

**itconsulting**  
Consulenza & Formazione

**SQL e Dintorni**

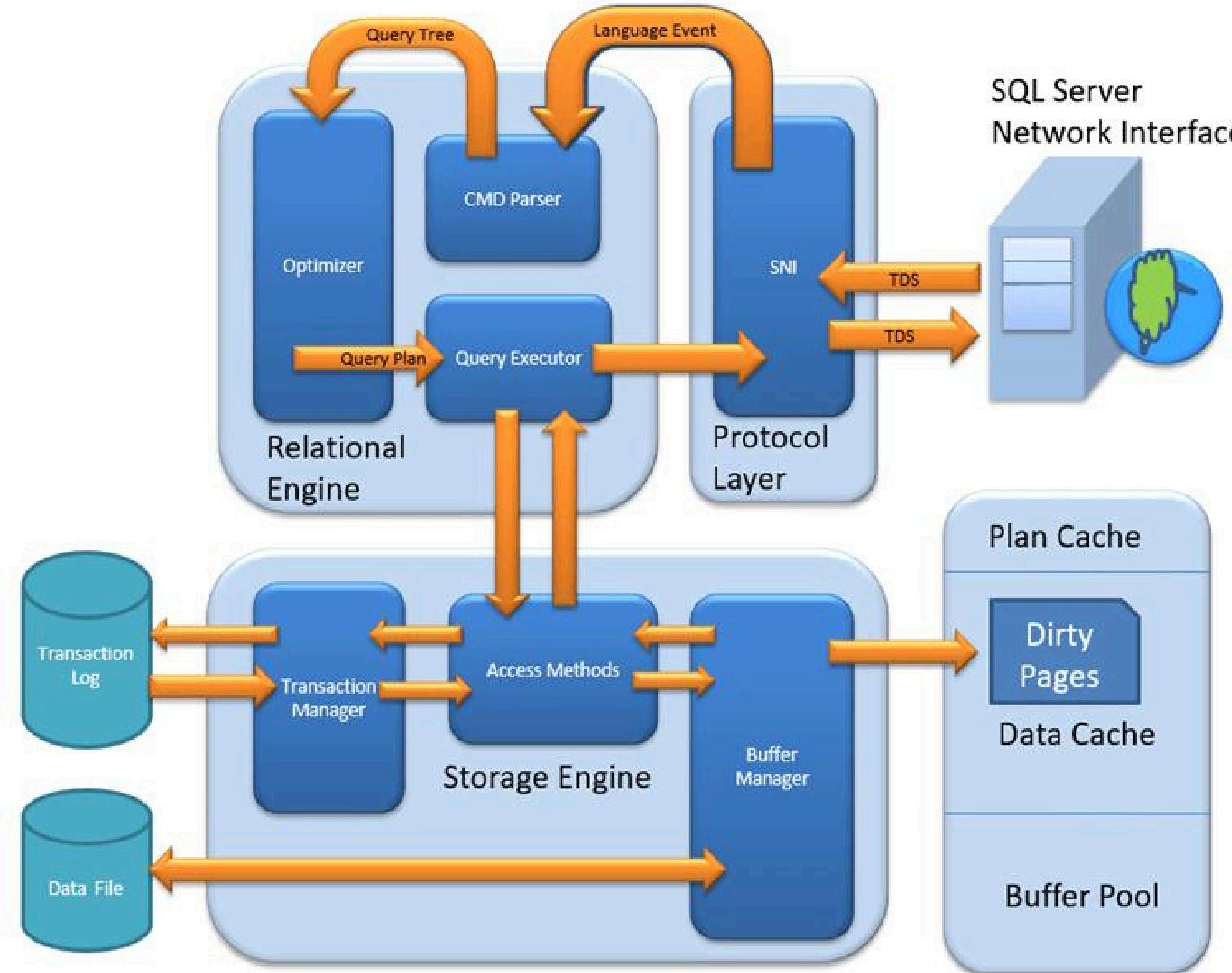
# **SQL (Structured Query Language)**



**Structured Query Language (SQL) è un linguaggio di interrogazione (query) utilizzato per creare, modificare e gestire i dati in un database relazionale.**

**Si tratta di un linguaggio specifico di dominio (DSL) usato per comunicare con i sistemi di gestione di database relazionali (RDBMS).**

# SQL (Structured Query Language)



# SQL: COSA CI SERVE

Useremo un DBMS(database management system) ovvero un sistema software progettato per consentire la creazione, la manipolazione e l'interrogazione efficiente di database, per questo detto anche "gestore o motore del database".

Nello specifico useremo MySQL che si basa sul modello relazionale quindi un RDBMS(relational database management system).

