la maggior parte dei browser ma si deve prestare attenzione ai requisiti richiesti affinchè funzioni. Tra questi, in particolare, è necessario che l'host su cui viene eseguito l'Auto Installer disponga di una versione 2.6 o superiori del linguaggio Python e gli eventuali nodi

remoti coinvolti devono avere tutto il necessario affinchè si possa utilizzare una connessione criptata ed autenticata (SSH, HTTPS, certificati). Comunque, sempre sul sito ufficiale, è disponibile una **lista completa ed aggiornata dei requisiti** da soddisfare.

Per finalità di studio, sperimentazione o sviluppo, un cluster MySQL può essere installato in un unico nodo. Mentre, al contrario, per adempiere a tutte le finalità di tolleranza ai guasti e ridondanza dei dati, il numero di nodi ideale è di almeno 6 nodi (2 per i dati, 2 per l'applicazione SQL, 2 per la gestione).

Qualora non si volesse o non fosse possibile avviare l'Auto Installer, si può ricorrere ad un'installazione manuale le cui istruzioni sono reperibili a **questo link**.

Sommariamente, sarà richiesto di:

- preparare le macchine che formeranno il cluster approntando una corretta configurazione di sistema e di rete;
- scegliere per ognuna un ruolo da svolgere, ed in base a questo preparare il programma che dovrà essere utilizzato: *mysqld* per i nodi applicativi, *ndbd* per i data node (o *ndbmtd* nella versione multi-threaded), *ndb_mgmd* per il server di gestione e *ndb_mgm* per il management client;
- provvedere alla configurazione affinchè il sistema di gestione conosca i nodi che fanno parte del client.

Connectors e API: una panoramica

Un database server ovviamente non sarà utilizzato soltanto attraverso i suoi client, ma deve essere raggiungibile da applicazioni di vario genere. L'utilizzo più diffuso di MySQL è probabilmente quello di database per i siti e le applicazioni web; tuttavia è dotato di numerose interfacce applicative e connettori che lo rendono utilizzabile praticamente in qualsiasi contesto.

Cominceremo con una rapida carreggiata delle API, rimandandovi come al solito al **manuale** per una documentazione più dettagliata.

libmysqld (Embedded MySQL Server Library)

Una prima possibilità è quella di incorporare direttamente il server MySQL nell'applicazione. Ciò è possibile attraverso l'uso dell'apposita libreria *libmysqld*, ottenibile configurando MySQL con l'opzione *–with-embedded-server* (in fase di compilazione dei sorgenti). L'uso di questa libreria consente di ottenere una maggior velocità di esecuzione.

API C

Una delle interfacce applicative più ampiamente utilizzate è sicuramente quella per il linguaggio C. Si trova nella libreria **mysqlclient**. La maggior parte dei programmi client distribuiti con MySQL è scritta in C, quindi utilizza questa libreria per connettersi al server. Anche le altre API usano mysqlclient. Questa interfaccia consente l'utilizzo delle **prepared statements** (vedere oltre) e supporta anche l'esecuzione di query multiple (non con le prepared statements però).

API PHP

PHP è uno dei principali "compagni di viaggio" di MySQL, essendo un linguaggio diffusissimo per applicazioni e siti web dinamici. Esistono tre interfacce per PHP – *mysql*, *mysqli* e *PDO* – che verranno trattate in un capitolo successivo della guida.

API Perl

Per l'interfacciamento con Perl esiste l'interfaccia **Perl DBI**, basata sul modulo DBI (interfaccia generica verso i database) e sul driver **DBD::mysql**.
Questa interfaccia sostituisce la vecchia *mysqlperl*, da considerare ormai obsoleta.

API C++

Per il linguaggio C++ è disponibile un **Connector ufficiale**, compatibile anche con JDBC 4.0. Tuttavia è ancora utilizzabile un'ulteriore libreria, anch'essa molto diffusa:

MySQL++. Prima della nascita del Connector, MySQL++ era la principale opzione per l'interazione tra il DBMS e i software in C++, tanto da essere supportata dagli stessi sviluppatori di MySQL

API Python

Anche per il linguaggio Python è stato previsto un apposito Connector che si è affiancato alla libreria di terze parti *MySQLdb*. Documentazione in merito può essere reperita sulla **documentazione ufficiale di MySQL**.

API Tcl

Anche per Tcl è disponibile una semplice interfaccia chiamata **MySQLtcl**. La trovate a **questo indirizzo**.

API Eiffel

Per il linguaggio Eiffel potete trovare l'interfaccia **Eiffel MySQL** a **questo indirizzo**.

ODBC

Il **Connector/ODBC** (precedentemente chiamato *MyODBC*) offre l'accesso a MySQL mediante lo standard industriale **ODBC** (**O**pen **D**ata**B**ase **C**onnectivity), disponibile per diverse piattaforme ma ampiamente usato nel sistema Windows. Un suo tipico utilizzo infatti consiste nell'accedere a database disponibili su macchine Unix/Linux da piattaforme Microsoft. Dopo aver installato il Connector/ODBC è necessario configurare un DSN (Data Source Name), a patto che la macchina client disponga dei necessari permessi di accesso al server. Dettagliate informazioni possono essere ricavate sempre dalla **documentazione ufficiale**.

.NET

Per le applicazioni .NET è disponibile **MySQL Connector/NET**, un driver ADO.NET scritto in C#. Supporta tutte le novità introdotte da MySQL 4.1 in poi (stored procedures, prepared statements ecc.)

Java

Per le applicazioni Java, che utilizzano i driver JDBC, MySQL fornisce connettività attraverso **MySQL Connector/J**. Si tratta di driver conforme a JDBC-3.0 e JDBC-4.0 (a seconda delle versioni), ed è di "Tipo 4", ossia è scritto in Java e comunica direttamente col server attraverso il protocollo MySQL.

Prepared statements

Facciamo un rapido accenno anche alle prepared statements, che abbiamo citato poco fa e che sono uno strumento molto utile per l'utilizzo del database da parte delle applicazioni; si tratta di una delle tante novità introdotte da MySQL nella versione 4.1.

Le prepared statements sono un modo di eseguire le istruzioni SQL dichiarando prima al server la sintassi della query con dei "segnaposto" per i valori che l'istruzione riceverà di volta in volta; di seguito l'istruzione viene eseguita inviando al server solo i valori di cui sopra. Questo modo di lavorare offre notevoli vantaggi nel caso in cui una istruzione debba essere eseguita più volte con valori diversi: infatti il parsing della query avverrà una volta sola e ciò comporta una maggiore velocità; inoltre anche la quantità di dati che viaggiano di volta in volta dal client al server risulterà minore in questo caso.

Un altro vantaggio delle prepared statements è che offrono protezione dalle SQL injection, in quanto ogni segnaposto sarà sostituito sempre da un valore e non c'è quindi la possibilità che l'introduzione di valori "maligni" alteri la struttura della query.

Le prepared statements sono concepite per essere utilizzate con le interfacce applicative, ma esiste anche la possibilità di utilizzarle direttamente con il linguaggio SQL: ciò può essere utile qualora si vogliano fare delle verifiche sul loro funzionamento oppure non si disponga di un'interfaccia applicativa che le supporta (ad es. la vecchia estensione *mysql* di PHP).

Vediamo alcuni semplicissimi esempi:

```
PREPARE stmt1 FROM 'SELECT * FROM tabella1 WHERE campo1 = ? AND campo2 = ?';
SET @a=5;
SET @b='ora';
EXECUTE stmt1 USING @a, @b;
SET @a=8;
EXECUTE stmt1 USING @a, @b;
```

Con la prima istruzione prepariamo la query che dovrà estrarre i dati da tabella1 in base al valore delle colonne campo1 e campo2; utilizziamo il segnaposto (il punto interrogativo) per indicare che quelli saranno i valori variabili della query. Quando eseguiremo l'istruzione quindi, alla quale abbiamo dato il nome 'stmt1', dovremo passare tali valori.

Con la seconda istruzione assegniamo il valore 5 alla variabile @a e il valore 'ora' alla variabile @b quindi eseguiamo l'istruzione che avevamo preparato in precedenza, passando tali valori che verranno sostituiti ai punti interrogativi nella query. Dopo la prima esecuzione, modifichiamo il valore di @a ed eseguiamo nuovamente la query (il valore di @b è rimasto invariato).

Naturalmente è anche possibile preparare istruzioni senza alcun parametro: in questo caso potremo richiamarle omettendo la clausola USING. Le istruzioni che è possibile preparare sono CREATE TABLE, DELETE, DO, INSERT, REPLACE, SELECT, SET, UPDATE e la maggior parte delle SHOW.