Cosa faremo :

- Scaricheremo e installeremo MySQL
- Andremo a vedere alcune connotazioni basilari di MySQL

# INSTALLIAMO MySQL

**Andiamo ad installare MySQL community (non la versione web)  
al seguente link:**

**<https://dev.mysql.com/downloads/installer/>**

**MySQL Installer 8.0.32**

Select Operating System:

Microsoft Windows

Looking for previous GA versions?

Windows (x86, 32-bit), MSI Installer	8.0.32	2.4M	Download
(mysql-installer-web-community-8.0.32.0.msi)		MD5: 0f882590f8338adc614e9dc5cb00ca0b   Signature	
Windows (x86, 32-bit), MSI Installer	8.0.32	437.3M	Download
(mysql-installer-community-8.0.32.0.msi)		MD5: a29b5817cba2c7bc0e0b97e897c2591f   Signature	

**!** We suggest that you use the MD5 checksums and GnuPG signatures to verify the integrity of the packages you download.



**(non è obbligatorio creare un account per l'installazione)**

# INSTALLIAMO MySQL

**Proseguiamo avviando l'installer scaricato, eseguiamo i seguenti passaggi:**

- 1. selezioniamo Developer Default (per semplicità), andiamo avanti**
- 2. risolviamo le richieste manuali, facciamo il check e andiamo avanti**
- 3. facciamo execute per avvia l'installazione, una volta finita l'installazione andiamo avanti**
- 4. passiamo alla configurazione, rimaniamo tutto default e andiamo avanti fino all'inserimento della password scegliendo una password semplice come "root" per semplificarci i passaggi durante il corso**
- 5. continuiamo ad andare avanti nella configurazione, se non si vuole far avviare i servizi di MySQL all'avvio di windows allora bisogna rimuovere la spunta su "start the MySQL server at system startup" in windows service**
- 6. continuiamo a rimanere le configurazioni di default fino all'ultimo step, facciamo execute per applicare la configurazione di MySQL Server.**
- 7. andiamo avanti rimanendo i valori di default fino alla configurazione del server example dove dobbiamo inserire la password scelta prima e fare il check, andiamo avanti fino all'execute**
- 8. andiamo a concludere l'installazione proseguendo fino all'ultimo step avviando la shell e la workbench.**

# MySQL



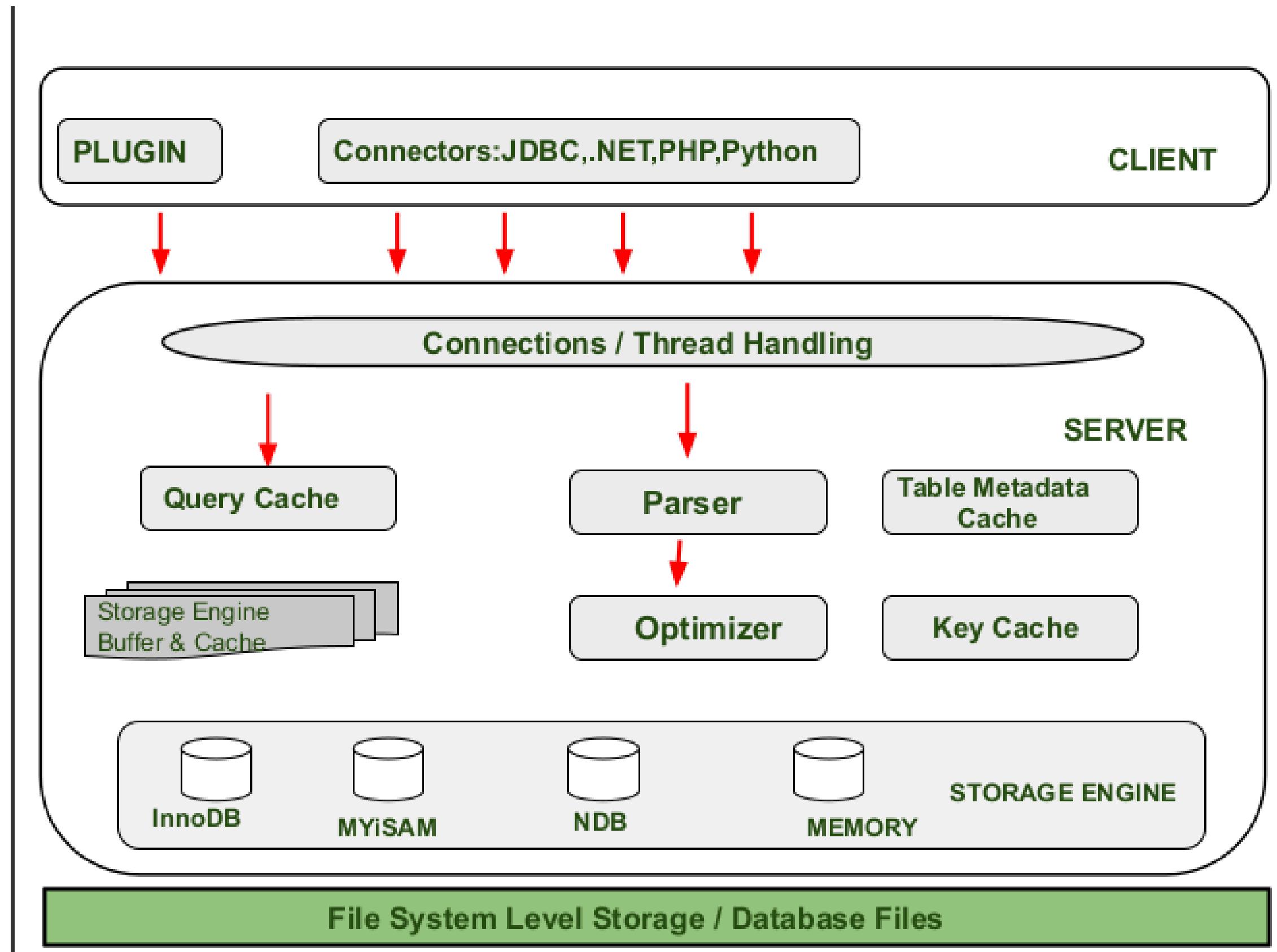
**MySQL o Oracle MySQL è un relational database management system (RDBMS) composto da un client a riga di comando e un server.**

**Ambo i costituenti sono multipiattaforma e sono disponibili ufficialmente su tutte le distribuzioni conosciute, quali Debian, Ubuntu e CentOS, sebbene lo abbiano sostanzialmente sostituito con MariaDB a partire dal 2012.**

**È software libero rilasciato a doppia licenza, compresa la GNU General Public License, sviluppato per essere il più possibile conforme agli standard ANSI SQL e ODBC SQL.**

**I sistemi e i linguaggi di programmazione che lo supportano sono molto numerosi, fra cui ODBC, Java, Mono, .NET, PHP, Python.**

# Architettura - MySQL



# SQL

**SQL è un linguaggio standard per l'archiviazione, la manipolazione e il recupero dei dati nei database.**

**Nonostante sia uno standard ANSI/ISO ci sono molte versioni SQL ed estensioni proprietarie dei vari software di database.**

**Un database si può riassumere in una serie di tabelle identificate con un nome e possono essere relazionate tra loro, queste tabelle sono divise in colonne per ogni attributo e in righe (record) per le informazioni che vogliamo salvare.**

	Code	Name	Continent	Region	SurfaceArea	IndepYear	Population	LifeExpectancy	GNP	GNPOld	LocalName
▶	ABW	Aruba	North America	Caribbean	193.00	NULL	103000	78.4	828.00	793.00	Aruba
	AFG	Afghanistan	Asia	Southern and Central Asia	652090.00	1919	22720000	45.9	5976.00	NULL	Afganistan/Afghanistan
	AGO	Angola	Africa	Central Africa	1246700.00	1975	12878000	38.3	6648.00	7984.00	Angola
	AIA	Anguilla	North America	Caribbean	96.00	NULL	8000	76.1	63.20	NULL	Anguilla
	ALB	Albania	Europe	Southern Europe	28748.00	1912	3401200	71.6	3205.00	2500.00	Shqipëria
	AND	Andorra	Europe	Southern Europe	468.00	1278	78000	83.5	1630.00	NULL	Andorra
	ANT	Netherlands Antilles	North America	Caribbean	800.00	NULL	217000	74.7	1941.00	NULL	Nederlandse Antillen

# LE QUERY

L'interrogazione di un database avviene mediante una **query**.

Ovvero una espressione o una serie di espressioni con delle richieste da fare al database per avere una risposta, sotto forma di tabella.

1 • | **SELECT \* FROM world.country;**

Questa **query** sta interrogando il database "world" andando a selezionare la tabella "country" dando quindi come risultato direttamente le righe della tabella country.

Code	Name	Continent	Region	SurfaceArea	IndepYear	Population	LifeExpectancy	GNP	GNPOld	LocalName
ABW	Aruba	North America	Caribbean	193.00	NULL	103000	78.4	828.00	793.00	Aruba
AFG	Afghanistan	Asia	Southern and Central Asia	652090.00	1919	22720000	45.9	5976.00	NULL	Afganistan/Afghanistan
AGO	Angola	Africa	Central Africa	1246700.00	1975	12878000	38.3	6648.00	7984.00	Angola
AIA	Anguilla	North America	Caribbean	96.00	NULL	8000	76.1	63.20	NULL	Anguilla
ALB	Albania	Europe	Southern Europe	28748.00	1912	3401200	71.6	3205.00	2500.00	Shqipëria
AND	Andorra	Europe	Southern Europe	468.00	1278	78000	83.5	1630.00	NULL	Andorra
ANT	Netherlands Antilles	North America	Caribbean	800.00	NULL	217000	74.7	1941.00	NULL	Nederlandse Antillen

# ESEMPIO DI QUERY

In questo esempio andiamo a recuperare solo la colonna "Continent" della tabella country dando come risultato una tabella con una singola colonna con solo i valori di Continent di tutte le righe di country.