PHP: API per l'accesso al DB

Una delle installazioni più comuni per lo sviluppo web è la cosiddetta piattaforma **LAMP** (**L**inux, **A**pache, **M**ySQL, **P**HP), che utilizza un server web Apache per ospitare applicativi scritti in PHP, i quali a loro volta sfruttano un database MySQL. In realtà, la stessa installazione può avere luogo su sistemi Windows ed un'interfacciamento tra PHP e MySQL può essere impiegato in programmi non destinati al web ma finalizzati ad altri scopi. In questa lezione vedremo quali sono e come usare le principali API per accedere ad un database MySQL.

Panoramica sulle API disponibili

Prima di iniziare il nostro percorso, è necessario fare chiarezza sulle API disponibili:

- *ext/mysql*: le API tradizionali, introdotte nella versione 2.0 del linguaggio e alla base di moltissimi applicativi tuttora al lavoro nel web. Sono deprecate dalla versione 5.5 di PHP;
- *ext/mysqli*: la versione più moderna delle API precedenti. Introdotte in PHP 5.0 sono ricche di funzionalità: basate sul paradigma della programmazione a oggetti, più efficienti, dispongono i *prepared statements* e supportano molte altre funzionalità del DBMS, come le **stored procedures** e **transazioni**;
- *PDO_MySQL*: è un'estensione che permette di fare interagire PHP con molti DBMS, non solo MySQL, per poter rendere il codice indipendente (e quindi riutilizzabile) rispetto alla tecnologia di memorizzazione dei dati che si usa.

La documentazione ufficiale del DBMS si esprime chiaramente: per i nuovi sviluppi è consigliato utilizzare l'estensione *mysqli* o *PDO*, mentre le API tradizionali *mysql* (essendo deprecate) dovrebbero essere utilizzate solo per necessità di manutenzione dei progetti più datati.

La diffusione delle vecchie API è così estesa anche nella cultura degli sviluppatori attuali, e tale problematica è così sottovalutata, che la documentazione del linguaggio PHP dedica una pagina apposita alla **scelta delle API**, confrontando le tre tecnologie sopra citate.

In questa sede vedremo, in brevi esempi, tutte queste API: *mysqli* e *PDO* come scelte

primarie, e per completezza anche le funzioni dell'estensione *mysql*.

Codice di esempio

Tutti gli esempi presentati nel seguito mostrano le stesse fasi:

- **connessione al DBMS**;
- un'operazione di **inserimento**;
- l'**esecuzione di una query**, con successiva lettura dei dati prelevati.

Lo script PHP avrà bisogno di collegarsi al database fornendo l'indirizzo di rete, credenziali di accesso di un account MySQL valido (username e password) ed il nome del database stesso.

Assumeremo, per comodità, che i dati appena indicati siano resi disponibili, in ognuno dei successivi esempi, con le seguenti variabili, da valorizzare in base alla propria configurazione:

```
$host = "...";

$user = "...";

$pass = "…";

$dbname = "…";
```

Il database conterrà una tabella sola, *persone*, la cui struttura può essere prodotta nel seguente modo:

```
CREATE TABLE `persone` (

  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,

  `nome` varchar(50) NOT NULL,

  `cognome` varchar(50) NOT NULL,

  `eta` tinyint(3) unsigned NOT NULL,

  PRIMARY KEY (`id`)

) ENGINE=InnoDB;
```

Usare mysqli

```
$my = new mysqli($host, $user, $pass,$dbname);

if ($my->connect_errno) {

    echo "Errore in connessione al DBMS: ".$my->error;

        exit();
```

```
} else {


    $nome="Carlo";

    $cognome="Verdi";

    $eta=25;

    $stmt = $my->prepare("INSERT INTO persone(cognome, nome, eta) VALUES (?, ?, ?)");

    $stmt->bind_param('ssi',$nome,$cognome,$eta);

    $stmt->execute();

    echo "Ultimo ID generato: ".$my->insert_id;


    $query="SELECT * FROM persone";

    $r=$my->query($query);

    while ($row = $r->fetch_assoc())

    {

        printf("ID: %s  Name: %s", $row["id"],$row["cognome"]." ".$row["nome"]);

    }

    $r->close();

}
```

L'avvio all'interazione con il DBMS si ottiene con un oggetto di classe `mysqli`. Viene istanziato passando al costruttore i parametri di connessione e le credenziali di accesso.

Appena inizializzato, controlliamo se la connessione è avvenuta con successo, verificando il valore della proprietà `connect_errno`: in caso di errore, essa ne conterrà il codice numerico; se invece l'operazione è stata svolta con successo, sarà valorizzata `0`.

Se il nostro script ha avuto accesso al DBMS, procediamo con l'inserimento, che avviene mediante *prepared statement*:

- la query viene preparata scrivendo in SQL tutta la sua parte fissa, mentre i parametri variabili sono rimpiazzati da punti interrogativi (`?`). Tutto questo tramite il metodo `prepare`;
- si usa il metodo `bind_param` per passare i parametri reali. Ne saranno accettati tanti quanti sono i punti interrogativi nello statement. Come primo argomento viene passata una stringa contenente un flag per ogni parametro: i flag specificano il tipo di dato del valore: `s` per string, `i` per integer, `d` per double e `b`

per blob;
- dopo il binding dei parametri, si passa all'esecuzione vera e propria, tramite il metodo `execute`.

Considerato che la tabella ha una chiave primaria intera generata in automatico con autoincremento, si potrà leggere l'ultimo `id` creato con il metodo `insert_id`.

Per le interrogazioni, si usa il metodo `query` cui si passa un comando `SELECT` completo. I risultati, nell'esempio, vengono letti iterativamente con `fetch_assoc` che per ogni record crea un array associativo, quindi consultabile usando i nomi dei campi come chiavi.

Oltre a `fetch_assoc`, esiste anche il metodo `fetch_array` che permette di ottenere ogni record non solo in modalità associativa ma anche in formato numerico, nonchè in entrambi i formati contemporaneamente. La scelta verrà indicata dal programmatore al momento di invocare il metodo tramite il parametro in input, da scegliere tra `MYSQLI_NUM` (numerico), `MYSQLI_ASSOC` (associativo), `MYSQLI_BOTH` (entrambi).

Un aspetto interessante di *mysqli* consiste nella possibilità di attivare le stesse funzionalità in modalità procedurale, usando quindi delle funzioni piuttosto che l'approccio a oggetti. Il primo dei due, dall'aspetto più "tradizionale", potrebbe risultare più familiare a programmatori PHP non del tutto convertiti ancora al modello ad oggetti. Una comparazione è approfonditamente discussa in un'**apposita pagina del manuale online di PHP**.

PDO

```
try {

    $conn = new PDO("mysql:host=$host;dbname=$dbname", $user, $pass);



    $nome="Carlo";

    $cognome="Verdi";

    $eta=25;

    $stmt = $conn->prepare("INSERT INTO persone(cognome, nome, eta) VALUES (:cognome,
:nome, :eta)");

    $stmt->bindParam(':cognome', $cognome, PDO::PARAM_STR);

    $stmt->bindParam(':nome', $nome, PDO::PARAM_STR);

    $stmt->bindParam(':eta', $eta, PDO::PARAM_INT);

    $stmt->execute();

    echo "Ultimo ID generato: ".$conn->lastInsertId();
```

```
    $query = "SELECT * FROM persone";
    foreach ($conn->query($query) as $row) {
    print $row['id'] . "\t";
        print $row['nome'] . "\t";
        print $row['cognome'] . "\n";
    }
}
catch(PDOException $e)
{
    echo $query . "<br>" . $e->getMessage();
}
```

PDO, a differenza di *mysqli* non è espressamente orientato a MySQL ma è costituito da classi generiche potenzialmente adattabili a qualunque DBMS.

Nel codice precedente si istanzia un oggetto `PDO`, il cui costruttore riceve in input una stringa di connessione che specifica il tipo del DBMS, indirizzi di rete dell'istanza cui connettersi e credenziali di accesso.

Il codice PHP che ne deriva può essere riutilizzato indipendentemente dal DBMS a supporto. L'unico requisito è che esista un driver apposito che faccia da interprete tra PDO ed il tipo di database scelto.

Anche in questo esempio si è svolto un inserimento e subito dopo una query. La prima operazione ha richiesto l'uso di un *prepared statement*, come spiegato nel paragrafo precedente per *mysqli*. Si è pertanto invocato il metodo `prepare` che predispone un comando generico contenente la struttura base dell'`INSERT` che si vuole realizzare. Successivamente, con `bindParam`, si specificano i singoli parametri da utilizzare, per ognuno dei quali deve essere indicato il tipo di dato (in questo caso: `PDO::PARAM_STR` per le stringhe e `PDO::PARAM_INT` per i numeri interi). Infine, con `execute` si esegue lo statement.