	Continent
▶	North America
	Asia
	Africa
▶	North America
	Europe
	Europe

```
1 • SELECT Continent  
2   FROM world.country;
```

Code	Name	Continent	Region
ABW	Aruba	North America	Caribbean
AFG	Afghanistan	Asia	Southern and Central Asia
AGO	Angola	Africa	Central Africa
AIA	Anguilla	North America	Caribbean
ALB	Albania	Europe	Southern Europe
AND	Andorra	Europe	Southern Europe

# **SQL: SINTASSI**

**SQL è un linguaggio da una sintassi molto specifica che non è sensibile alle variazioni tra MAIUSCOLE e minuscole: "select" e "SELECT" sono uguali. Non è lo stesso per il nome delle tabelle e delle colonne.**

**Alcuni sistemi di database richiedono un punto e virgola alla fine di ogni istruzione SQL.**

**Il punto e virgola è il modo standard per separare ogni istruzione SQL nei database che consentono l'esecuzione di più istruzioni SQL nella stessa chiamata al server (come MySQL).**

# SQL: SINTASSI

## Alcuni dei comandi SQL più importanti:

- **SELECT** - estrae i dati da un database
- **UPDATE** - aggiorna i dati in un database
- **DELETE** - elimina i dati da un database
- **INSERT INTO** - inserisce nuovi dati in un database
- **CREATE DATABASE** - crea un nuovo database
- **ALTER DATABASE** - modifica un database
- **CREATE TABLE** - crea una nuova tabella
- **ALTER TABLE** - modifica una tabella
- **DROP TABLE** - elimina una tabella
- **CREATE INDEX** - crea un indice (chiave di ricerca)
- **DROP INDEX** - elimina un indice

# SQL: COMMENTI

**Come in Java è possibile inserire del testo che viene ignorato.**

**In SQL si può commentare una riga inserendo "--" prima del testo da commentare:**

- 1.-- selezioniamo tutte le colonne**
- 2.SELECT \* FROM table\_name -- dalla tabella**

**È possibile anche fare un commento multi riga inserendolo tra "/\*" e "\*/".**

- 1./\* selezioniamo tutte le colonne**
- 2.SELECT \* FROM table\_name \*/**
- 3. SELECT \* FROM world.country -- dalla tabella country del database world**

# SQL: SELECT

**L'istruzione SELECT serve per selezionare i dati da un database.**

**1. SELECT \* FROM table\_name;**

**I dati restituiti vengono archiviati in una tabella temporanea, denominata result-set.**

**1. SELECT column1, column2, ...**

**2. FROM table\_name;**

**Con SELECT andiamo a selezionare le colonne di una tabella andando a specificare il nome, invece il "\*" serve ad indicare tutte le colonne.**

**Con la "," possiamo selezionare più colonne contemporaneamente.**

# SQL: SELECT DISTINCT

L'istruzione **SELECT DISTINCT** funziona esattamente come **SELECT** solo che restituisce valori distinti tra loro, quindi senza ripetizioni.

- 1. `SELECT DISTINCT column1, column2, ...`**
- 2. `FROM table_name;`**

Con **SELECT DISTINCT** possiamo ad esempio sapere il numero di valori unici di una colonna usando il comando **COUNT** che conta i valori.

- 1. `SELECT COUNT(DISTINCT Name) FROM world.country;`**

	<code>COUNT(DISTINCT Name)</code>
▶	239

# SQL: WHERE Clause

L'istruzione WHERE è definita una "clausula", serve per mettere una condizione in modo da filtrare i record che andranno nel result-set.

- 1. SELECT column1, column2, ...**
- 2. FROM table\_name**
- 3. WHERE condition;**

Le condizioni possono essere di vario tipo e vanno scritte insieme all'operatore WHERE, ad esempio con valori stringa che vanno scritti tra singoli apici come ('testo').

- 1. SELECT \* FROM world.country**
- 2. WHERE Region ='Antarctica';**

# SQL: WHERE Clause

Un altro esempio può essere con un valore numerico:

```
1. SELECT * FROM world.country  
2. WHERE Population=0;
```

Le operazioni che si possono fare come condizione sono:

- = - uguale;
- > - maggiore di;
- < - minore di;
- >= - maggiore uguale di;
- <= - minore uguale di;
- != - non uguale(in alcune versione di SQL si scrive "!=", in altre ancora accetta entrambe le scritture come in MySQL);
- BETWEEN - tra un certo intervallo ("BETWEEN 0 and 10", valori compresi);
- LIKE - cerca un pattern, ovvero cerca una egualianza parziale o totale specificando il pattern fisso da trovare e le parti del dato variabili con un *metacarattere* come il "%" (esempio: "City LIKE 's%' " per trovare i valori che iniziano con "s" o "S");
- IN - per specificare più valori possibili (esempio "IN (0, 1000)");

# **SQL: WHERE (IL VALORE NULL)**

**Un campo con valore **NULL** è un campo senza valore che è diverso da un valore zero o da un campo che contiene spazi.**

**Un campo con un valore **NULL** è un campo che è stato lasciato vuoto durante la creazione di record.**

**Non è possibile verificare i valori **NULL** con operatori di confronto (come =, < o <>). Bisogna usare gli operatori **IS NULL** & **IS NOT NULL**.**

- 1. SELECT column\_names**
- 2. FROM table\_name**
- 3. WHERE column\_name IS NULL;**

# **SQL: AND - OR - NOT**

**Questi operatori permettono di combinare più condizioni, stanno ad indicare:**

**AND - visualizza il record se tutte le condizioni da sono soddisfatte;**

**OR - visualizza il record se almeno una condizione è soddisfatta;**

**NOT - visualizza il record se la condizione NON sono soddisfatte;**

**1. SELECT column1, column2, ...**

**2. FROM table\_name**

**3. WHERE condition1 AND condition2 AND condition3 ...;**

**Questi operatori possono essere affiancati da parentesi per indicare un gruppo di condizioni**

**1. SELECT \* FROM table\_name**

**2. WHERE condition1 AND (condition2 OR NOT condition3);**

# SQL: ORDER BY

L'istruzione ORDER BY serve ad ordinare il result-set su una o più colonne con due possibili criteri:

- **ASC - crescente**
- **DESC - decrescente**

Non è obbligatorio specificare il criterio di ordinamento, di default prende il criterio crescente (ASC).

1. **SELECT column1, column2, ...**
2. **FROM table\_name**
3. **ORDER BY column1, column2, ... ASC | DESC;**

# SQL: ORDER BY

In questo esempio ordiniamo in base alla colonna Region (string) e con ulteriore ordinamento decrescente in base alla colonna SurfaceArea (int).

Così facendo avremo un result-set ordinato in base alla regione, e le regioni uguali ordinate in base alla superficie più ampia.

```
1. SELECT * FROM world.country  
2. ORDER BY Region, SurfaceArea DESC;
```

Region	SurfaceArea
Antarctica	13120000.00
Antarctica	7780.00
Antarctica	3903.00
Antarctica	359.00
Antarctica	59.00
Australia and New Zealand	7741220.00
Australia and New Zealand	270534.00
Australia and New Zealand	135.00
Australia and New Zealand	36.00
Australia and New Zealand	14.00
Baltic Countries	65301.00
Baltic Countries	64589.00

# **SQL: GROUP BY**

**L'istruzione GROUP BY serve ad raggruppare i record con stessi valori del result-set su una o più colonne.**

**Viene spesso utilizzato con funzioni per raggruppare il result-set.**

- 1. SELECT column\_name(s)**
- 2. FROM table\_name**
- 3. WHERE condition**
- 4. GROUP BY column\_name(s)**
- 5. ORDER BY column\_name(s);**

# SQL: GROUP BY

Un esempio:

- 1. SELECT Country, COUNT(CustomerID)**
- 2. FROM Customers**
- 3. GROUP BY Country;**

Country	COUNT(CustomerID)
Argentina	3
Austria	2
Belgium	2
Brazil	9
Canada	3

# SQL: GROUP BY

Un esempio con l'ORDER BY:

```
1. SELECT Country, COUNT(CustomerID)  
2. FROM Customers  
3. GROUP BY Country  
4. ORDER BY COUNT(CustomerID) DESC;
```

Country	COUNT(CustomerID)
USA	13
Germany	11
France	11
Brazil	9
UK	7

# SQL: INSERT INTO

L'istruzione **INSERT INTO** serve per inserire nuovi record in una tabella.

- 1. `INSERT INTO table_name (column1, column2, column3, ...)`**
- 2. `VALUES (value1, value2, value3, ...);`**

È possibile non specificare le colonne se aggiungiamo valori a tutte le colonne ma bisogna assicurarsi che i valori abbiano l'ordine corretto delle colonne in cui devono essere inseriti.