L'interrogazione viene eseguita con il comando `query` ed il risultato è direttamente fruibile tramite un ciclo `foreach`. Come si vede ogni record potrà essere letto specificando, in modalità associativa, i nomi dei campi.

Estensione mysql

Le funzioni che fanno parte dell'estensione *mysql* di PHP sono tuttora impiegate in tantissimi applicativi.

```
$r = mysql_connect($host, $user, $pass);

if (!$r) {

    echo "Could not connect to server\n";

    exit();

} else {

    mysql_select_db($dbname);


    $insert="INSERT INTO persone(cognome, nome, eta) VALUES ('Carlo', 'Verdi', 25)";

    mysql_query($insert);

    echo "Ultimo ID generato: ".mysql_insert_id();


    $query="SELECT * FROM persone";

    $r=mysql_query($query);

    while ($row = mysql_fetch_array($r, MYSQL_ASSOC))

    {

        printf("ID: %s  Name: %s", $row["id"],$row["cognome"]." ".$row["nome"]);echo
"\n";

    }

    mysql_free_result($r);

}

mysql_close();
```

Come si vede, in questo caso l'approccio è totalmente procedurale. L'esempio mostra delle funzioni che ricordano molto i metodi impiegati nel precedente esempio di *mysqli*:

- `mysql_connect`: permette di connettersi al DBMS utilizzando le credenziali indicate;
- `mysql_select_db`: indica quale database si vuole utilizzare tra quelli a disposizione del profilo di connessione;
- `mysql_query`: offre la possibilità di inviare sia comandi di modifica che normali interrogazioni;
- `mysql_fetch_array`: consente di visionare uno alla volta i record ottenuti tramite la query. Il parametro passato in input indica che l'accesso avverrà in maniera associativa: ogni record verrà fornito sotto forma di array, le cui chiavi avranno lo stesso nome dei campi della tabella;
- `mysql_free_result`: libera la memoria dalle risorse utilizzate dai risultati della query;

- `mysql_close`: chiude la connessione.

Come spiegato nella documentazione ufficiale e già ricordato all'inizio di questa lezione, tali funzioni sono state deprecate a partire da PHP 5.5, e quindi non dovrebbero più essere utilizzate in nuovi progetti.

Java: API per l'accesso al DB

Su Java è possibile accedere ad un database MySQL tramite il protocollo JDBC, uno standard industriale indipendente dal DBMS impiegato. Il suo utilizzo permette di impostare il codice in maniera che rispetti il principio, caro al mondo Java, del *Write Once, Run Anywhere*: un software riutilizzabile al di là delle tecnologie e dei contesti utilizzati inizialmente.

Affinchè possa interagire con JDBC, un DBMS deve offrire appositi driver che implementino le classi generiche che lo costituiscono. MySQL mette a disposizione un connector per il linguaggio Java, chiamato **Connector/J** e **reperibile sul sito ufficiale di MySQL**. Lo si può scaricare in formato sorgente o binario, ma quest'ultima opzione risulta più comoda, dal momento che richiederà il semplice inserimento di un archivio *.jar* nel *CLASSPATH* del proprio progetto.

L'utilizzo di JDBC con MySQL verrà illustrato mediante un esempio. Ci si occuperà per prima cosa del caricamento del driver e della gestione della connessione. Successivamente si svolgeranno due operazioni: un inserimento tramite `PreparedStatement` ed una query che leggerà il contenuto di una tabella.

Assumeremo di accedere ad un database con i seguenti parametri di connessione:

Parametro	Valore
indirizzo di rete	`localhost`
porta TCP	`3306` (il valore di default)
username	`root`
password	`secret`
nome del database	`Contatti`

Si farà accesso ad una sola tabella, *persone*, la cui struttura può essere prodotta come segue:

```
CREATE TABLE `persone` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `nome` varchar(50) NOT NULL,
  `cognome` varchar(50) NOT NULL,
  `eta` tinyint(3) unsigned NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

Driver e stringa di connessione

Prima di vedere il codice in azione, è bene soffermarsi sull'esecuzione della

connessione.

Una volta inserito il driver nel proprio CLASSPATH, è necessario che il codice Java ne carichi a runtime la classe principale, denominata `com.mysql.jdbc.Driver`. Ecco come:

```
try
{
    Class.forName("com.mysql.jdbc.Driver");
}
catch (ClassNotFoundException e)
{
    // gestione dell'eccezione
}
```

Nel caso il driver non sia disponibile, verrà sollevata un'eccezione di tipo `ClassNotFoundException`.

L'interazione tra il programma ed il DBMS si baserà su una connessione aperta, rappresentata da un oggetto di classe `Connection`.

I parametri saranno tutti inclusi in una stringa di connessione, che nel nostro esempio sarà la seguente:

```
String connectionString="jdbc:mysql://localhost:3306/Contatti?user=root&password=secret'
```

Il suo formato ricorda quello di un indirizzo di rete. Tipicamente inizia con il prefisso `jdbc`, seguito da un altro termine che indica il DBMS utilizzato. Il resto è una concatenazione dell'indirizzo del database (il formato è *`indirizzo_host`*`:`*`porta_TCP`*`/`*`nome_database`*) e di una *querystring* (ciò che segue il punto interrogativo) costituita dai parametri di accesso. Indichiamo solo username e password dell'account MySQL che utilizziamo, ma la documentazione ufficiale descrive tutte le ulteriori opzioni disponibili.

La classe `DriverManager` userà la stringa di connessione per aprire la connessione:

```
Connection connection;
...
...
connection = DriverManager.getConnection(connectionString);
```

È importante notare che i parametri specifici del DBMS utilizzato si evincono per lo più dalla connection string e dal driver invocato. Se si volesse **riutilizzare il codice** con un altro tipo di database, sarebbe sufficiente modificare soltanto queste due stringhe.

L'esempio

```
String connectionString = "...";

try {
        Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException e) {
        e.printStackTrace();
}

Connection connection = null;

try {
        connection = DriverManager.getConnection(connectionString);
        PreparedStatement prepared = connection.prepareStatement("insert into persone (c
        prepared.setString(1, "Marroni");
        prepared.setString(2, "Enrico");
        prepared.setInt(3, 55);

        prepared.executeUpdate();

        Statement stm = connection.createStatement();
        ResultSet rs = stm.executeQuery("select * from persone");
        while (rs.next()) {
                System.out.println(rs.getString("cognome") + " " + rs.getString("nome")
        }