- 1. `INSERT INTO table_name`**
- 2. `VALUES (value1, value2, value3, ...);`**

# SQL: UPDATE

L'istruzione **UPDATE** serve per modificare i record esistenti in una tabella.  
Viene tendenzialmente usato insieme a WHERE per filtrare i record da modificare altrimenti modificherebbe tutti i record della tabella.

- 1. UPDATE table\_name**
- 2. SET column1 = value1, column2 = value2, ...**
- 3. WHERE condition;**

**Con SET andiamo a specificare i valori da cambiare indicando la colonna con l'assegnazione del valore.**

- 1. UPDATE Customers**
- 2. SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'**
- 3. WHERE CustomerID = 1;**

# SQL: DELETE

L'istruzione **DELETE** serve per eliminare i record esistenti in una tabella.

Viene tendenzialmente usato insieme a WHERE per filtrare i record da eliminare altrimenti eliminerebbe tutti i record della tabella.

1. **DELETE FROM table\_name**
2. **WHERE condition;**

Questa istruzione va ad eliminare solo i record, la struttura della tabella rimane invariata pure se eliminano tutti i record senza specificare una condizione.

1. **DELETE FROM Customers**
2. **WHERE CustomerName='Mario Rossi';**

# Esercizi:

## 1) Utilizzo di DISTINCT e WHERE

Elencare, senza ripetizioni, tutte le regioni (Region) dei paesi che appartengono al continente (Continent) 'Europe'.

## 2) Combinazione di WHERE, ORDER BY

Elencare i nomi (Name) e la popolazione (Population) delle città (City) degli Stati Uniti (CountryCode = 'USA') che hanno una popolazione superiore a 1.000.000 abitanti, ordinando i risultati dalla città più popolosa alla meno popolosa.

## 3) GROUP BY con funzioni di aggregazione

Mostrare per ogni continente (Continent) presente nella tabella Country:

- Il numero totale di paesi appartenenti a ciascun continente.
- La popolazione totale del continente.

Ordinare il risultato per popolazione totale in ordine decrescente.

# Esercizi:

## Esercizio sui comandi SQL: GROUP BY, ORDER BY e INSERT INTO

**Si consideri un database che contiene informazioni su una libreria. Nel database è presente una tabella chiamata Libri con la seguente struttura:**

```
Libri (
    id INT PRIMARY KEY,
    titolo VARCHAR(100),
    autore VARCHAR(100),
    genere VARCHAR(50),
    prezzo DECIMAL(5,2),
    anno_pubblicazione INT
)
```

### 1) Inserimento dati (INSERT INTO)

Inserire almeno 6 nuovi libri nella tabella Libri usando il comando SQL INSERT INTO.

I libri devono appartenere a generi e autori diversi, ed essere pubblicati in anni differenti.

### 2) Aggregazione e raggruppamento (GROUP BY)

Scrivere una query che, usando il comando GROUP BY, mostri per ogni genere:

- il numero totale di libri presenti;
- il prezzo medio dei libri appartenenti a quel genere.

La query dovrà restituire il risultato ordinato alfabeticamente per genere.

### 3) Ordinamento risultati (ORDER BY)

Scrivere una query che elenchi tutti i libri pubblicati dopo l'anno 2010 ordinati in modo decrescente per anno di pubblicazione e, in caso di anno uguale, in ordine crescente per prezzo.

# SQL: SELECT TOP (**LIMIT**)

L'istruzione **SELECT TOP** serve per specificare il numero di record da restituire.

Non tutti i sistemi di database supportano questa istruzione. MySQL supporta l'operazione **LIMIT** per selezionare un numero limitato di record.

- 1. `SELECT column_name`**
- 2. `FROM table_name`**
- 3. `WHERE condition`**
- 4. `LIMIT number;`**

**LIMIT** quindi ci permette di indicare il numero massimo di record da recuperare da un database, questa operazione tendenzialmente si esegue per tavole con un grande numero di record ottimizzando le prestazioni.

- 1. `SELECT * FROM Customers`**
- 2. `LIMIT 50;`**

# SQL: ALIAS

Gli **ALIAS** SQL servono per assegnare un nome temporaneo ad un elemento come una tabella, una colonna, il risultato di una funzione ecc ecc...; In questo modo possiamo richiamare direttamente il nome temporaneo assegnato all'elemento.

Gli **ALIAS** possono essere utili quando:

- sono coinvolte più tabelle in una QUERY;
- vengono usate le funzioni in una QUERY;
- i nomi delle colonne sono grandi o poco leggibili;
- due o più colonne vengono combinate insieme;

Per assegnare un **ALIAS** viene usata la parola chiave **AS** quando andiamo a selezione o richiamare in **SELECT** o in **FROM** l'elemento a cui assegnare il nome:

1. **SELECT column\_name AS aliasColumnName**
2. **FROM table\_name AS aliasTableName;**

# SQL: MIN() - MAX()

**Sono delle funzioni che restituiscono rispettivamente il valore minimo e il valore massimo di una colonna, hanno lo stesso tipo di sintassi:**

- 1. SELECT MIN(column\_name)**
- 2. FROM table\_name**
- 3. WHERE condition;**

**Nel prossimo esempio prenderemo il prezzo più alto da una tabella di prodotti e associeremo il risultato ad un nome temporaneo tramite la parola chiave **AS** (ovvero un **ALIAS**):**

- 1. SELECT MAX(Price) **AS** LargestPrice**
- 2. FROM Products;**

# SQL: COUNT() - AVG() - SUM()

**Sono delle funzioni che eseguono un calcolo restituendo un valore specifico:**

- **COUNT()** - restituisce il numero di record selezionati;
- **AVG()** - restituisce il valore medio di una colonna con valori numerici;
- **SUM()** - restituisce la somma totale di una colonna con valori numerici;

**Tutte e tre le funzioni hanno lo stesso tipo di sintassi e possono essere associate a delle condizioni con WHERE:**

- 1. SELECT SUM(column\_name)**
- 2. FROM table\_name**
- 3. WHERE condition;**

# Esercizio

**Si consideri una tabella chiamata Vendite con la seguente struttura, almeno 20 elementi generati:**

- 1. Vendite (**
- 2. id INT,**
- 3. prodotto VARCHAR(100),**
- 4. categoria VARCHAR(50),**
- 5. quantita INT,**
- 6. prezzo\_unitario DECIMAL(6,2),**
- 7. data\_vendita DATE**
- 8.)**

**Scrivi le query SQL per rispondere alle seguenti richieste:**

- Totale vendite per categoria**

**Visualizza, per ogni categoria, il numero totale di vendite effettuate.**

- Prezzo medio per categoria**

**Mostra, per ogni categoria, il prezzo medio dei prodotti venduti.**

- Quantità totale venduta per ogni prodotto**

**Mostra il totale delle quantità vendute (SUM) per ciascun prodotto.**

- Prezzo massimo e minimo venduto nella tabella**

**Mostra il prezzo massimo e il prezzo minimo tra tutti i prodotti venduti.**

- Numero totale di righe nella tabella**

**Conta quante vendite sono state registrate nella tabella Vendite.**

- I 5 prodotti più costosi (in base al prezzo\_unitario)**

**Elenca i 5 prodotti più costosi ordinati in modo decrescente rispetto al prezzo.**

- I 3 prodotti meno venduti per quantità totale**

**Mostra i nomi dei 3 prodotti con la quantità totale più bassa venduta (usa SUM e LIMIT).**

# SQL: WILDCARD CHARACTERS

In italiano metacaratteri, sono dei caratteri che servono per sostituire uno o più caratteri in una stringa.

Vengono utilizzati nelle condizioni come WHERE e LIKE sono:

- "%" - zero o più caratteri ("bl%" = bl, black, blue e blob);
- "\_" - un singolo carattere ("h\_t" = hot, hat e hit);

Il resto dei metacaratteri hanno bisogno della funzione "REGEXP\_LIKE(colonna, pattern)"

- "[]" - ogni singolo carattere all'interno delle parentesi ("h[oa]t" = hot e hat);
- "^" - qualsiasi carattere non compreso tra parentesi ("h[^oa]t" = hit);
- "-" - qualsiasi carattere all'interno del range specificato ("c[a-b]t" = cat e cbt);

I metacaratteri possono essere usati in combinazione come ad esempio '\_r%' che rappresenta valori con almeno una r al secondo carattere.

# **SQL: LIKE**

**È un operatore che cerca un pattern preciso usando anche i metacaratteri:**

- 1. SELECT column1, column2, ...**
- 2. FROM table\_name**
- 3. WHERE columnN LIKE pattern;**

**In questo esempio seleziona tutti i clienti con il nome che inizia per "a":**

- 1. SELECT \* FROM Customers**
- 2. WHERE CustomerName LIKE 'a%';**

# SQL: IN

**È un operatore che consente di specificare più valori in una clausola, una scorciatoia per più condizioni OR.**

- 1. SELECT column\_name(s)**
- 2. FROM table\_name**
- 3. WHERE column\_name IN (value1, value2, ...);**

**È possibile inserire anche una table all'interno dei valori possibili:**

- 1. SELECT column\_name(s)**
- 2. FROM table\_name**
- 3. WHERE column\_name IN (SELECT STATEMENT);**

# **SQL: IN**

**Esempi di utilizzo:**

- 1. SELECT \* FROM Customers**
- 2. WHERE Country IN ('Germany', 'France', 'UK');**