} catch (SQLException e) {
        e.printStackTrace();
} catch (Exception e) {
        System.out.println(e.getMessage());
} finally {
        try {
                if (connection != null)
                        connection.close();
        } catch (SQLException e) {
                // gestione errore in chiusura
        }
}
```

Dando per assodati i concetti relativi alla connessione, illustrati nel paragrafo precedente, vediamo da vicino le due operazioni eseguite sulla tabella.

Per prima cosa, dobbiamo ottenere uno `Statement` dalla connessione. Questo rappresenta il comando da eseguire, e sarà configurato in primis con il comando SQL ed eventualmente con altri parametri.

Per l'inserimento, utilizziamo la modalità dei `PreparedStatement`, già introdotti in una **lezione precedente**.

Tramite il metodo `prepareStatement`, creaiamo un oggetto di questo tipo a partire da un comando SQL in cui i punti interrogativi sostituiranno i parametri da utilizzare. Si invoca poi il metodo `executeUpdate`, dal momento che lo Statement mira ad apportare modifiche ai dati.

Per la query utilizziamo uno Statement comune e l'interrogazione viene formulata tramite un classico comando `SELECT`. In questo caso avremo bisogno di leggere i risultati ottenuti dall'esecuzione del comando `executeQuery`, raccolti in un oggetto `ResultSet`.

Quest'ultimo contiene un cursore che punta ai dati presenti in un determinato record dei risultati. Per fruirne sarà necessario spostarlo da un record all'altro in base alle nostre esigenze, ed una volta posizionato leggere i valori dei campi ivi presenti.

Il nostro scopo, per semplicità, sarà quello di stampare l'intero set dei risultati; pertanto passiamo da un record al successivo tramite il metodo `next()`, invocato nel ciclo `while`.

Ci si può muovere anche in altre direzioni sfruttando metodi come `previous` (torna al precedente), `first` (spostamento al primo record) o `last` (spostamento all'ultimo record).

I dati di un record potranno essere letti tramite metodi specifici per ogni tipo di dato: nel codice abbiamo utilizzato `getString` e `getInt` per accedere, rispettivamente, a stringhe e numeri interi. Esistono anche `getFloat` e `getDouble` (numeri in virgola mobile), `getDate` e `getTime` (per informazioni temporali) e diversi altri metodi simili, indicati nella **documentazione**.

Infine la connessione va chiusa. Considerando che tutte le operazioni sono state incluse in un blocco `try/catch` che gestisce le eccezioni di tipo `SQLException` (errori di interazione con il database), la chiusura della connessione dovrebbe essere inserita in un conseguente blocco `finally`, in modo da essere eseguita in ogni caso, sia di successo che si insuccesso.

Oltre JDBC

Il protocollo JDBC permette di svolgere qualunque interazione con il DBMS. Il suo svantaggio è che si tratta di un approccio "di base", che può richiedere la scrittura di molto codice, spesso piuttosto ripetitivo.

Esistono per questo altre soluzioni, sempre basate su JDBC, che permettono un approccio più flessibile e mantenibile per l'interazione Java-MySQL. Se ne nominano di seguito i principali:

- **Spring Framework e template JDBC**: il framework Spring è molto usato nella gestione di grandi progetti Java. Tra le sue numerosissime funzionalità, esso offre anche i *JdbcTemplate* che facilitano il lavoro del programmatore svolgendo in autonomia operazioni delicate come apertura e chiusura di connessioni e gestione delle risorse, nonché la preparazione degli Statement, la lettura di risultati e l'attivazione di transazioni;
- **Hibernate e altri OR/M**: il filone degli OR/M, di cui Hibernate è un illustre rappresentante, propone un approccio diverso per l'accesso ai dati, basato meno sull'invio diretto dei comandi SQL, quanto piuttosto fondato sul mapping tra sistema informativo relazionale (incluso nel database) e modello a oggetti realizzato nel programma Java;
- **iBatis**: un framework dedicato alla persistenza dei dati che facilita l'integrazione tra il programma e il database.

C#: API per l'accesso al DB

Nonostante le tecnologie Microsoft dispongano di soluzioni proprie per la persistenza dei dati (si pensi a SQL Server, ma anche ad Access), l'integrazione dei linguaggi del framework .NET con MySQL non è rara.

A tale scopo esiste un pacchetto, un Connector, offerto da MySQL per l'integrazione con applicativi .NET. In questa lezione vedremo proprio come utilizzare il **Connector/NET** su C#.

Connector/NET: download ed installazione

Connector/NET può essere scaricato tramite l'**apposita pagina/** del sito ufficiale di MySQL. È disponibile nelle versioni per Mono (piattaforma .NET utilizzabile su sistemi Unix-like), Windows o come codice sorgente e, per ognuno di questi casi, sono fornite le appropriate istruzioni di installazione.

Focalizzandoci sull'**installazione per Windows**, esistono due modi per utilizzare il driver. Al primo si è già accennato in una lezione di questa guida: l'installazione di MySQL su Windows porta con sé una serie di tool accessori, tra cui proprio i Connectors per la programmazione.

Altrimenti, lo si può comunque scaricare in modalità standalone, separatamente dal DBMS, come pacchetto in formato *.msi*.

L'esempio

Nel seguito vedremo come utilizzare le classi fornite con Connector/NET.

Il server MySQL utilizzato come controparte avrà i seguenti parametri di connessione:

Parametro	Valore
Indirizzo di rete	`localhost`
Porta TCP	`3306` (il valore di default)
Username	`root`
Password	`secret`
Database	`Contatti`

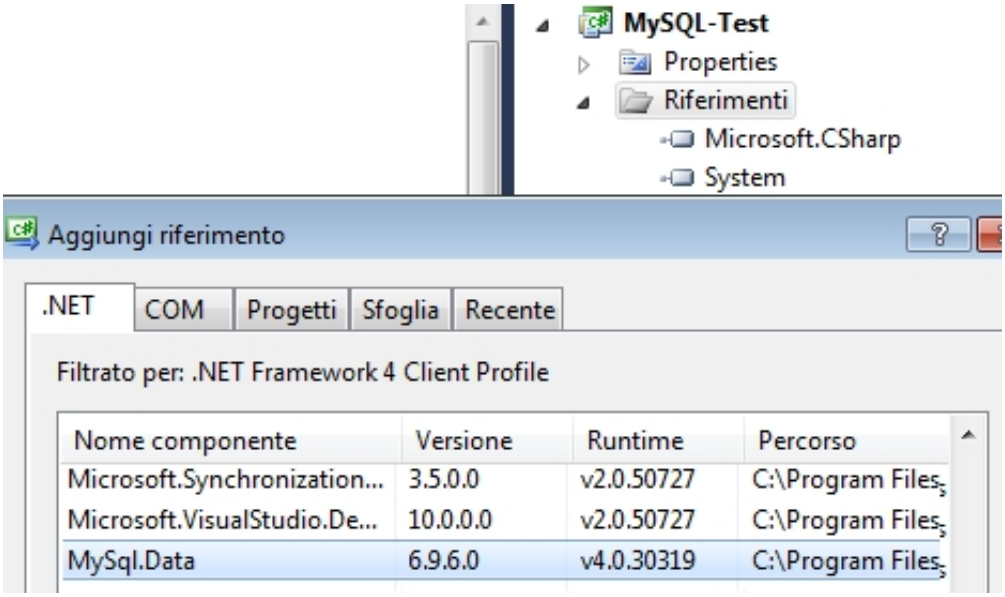
Infine, la tabella che manipoleremo sarà così definita:

```
CREATE TABLE `persone` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `nome` varchar(50) NOT NULL,
  `cognome` varchar(50) NOT NULL,
```

```
  `eta` tinyint(3) unsigned NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB;
```

Il codice

Dopo l'installazione del Connector, dovremo renderlo disponibile nel nostro progetto. Supponendo di utilizzare Visual Studio, l'IDE fornito da Microsoft per il framework .NET, dovremo inserire un riferimento alle classi, come mostrato nella figura seguente:



L'esempio che segue apre una connessione dal programma C# al server MySQL e svolge un'operazione di inserimento tramite Prepared Statement, ed una query che legge e stampa in output il contenuto della tabella.

La connessione, stabilita tramite oggetto `MySqlConnection`, è definita mediante una connection string i cui parametri hanno il seguente significato:

Parametro	Descrizione
Server	Indirizzo IP a cui risponde il server MySQL
Uid	Username dell'account MySQL
Password	Password dell'account MySQL
Database	Nome del database a cui collegarsi

```
MySqlConnection conn = new MySqlConnection("SERVER=localhost;database=Contat
MySqlCommand cmd = new MySqlCommand();

try
{
    conn.Open();

    String insert = "INSERT INTO persone (cognome,nome,eta) VALUES (@nome,@c

    cmd.Connection = conn;
```

```
        cmd.CommandText = insert;
        cmd.Prepare();

        cmd.Parameters.AddWithValue("@nome", "Nunzio");
        cmd.Parameters.AddWithValue("@cognome", "Neri");
        cmd.Parameters.AddWithValue("@eta", 28);

        cmd.ExecuteNonQuery();

        String query = "SELECT * FROM persone";
        MySqlCommand qrycmd = new MySqlCommand(query, conn);
        MySqlDataReader reader = qrycmd.ExecuteReader();
        while (reader.Read())
        {
            Console.WriteLine(reader["cognome"] + " " + reader["nome"]);
        }
    }
    catch (MySqlException e)
    {
        // gestione dell'eccezione
    }
    finally
    {
        if (conn!=null)
            conn.Close();
    }
```

L'esecuzione dei comandi si snoda lungo le seguenti di fasi:

- connessione al database;
- definizione del comando da eseguire;
- eventuale lettura dei risultati dell'operazione svolta;
- chiusura della connessione.

La connessione deve essere aperta prima dell'utilizzo con il metodo `Open`, e chiusa al termine delle operazioni con `Close`. Una sua proprietà interna, `State`, fornisce informazione sullo stato attuale della connessione.

Che si voglia eseguire una operazione di modifica o un'interrogazione, è comunque necessario istanziare un oggetto `MySqlCommand`. Un oggetto di questo tipo, per agire, ha bisogno di almeno due elementi: un comando SQL (definito nella proprietà `CommandText`) ed una connessione tramite la quale comunicare con il DBMS (definita nella proprietà `Connection`).

Il comando relativo all'inserimento viene svolto come Prepared Statement, pertanto si assegnano le proprietà relative al comando SQL e alla connessione, e viene invocato il metodo `Prepare`.

I parametri vengono definiti in SQL con un nome simbolico preceduto da `@`: ad esempio, `@nome`.

Al momento dell'esecuzione della query devono essere assegnati i parametri alla proprietà `Parameters` tramite il metodo `AddWithValue`.

Nel caso della query, invece, non si usa un Prepared Statement, perciò il comando SQL e la connessione vengono passati direttamente tramite costruttore.

Comunque venga definito un `MySqlCommand`, affinchè svolga le operazioni per le quali è stato configurato se ne deve ordinare l'esecuzione. Questi i metodi che lo permettono:

- `ExecuteNonQuery`: utilizzato per i comandi di modifica, fornisce come risultato il numero di righe coinvolte. Nell'esempio lo usiamo per eseguire l'`INSERT`;
- `ExecuteReader`: usato per le query di selezione, restituisce un cursore forward-only e read-only;
- `ExecuteScalar`: ritorna un solo valore presente alla prima colonna della prima riga, mentre tutti gli altri dati risultati vengono ignorati. Tipicamente è usato con operazioni tipo `SELECT COUNT` e simili, che forniscono un solo valore.

Il risultato della query viene letto tramite un oggetto `MySqlDataReader`, che rappresenta il cursore che legge i dati. Il suo utilizzo prevede lo scorrimento in avanti, una riga alla volta, ad ogni invocazione del metodo `Read`. La riga raggiunta può essere letta semplicemente specificando il nome di un campo della tabella tra parentesi quadre.

Usare ADO.NET

Il codice visto nel paragrafo precedente ha il difetto di utilizzare classi espressamente orientate a MySql. Se si volesse utilizzare lo stesso stralcio di C# nell'interazione con un altro DBMS lo si dovrebbe modificare radicalmente. Una soluzione alternativa esiste, ed è basata su **ADO.NET**, il sottosistema di accesso ai dati offerto dal .NET Framework.

ADO.NET offre una serie di vantaggi tra cui l'uniformità nell'accesso ai dati in termini di classi utilizzate, tipi di dato, metodi di progettazione.

In pratica, permette di scrivere del codice generico, riutilizzabile con qualsiasi DBMS che offra un driver da integrare con ADO.NET. Il Connector/NET di MySQL nasce perfettamente compatibile con ADO.NET, il che permette di utilizzarlo come driver.

Per fare ciò è necessario sostituire tutti i riferimenti alle classi del Connector con le corrispondenti classi astratte di ADO.NET, e lo svolgimento del codice seguirà il medesimo percorso:

- connessione: useremo `DbConnection` anziché `MySqlConnection`;
- esecuzione del comando: ci sarà `DbCommand`, non più `MySqlCommand`;
- lettura dei risultati: il cursore sarà gestito con `DbDataReader` al posto di `MySqlDataReader`.

La connection string avrà lo stesso formato, ma la connessione sarà ottenuta in una maniera diversa:

```
DbProviderFactory factory = DbProviderFactories.GetFactory("MySql.Data.MySqlClient");
DbConnection conn = factory.CreateConnection();
```

`DbProviderFactory` rappresenta un oggetto che permette la creazione di una connessione di un tipo specifico, indicato nel parametro di `GetFactory` ma gestito tramite riferimenti a classi generiche. In un codice così impostato la stringa `MySql.Data.MySqlClient` è l'unico elemento che svela l'uso di MySQL: la sua sostituzione permetterebbe il riutilizzo integrale del codice C# con un altro DBMS.

A titolo di esempio, quello che segue è il codice che permette lo svolgimento della query:

```
String query = "SELECT * FROM persone";
DbCommand qrycmd = conn.CreateCommand();
qrycmd.CommandText = query;
qrycmd.Connection = conn;
DbDataReader reader = qrycmd.ExecuteReader();
while (reader.Read())
{
    Console.WriteLine(reader["cognome"] + " " + reader["nome"]);
}
```

Esempio: struttura del DB di un albergo

Dopo avere fatto un'ampia carrellata su MySQL, chiudiamo questa guida con un breve tutorial dedicato ad un ipotetico database per la gestione delle prenotazioni di un piccolo albergo.

Si tratta di un argomento che presenta una certa complessità, per cui non avremo certamente la pretesa di esplorarlo in maniera esauriente o – tanto meno – professionale. Tutto quello che vogliamo fare è disegnare un piccolo database con qualche query di esempio.

Per rendere le cose un po' meno complesse di come sarebbero in realtà, faremo qualche assunzione un pochino semplicistica sul modo in cui il nostro albergo viene gestito. Diciamo quindi che si tratta di un piccolo hotel in una località marittima, con le stanze distribuite su quattro piani; ci sono due stanze singole, due doppie, sette matrimoniali e due triple; tutte le stanze matrimoniali hanno la possibilità di aggiungere un terzo letto, e tre di queste, più grandi, possono ospitarne anche un quarto; anche le due stanze triple hanno la possibilità di aggiungere il quarto letto.

Prevediamo anche alcune caratteristiche che diversificano le camere: alcune hanno un televisore, alcune hanno la vista sul mare, alcune hanno l'aria condizionata e, infine, alcune sono riservate ai fumatori. Tuttavia, come abbiamo detto prima vogliamo semplificare un po' le cose, per cui stabiliamo che queste caratteristiche non incidono sul prezzo della camera, che dipende invece esclusivamente dal periodo e dal numero di posti letto. Prevediamo anche un costo fisso, indipendente dalla stagione, per i letti aggiunti e per altri supplementi come la culla in camera; infine decidiamo che il trattamento standard fornito dal nostro hotel è la mezza pensione, per cui chiederemo un supplemento per la pensione completa e concederemo una riduzione per chi vuole solo bed & breakfast; anche queste differenze saranno indipendenti dal periodo stagionale.

Naturalmente la prima cosa da fare sarà creare il database, che chiameremo 'hotel':

CREATE DATABASE hotel DEFAULT CHARACTER SET latin1 DEFAULT COLLATE latin1_general_ci; USE hotel;

Stabiliamo quindi che il database userà il charset dei caratteri latini con la relativa collation generica; dopo averlo creato usiamo il comando USE per farlo diventare il database di default per le istruzioni successive.

Per prima cosa prevediamo una semplicissima tabella in cui registrare i dati dei **clienti**:

```
CREATE TABLE `clienti` (
  `id` mediumint unsigned NOT NULL auto_increment,
  `nominativo` varchar(100) NOT NULL,
  `indirizzo` varchar(200) NOT NULL,
  `telefono` varchar(15) NOT NULL,
  PRIMARY KEY  (`id`)
) ENGINE=InnoDB;
```

Come vedete, memorizziamo in questa tabella solo i dati essenziali: nominativo, indirizzo e telefono; usiamo un id autoincrementante come chiave della tabella.

Avremo poi una tabella relativa alle **camere**:

```
CREATE TABLE `camere` (
  `numero` smallint unsigned NOT NULL,
  `piano` tinyint unsigned NOT NULL,
  `tipo` enum('singola','doppia','matrimoniale','tripla') NOT NULL,
  `lettiAggiunti` set('terzo','quarto') NOT NULL,
  `optionals` set('fumatori','ariaCondizionata','vistaMare','tv')
               NOT NULL,
   PRIMARY KEY  (`numero`)
) ENGINE=InnoDB;
```

Utilizziamo quindi il numero della camera come chiave primaria; oltre al piano a cui è situata la stanza, registriamo il tipo di camera in un campo di tipo enum, elencando i possibili valori che questo può assumere; inoltre utilizziamo due campi di tipo set per segnalare gli eventuali letti che possono essere aggiunti e le caratteristiche opzionali della stanza; l'uso dei campi set in questo caso è ovviamente dovuto al fatto che più di uno dei valori elencati possono essere utilizzati contemporaneamente.

Avremo poi la tabella contenente i **prezzi** per ogni tipo di camera, relativamente ai diversi periodi:

```
CREATE TABLE `prezzi` (
  `periodoDal` date NOT NULL,
  `periodoAl` date NOT NULL,
  `tipoCamera` enum('singola','doppia','matrimoniale','tripla')
               NOT NULL,
  `prezzo` decimal(6,2) NOT NULL
) ENGINE=InnoDB;
```

In questa tabella utilizziamo due campi di tipo date per indicare l'inizio e la fine del periodo di validità di ogni tariffa; abbiamo poi un campo per il tipo di camera, equivalente a quello della tabella camere, e infine un campo di tipo decimal per il prezzo: stiamo un po' larghi e prevediamo quattro cifre intere e due decimali.

Prevediamo ora una piccola tabellina per i **prezzi dei supplementi**, che, come abbiamo detto, sono indipendenti dalle stagionalità:

```
CREATE TABLE `supplementi` (
  `codice` tinyint unsigned NOT NULL auto_increment,
  `voce` char(20) NOT NULL,
  `prezzo` decimal(5,2) NOT NULL,
  PRIMARY KEY  (`codice`)
) ENGINE=InnoDB;
```

In questa tabella usiamo un id autoincrementante (piccolo perchè i valori saranno pochi), un campo per descrivere la voce e uno per il prezzo; da notare che questo campo potrà avere valori negativi, in quanto in alcuni casi (ad esempio il trattamento di soggiorno e prima colazione) non si tratta di supplementi ma di riduzioni.

Passiamo ora alle **prenotazioni**, che memorizzeremo nella seguente tabella:

```
CREATE TABLE `prenotazioni` (
  `id` mediumint unsigned NOT NULL auto_increment,
  `periodoDal` date NOT NULL,
  `periodoAl` date NOT NULL,
  `camera` smallint unsigned NOT NULL,
  `idCliente` mediumint unsigned NOT NULL,
  `prezzoTotale` decimal(7,2) NOT NULL,
  PRIMARY KEY  (`id`),
  KEY `camera` (`camera`),
  KEY `idCliente` (`idCliente`)
) ENGINE=InnoDB;
```

Anche in questo caso usiamo un identificativo autoincrementante, seguito dalle date di inizio e fine soggiorno, dai riferimenti a camera e cliente (che sono chiavi esterne sulle rispettive tabelle), e dal prezzo totale della prenotazione; per favorire i collegamenti con le tabelle dei clienti e delle camere abbiamo indicizzato anche i campi relativi.

L'ultima tabella del nostro database ci serve per memorizzare i **supplementi relativi a ciascuna prenotazione**. Si tratta di una relazione 1:N, nel senso che ad ogni prenotazione possono corrispondere più supplementi (o riduzioni): ad esempio la culla, il terzo letto, la pensione completa ecc.; questo è il motivo per cui è necessaria una tabella distinta da quella delle prenotazioni, in base alle regole della normalizzazione.

```
CREATE TABLE `supplementi_prenotati` (
  `idPrenotazione` mediumint unsigned NOT NULL,
  `codiceSupplemento` tinyint unsigned NOT NULL
) ENGINE=InnoDB;
```

Come vedete qui non abbiamo altro che i riferimenti alla tabella delle prenotazioni e a quella dei supplementi.

Provvediamo ora, prima di passare agli esempi sulle query, a riempire le tabelle relative alle camere e ai prezzi:

INSERT INTO `camere` VALUES
(101, 1, 'singola', '', 'ariaCondizionata'),
(102, 1, 'doppia', '', 'ariaCondizionata,vistaMare,tv'),
(103, 1, 'doppia', 'terzo', 'ariaCondizionata,vistaMare,tv'),
(104, 1, 'matrimoniale', 'terzo,quarto', 'ariaCondizionata'),
(201, 2, 'matrimoniale', 'terzo', 'ariaCondizionata,vistaMare,tv'),
(202, 2, 'matrimoniale', 'terzo', 'ariaCondizionata,vistaMare,tv'),
(203, 2, 'matrimoniale', 'terzo,quarto', 'ariaCondizionata'),
(301, 3, 'matrimoniale', 'terzo', 'ariaCondizionata,vistaMare,tv'),
(302, 3, 'matrimoniale', 'terzo', 'ariaCondizionata,vistaMare,tv'),
(303, 3, 'matrimoniale', 'terzo,quarto', 'ariaCondizionata'),
(401, 4, 'singola', '', 'fumatori'),
(402, 4, 'tripla', 'quarto', 'fumatori,vistaMare'),
(403, 4, 'tripla', 'quarto', 'fumatori');

INSERT INTO `prezzi` VALUES
('2006-05-01', '2006-05-31', 'singola', '50.00'),
('2006-05-01', '2006-05-31', 'doppia', '90.00'),
('2006-05-01', '2006-05-31', 'matrimoniale', '90.00'),
('2006-05-01', '2006-05-31', 'tripla', '130.00'),
('2006-06-01', '2006-06-30', 'singola', '55.00'),
('2006-06-01', '2006-06-30', 'doppia', '95.00'),
('2006-06-01', '2006-06-30', 'matrimoniale', '95.00'),
('2006-06-01', '2006-06-30', 'tripla', '140.00'),
('2006-07-01', '2006-07-31', 'singola', '65.00'),
('2006-07-01', '2006-07-31', 'doppia', '120.00'),
('2006-07-01', '2006-07-31', 'matrimoniale', '120.00'),
('2006-07-01', '2006-07-31', 'tripla', '160.00'),
('2006-08-01', '2006-08-31', 'singola', '80.00'),
('2006-08-01', '2006-08-31', 'doppia', '150.00'),

(‘2006-08-01’, ‘2006-08-31’, ‘matrimoniale’, ‘150.00’),
(‘2006-08-01’, ‘2006-08-31’, ‘tripla’, ‘200.00’),
(‘2006-09-01’, ‘2006-09-30’, ‘singola’, ‘50.00’),
(‘2006-09-01’, ‘2006-09-30’, ‘doppia’, ‘90.00’),
(‘2006-09-01’, ‘2006-09-30’, ‘matrimoniale’, ‘90.00’),
(‘2006-09-01’, ‘2006-09-30’, ‘tripla’, ‘130.00’);

INSERT INTO `supplementi` VALUES
(1, ‘culla’, ‘8.00’),
(2, ‘letto aggiuntivo’, ‘30.00’),
(3, ‘uso singola’, ‘-15.00’),
(4, ‘bed&breakfast’, ‘-10.00’),
(5, ‘pensione completa’, ‘5.00’);

Ora possiamo passare all’utilizzo del nostro database!

Esempio: interrogare il DB di un albergo

Vedremo in questa lezione come utilizzare il database che abbiamo impostato nella lezione precedente.

Innanzitutto decidiamo di creare una **stored function** per il calcolo del prezzo di una camera, dato il periodo di interesse e il tipo di camera:

```
DELIMITER //
CREATE FUNCTION prezzo(arrivo date, partenza date,
        tipo enum('singola','doppia','matrimoniale','tripla'))
RETURNS DECIMAL(7,2)
READS SQL DATA
BEGIN
        DECLARE varData DATE;
        DECLARE varTotale DECIMAL(7,2) default 0;
        DECLARE varPrezzo DECIMAL(7,2);
        SET varData = arrivo;
        WHILE varData < partenza DO
                SELECT prezzo INTO varPrezzo FROM prezzi
                WHERE varData BETWEEN periodoDal AND periodoAl
                AND tipoCamera = tipo;
                SET varTotale = varTotale + varPrezzo;
                SET varData = DATE_ADD(varData,INTERVAL 1 day);
        END WHILE;
        return varTotale;
END; //
DELIMITER ;
```

Come vedete la funzione, che si chiama ‘prezzo’, accetta come parametri in input le date di arrivo e partenza e il tipo di camera, che ricalca lo stesso tipo di campo presente nella tabella relativa.

Passiamo ora alla fase delle prenotazioni: il problema più importante da risolvere è quello di cercare sul database le disponibilità in base alle prenotazioni già presenti e alle richieste che riceviamo. Prima di tutto però dobbiamo accertarci che, nel momento in cui andiamo ad effettuare una prenotazione, nessun altro possa intervenire sulle tabelle, per cui dovremo effettuare un LOCK. Quindi effettueremo la query di ricerca.

Immaginiamo che ci sia stata richiesta la disponibilità di una camera matrimoniale per la settimana dal 3 al 10 giugno:

```
LOCK TABLES camere c READ, prenotazioni p READ,
        supplementi READ, prezzi READ, clienti WRITE,
        prenotazioni WRITE, supplementi_prenotati WRITE;
SET @inizioPeriodo = '2006-06-03';
SET @finePeriodo = '2006-06-10';
```

```
SELECT c.* FROM camere c
WHERE tipo = 'matrimoniale'
AND NOT EXISTS
(SELECT * FROM prenotazioni p WHERE p.camera = c.numero
AND (p.periodoDal < @finePeriodo and @inizioPeriodo < p.periodoAl)
);
```

La ricerca di disponibilità viene effettuata attraverso una **subquery correlata** che, per ogni camera del tipo che ci interessa, va a verificare che non esistano prenotazioni che si intersecano con il periodo richiesto (per verificare le intersezioni bisogna confrontare le date di inizio con quelle di fine periodo). Questa query ci restituirà l'elenco delle camere che hanno disponibilità nel periodo richiesto, dandoci la possibilità di scegliere quale assegnare al cliente che ha effettuato la richiesta.

Ipotizziamo ora di scegliere la camera 201, e che il cliente abbia richiesto il trattamento di pensione completa, nonchè la presenza della culla in camera. Dovremo quindi inserire i dati del cliente in tabella, poi registrare la prenotazione calcolandone il prezzo totale:

```
INSERT INTO clienti SET
        nominativo = 'Rossi Mario',
        indirizzo = 'via Manzi, 2 - 00153 Roma',
        telefono = '06 86123920';
SELECT LAST_INSERT_ID() INTO @codCliente;
```

Inseriti i dati del cliente in tabella, utilizziamo la funzione LAST_INSERT_ID() per recuperare l'identificativo e assegnarlo ad una variabile, quindi procediamo a calcolare il prezzo totale:

```
SELECT prezzo(@inizioPeriodo,@finePeriodo,'matrimoniale') +
(SELECT SUM(prezzo * DATEDIFF(@finePeriodo,@inizioPeriodo))
FROM supplementi WHERE codice IN (1,5))
INTO @prezzoTotale;
```

Questa query effettua la somma di due valori: il primo è il risultato della chiamata alla stored function *prezzo* che abbiamo definito in precedenza, che definisce il prezzo base della camera, mentre il secondo calcola il prezzo dei supplementi leggendo dalla tabella omonima i valori relativi ai codici 1 e 5 (culla e pensione completa) e moltiplicandoli per i giorni di permanenza, dati dalla differenza fra le due date (in questa seconda query la keyword 'prezzo' si riferisce alla colonna della tabella supplementi). Il risultato complessivo va nella variabile @prezzoTotale, che conterrà quindi il valore 756, dato dal prezzo della matrimoniale (95 * 7) più i supplementi pari a (8+5)*7.

Possiamo quindi ora utilizzare tale valore per inserire la prenotazione:

```
INSERT INTO prenotazioni VALUES (null, @inizioPeriodo,
        @finePeriodo, 202, @codCliente, @prezzoTotale);
SELECT LAST_INSERT_ID() INTO @idPrenotazione;
INSERT INTO supplementi_prenotati VALUES
        (@idPrenotazione,1),
        (@idPrenotazione,5);
UNLOCK TABLES;
```

Abbiamo così terminato il ciclo della prenotazione con lo sblocco delle tabelle che avevamo bloccato in precedenza. Come vedete nell'inserimento sulla tabella prenotazioni abbiamo usato *null* al posto della colonna id che riceve il valore auto_increment.

Avrete forse notato che nella istruzione iniziale di blocco delle tabelle abbiamo citato due volte la tabella prenotazioni. Ciò accade perchè nella query di ricerca delle disponibilità la tabella viene citata con l'alias p, mentre in fase di inserimento ciò non avviene (e nemmeno sarebbe possibile): di conseguenza siamo costretti ad acquisire il lock sia per l'alias, sia per il nome completo. Come possibilità alternativa avremmo potuto rinunciare all'uso dell'alias nella prima query; in questo caso sarebbe stato sufficiente il blocco in scrittura.

Naturalmente questa era una simulazione studiata per essere eseguita con il client mysql; in una realtà applicativa le query verranno eseguite attraverso le interfacce proprie del linguaggio utilizzato, e i dati che qui abbiamo salvato nelle variabili SQL verranno più probabilmente esportati nello script che gestisce la prenotazione.

Inoltre sarebbe stato possibile, utilizzando ad esempio lo storage engine InnoDB invece di MyISAM, effettuare il tutto all'interno di una transazione invece di usare i lock sulle tabelle. A questo proposito però è bene notare che, per garantirsi dal rischio che elaborazioni concorrenti potessero inserire prenotazioni nell'arco di tempo necessario alla nostra elaborazione, un sistema transazionale avrebbe dovuto lavorare con il massimo livello di isolamento, cioè SERIALIZABLE.

Concludiamo facendo alcune considerazioni ulteriori. La query di ricerca delle disponibilità può naturalmente essere arricchita con l'indicazione delle eventuali caratteristiche richieste per la stanza. Immaginiamo ad esempio di voler cercare la disponibilità di una camera doppia, per il periodo dall'8 al 22 luglio, con aria condizionata e vista mare, e la possibilità di un terzo letto aggiunto. La query sarebbe diventata così:

```
SELECT c.* FROM camere c
```

```
WHERE tipo = 'doppia'
AND FIND_IN_SET('ariaCondizionata',optionals) > 0
AND FIND_IN_SET('vistaMare',optionals) > 0
AND FIND_IN_SET('terzo',lettiAggiunti) > 0
AND NOT EXISTS
(SELECT * FROM prenotazioni p WHERE p.camera = c.numero
AND (p.periodoDal < '2006-07-22' and '2006-07-08' < p.periodoAl)
);
```

Vediamo ora quale sarebbe la query da effettuare per avere la lista delle camere disponibili un determinato giorno:

```
SELECT * FROM camere c WHERE NOT EXISTS
(SELECT * FROM prenotazioni p WHERE p.camera = c.numero
AND (p.periodoDal <= '2006-07-15'
     AND p.periodoAl > '2006-07-15' ));
```

Abbiamo usato l'operatore 'minore o uguale' per la data iniziale e solo 'maggiore' per la data finale in quanto consideriamo la stanza libera nella giornata in cui un soggiorno termina. Ovviamente basterebbe invertire la clausola NOT EXISTS con EXISTS per avere l'elenco delle stanze occupate.

Concludiamo con un'ultima query che ci restituisce l'elenco delle stanze occupate un determinato giorno con il nome dell'ospite di ciascuna ed il giorno in cui la stanza sarà liberata:

```
SELECT c.numero, c.piano, cl.nominativo,
p.periodoAl AS partenza FROM camere c
JOIN prenotazioni p ON p.camera = c.numero
LEFT JOIN clienti cl ON p.idCliente = cl.id
WHERE p.periodoDal <= '2006-07-15'
AND p.periodoAl > '2006-07-15'
```

Come vedete abbiamo effettuato una join fra tre tabelle (camere, prenotazioni e clienti), usando il numero di camera per relazionare le camere con le prenotazioni ed il codice cliente per legare la prenotazione all'ospite; la LEFT JOIN usata in quest'ultimo caso ci garantisce che una stanza che risulta occupata venga inclusa nel risultato anche se, per qualche motivo, il codice cliente memorizzato sulla tabella prenotazioni dovesse essere inesistente sulla tabella clienti. Ovviamente però questo sarebbe il segnale che qualcosa non ha funzionato nella nostra applicazione.

MariaDB

Nonostante MySQL resti il database relazionale Open Source più diffuso al mondo, dopo l'acquisizione nel 2008 di MySQL AB, azienda fondatrice del progetto, da parte di Sun Microsystems, sono nati diversi fork di MySQL che hanno assunto una notevole importanza nel panorama open source: i principali sono **MariaDB** e **Percona Server**.

Il primo dei due è stato fondato direttamente da Michael Widenius, uno dei padri di MySQL; il secondo è un prodotto di Percona, società informatica californiana esperta nel settore dei database.

In questa lezione ci concentreremo su MariaDB, alternativa molto diffusa che inizia a trovare molto spazio anche tra i software a corredo delle più famose distribuzioni Linux.

MariaDB: confronto con MySQL

Ci si potrebbe chiedere perchè mai si dovrebbe abbandonare MySQL in favore di MariaDB e quali caratteristiche si potrebbero trovare in più o in meno in quest'ultimo DBMS. Non è un caso se il team di MariaDB ha predisposto un'**apposita pagina della propria documentazione** per risolvere questi dubbi; quella che segue, comunque, è una sintesi dei punti principali.

Il primo aspetto su cui soffermarsi sono gli **Storage Engine** utilizzati. MariaDB include tutti i principali Storage Engine di MySQL, non senza crearne di propri. Tra questi, **Aria** ha un ruolo molto importante e rappresenta un'evoluzione di MyISAM in grado di supportare anche le transazioni.

È supportato anche **XtraDB** (ormai diventato lo Storage Engine di default per MariaDB), originariamente sviluppato da Percona come miglioramente di InnoDB.

La versione 10 del DBMS ha visto inoltre l'introduzione di nuove tipologie di tabelle, tra cui **Connect** (accesso a sorgenti dati esterne o remote in vari formati: Excel, JSON, XML e molti altri), **Spider** (supporta il partizionamento in modo da utilizzare dati presenti su più istanze di MariaDB insieme), **Sequence** (offre la possibilità di generare un elenco di numeri interi in ordine ascendente o discendente specificando valore di partenza e di fine nonché la modalità di incremento).

Un aAspetto interessante e molto moderno è l'introduzione di Storage Engine di tipo **NoSQL**, come ad esempio **Cassandra**.

Inoltre, MariaDB punta molto al **miglioramento delle prestazioni**, un obiettivo inseguito sfruttando vari filoni di sviluppo: aumentando il numero di connessioni

possibili, aggiungendo tipologie di indici o incrementando la velocità di ricerca di quelli esistenti, introducendo o ampliando meccanismi di cache di dati a supporto dei lavori di elaborazione, rendendo disponibili le cosiddette "colonne virtuali" automaticamente calcolate al momento del bisogno in base ad una funzione deterministica.

Iniziare ad usare MariaDB

Possiamo scaricare MariaDB dalla **pagina ufficiale del progetto**, sotto forma di codice sorgente o binario. Ormai, comunque, questo DBMS è ospitato nei repository di tutte le principali distribuzioni Linux: Ubuntu e Debian ma anche Fedora, CentOS, Mint e altre ancora.

Per facilitare l'installazione tramite repository, è inoltre disponibile una **pagina** che, nel giro di pochi click, permette all'utente di ottenere tutte le istruzioni di accesso ai repository di MariaDB per il proprio sistema operativo.

Le distribuzioni di MariaDB si dividono in due serie diverse: la 10.x.x e la 5.5.

Il motivo di ciò è che inizialmente MariaDB, in quanto figlio di MySQL, ha seguito gli stessi numeri di versione di quest'ultimo, fino alla 5.5. Successivamente, la mancata integrazione di alcune caratteristiche di MySQL 5.6 – considerate instabili – e, al contrario, l'ampliamento di MariaDB con nuove feature originali ha giustificato il significativo avanzamento del numero di versione a 10, rappresentando un deciso cambio generazionale.

Oltre al DBMS, nelle serie 5.5 e 10, dalla pagina dei download è possibile scaricare il progetto Cluster legato a MariaDB, denominato **MariaDB Galera Cluster**, nonchè una serie di librerie client e connector per la programmazione con **Java**, **C** e **ODBC**.

Per mantenere la compatibilità con il progenitore MySQL, il demone che esegue il server di MariaDB si chiama tuttora **mysqld**.

Diverse modalità di installazione di MariaDB configureranno il suo avvio automatico al boot del sistema operativo. Comunque lo si può avviare anche in modo più tradizionale:

- avvio diretto del demone *mysqld*;
- tramite la modalità controllata di *mysqld_safe*;
- con lo script *mysql.server*, usato in sistemi in stile System V.

Anche il programma client che verrà utilizzato risponde al nome di *mysql*. La modalità di accesso sarà quella nota:

```
mysql -u root -p
```

ma il prompt che ci accoglierà dopo l'autenticazione dimostrerà che stiamo accedendo ad un'istanza di MariaDB:

```
MariaDB [(none)] >
```

Dove ora è indicato `none` verrà collocato il nome del database che sceglieremo di utilizzare.

La configurazione del server viene custodita anche in questo caso all'interno del file *my.cnf* che, in base all'installazione specifica può essere frazionato in più collocazioni tra cartelle di sistema e home dell'utente.

Una volta approntata l'installazione di MariaDB, per il suo utilizzo è possibile applicare le conoscenze che si sono acquisite su MySQL in questa guida in materia di linguaggio SQL, gestione dei dati, strutturazione dei database. Ovviamente ciò che farà la differenza sarà l'apprendimento dei plugin e delle nuove caratteristiche che MariaDB offre rispetto al progetto di origine.

mysqld e i tool di base

L'installazione di MySQL porta con sè un gran numero di programmi che riguardano tutte le attività principali di amministrazione del DBMS. Si tratta, in genere, di strumenti a riga di comando sebbene, come vedremo in seguito, esistono diversi tool visuali altrettanto completi.

In questa lezione presenteremo una panoramica complessiva degli strumenti, per poi proseguire con approfondimenti specifici.

Panoramica degli strumenti

Come molti altri DBMS, MySQL viene eseguito come servizio o, in altre parole, come *daemon*. Un servizio o demone è un programma in esecuzione continua nel sistema operativo, il cui compito è quello di rimanere in attesa di richieste finalizzate alla fruizione di determinate funzionalità. Nel caso dei DBMS, tutto lo scambio di dati con il demone avrà come scopo la gestione dei database.

Il demone alla base del DBMS prende il nome di **mysqld**. Insieme ad esso vengono forniti altri programmi per l'avvio del server: *mysqld_safe*, pensato per un avvio più sicuro del server, e *mysqld_multi*, che permette l'avvio di più server installati nel sistema.

Tra gli altri strumenti messi a disposizione, dedicati a varie attività dello sviluppatore e dell'amministratore del DBMS, vi sono:

- **mysql**: il client ufficiale per interagire con i database. Verrà trattato in seguito;
- **mysqladmin**, per lo svolgimento di ogni genere di operazione di configurazione del server;
- **mysqlcheck**, che si occupa della manutenzione delle tabelle;
- **mysqldump**, indispensabile per il backup. Anch'esso verrà trattato nel corso della guida;
- **mysqlimport**, che permette di importare tabelle nei database;
- **mysqlshow**, che fornisce informazioni su database, tabelle, indici e molto altro.

Il server: avvio, arresto e controllo

Nelle installazioni più comuni, *mysqld* viene avviato in automatico all'avvio del sistema. Possiamo verificare che esso sia in esecuzione con *mysqladmin*, che possiede un'apposita funzionalità di ping:

```
mysqladmin -u root -p ping
```

Con questa istruzione, verrà richiesto da riga di comando, su qualunque sistema operativo, di effettuare un ping sul servizio. L'opzione *-u* specifica che la richiesta è fatta come utente *root*, amministratore del DBMS, mentre *-p* impone la richiesta di password per l'utente. Fornendo i corretti dati di autenticazione, se il DBMS è attivo, sarà stampato il messaggio "*mysql is alive*"; se invece non è attivo, verrà sollevato un errore di connessione.

Un altro modo per controllare lo stato del demone è verificare la sua presenza tra i processi in esecuzione. Su Windows lo si potrà fare con il Task Manager, mentre su Linux sarà solitamente sufficiente verificare l'output prodotto dal comando:

```
ps aux | grep mysql
```

Il risultato mostrato dovrebbe presentare più righe, di cui una contenente il percorso al programma *mysqld*.

Qualora il server non fosse in esecuzione, per avviarlo sarà necessario ricorrere ad uno degli strumenti predisposti appositamente per questo scopo, come *mysqld_safe*. In molte installazioni attuali, il DBMS viene predisposto tra i servizi di sistema. In questi casi il modo migliore per avviarlo o arrestarlo è utilizzare le apposite interfacce. Ad esempio, su Windows, si può utilizzare lo strumento *Servizi di sistema* accessibile tramite il Pannello di Controllo. Tra gli altri servizi sarà presente anche quello relativo a MySQL e sarà sufficiente verificarne lo stato e modificarlo. Anche sui sistemi Linux si possono avere opportunità simili. Ad esempio, nelle distribuzioni che usano il meccanismo *Upstart* come Ubuntu, Fedora ed altre, la consueta installazione del DBMS permette di avviare, arrestare e verificare lo stato da riga di comando, rispettivamente, con i seguenti comandi:

```
service mysql start
service mysql stop
service mysql status
```

Il client: interagire con MySQL

Per poter svolgere operazioni sui dati, il tipico client è il programma da riga di comando *mysql*.

Per prima cosa dobbiamo farci riconoscere. Le opzioni più comunemente utili in questo caso sono:

Opzione	Descrizione
`-u` o `--user`	Per introdurre il nome utente

`-p` o `--password`	Per richiedere l'accesso con password, che sarà richiesta a meno che non venga fornita come parametro. Si faccia attenzione al fatto che, usando la forma `-p`, la password deve essere concatenata all'opzione (esempio: `-ptopolino`) mentre nella forma estesa si deve usare il simbolo `=` (esempio: `--password=topolino`), che risulta preferibile in quanto più leggibile
`-h` o `--host`	Consente di specificare l'indirizzo IP o il nome della macchina su cui è in esecuzione il DBMS; il valore di default è `localhost`
`-P` o `--port`	Consente di specificare la porta TCP
`-D` o `--database`	Indica il nome del database cui si vorrà fare accesso

Un esempio di autenticazione tramite il comando `mysql` potrebbe quindi essere:

```
mysql -h mysql.mioserver.com -u dbadmin -p
```

In risposta, verrà richiesto di fornire la password. Si avrà così accesso all'istanza in esecuzione all'indirizzo *mysql.mioserver.com*, come utente *dbadmin*.

Dopo il login, ci troveremo di fronte il prompt di MySQL. I primi comandi utili da imparare sono:

Comando	Descrizione
`quit`	Per uscire dal programma *mysql*
`show databases;`	Per vedere i database accessibili tra quelli gestiti dal DBMS
`use`	Seguito dal nome del database, specifica su quale DB si vuole lavorare

Nel prompt di MySQL, come vedremo, potranno essere eseguiti anche comandi e query SQL. In questo caso non va dimenticato che il comando dovrà essere concluso da un punto e virgola.