**con NOT:**

- 1. SELECT \* FROM Customers**
- 2. WHERE Country NOT IN ('Germany', 'France', 'UK');**

**Confrontando valori di una table con un'altra table:**

- 1. SELECT \* FROM Customers**
- 2. WHERE Country IN (SELECT Country FROM Suppliers);**

# SQL: BETWEEN

**È un operatore che seleziona i valori (come: numeri, testo o date) all'interno di un determinato intervallo. L'operatore è inclusivo ovvero sono inclusi i valori di inizio e fine.**

- 1. SELECT column\_name(s)**
- 2. FROM table\_name**
- 3. WHERE column\_name BETWEEN value1 AND value2;**

**Un esempio di utilizzo con più condizioni:**

- 1. SELECT \* FROM Products**
- 2. WHERE Price BETWEEN 10 AND 20**
- 3. AND CategoryID NOT IN (1,2,3);**

# **SQL: BETWEEN**

**Un esempio utilizzando le date:**

- 1. SELECT \* FROM Orders**
- 2. WHERE OrderDate BETWEEN #07/01/1996# AND #07/31/1996#;**

**Un'altra sintassi per le date è:**

- 1. SELECT \* FROM Orders**
- 2. WHERE OrderDate BETWEEN '1996-07-01' AND '1996-07-31';**

**Un esempio utilizzando un ORDER BY:**

- 1. SELECT \* FROM Products**
- 2. WHERE ProductName NOT BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni'**
- 3. ORDER BY ProductName;**

# Esercizio

**Si consideri una tabella chiamata Clienti con la seguente struttura, almeno 20 dati inseriti:**

```
1. Clienti (  
2.   id INT,  
3.   nome VARCHAR(100),  
4.   cognome VARCHAR(100),  
5.   email VARCHAR(100),  
6.   eta INT,  
7.   citta VARCHAR(100)  
8.)
```

**Scrivere le query SQL per rispondere alle seguenti richieste:**

- **Clienti con email su dominio Gmail**
- **Seleziona tutti i clienti la cui email termina con @gmail.com.**
- **Clienti con nome che inizia con la lettera 'A'**
- **Mostra tutti i clienti il cui nome comincia con la lettera A.**
- **Clienti con cognome che contiene esattamente 5 lettere**
- **Mostra tutti i clienti il cui cognome è composto da esattamente 5 caratteri.**
- **Clienti con età compresa tra 30 e 40 anni (inclusi)**
- **Elenca i clienti che hanno un'età compresa tra 30 e 40 anni, inclusi gli estremi.**
- **Clienti che vivono in città il cui nome contiene 'roma' (maiuscole/minuscole ignorete)**
- **Mostra tutti i clienti che abitano in una città il cui nome contiene la stringa roma, indipendentemente da maiuscole o minuscole.**

# SQL: JOIN

**È una clausola che viene utilizzata per combinare righe da due o più tabelle, in base a una colonna correlata tra di loro.**

- 1. SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate**
- 2. FROM Orders**
- 3. INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;**

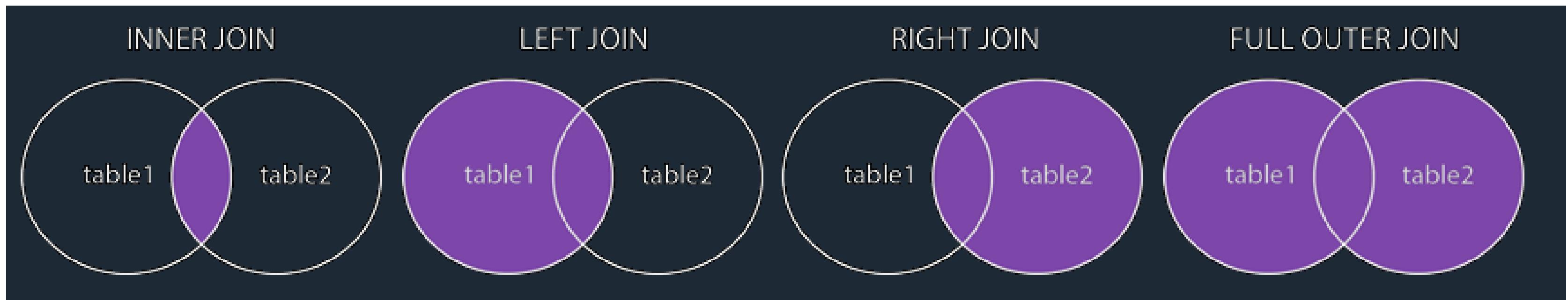
**In questo esempio stiamo andando a prendere i record che hanno lo stesso valore CustomerID unendo i valori selezionati generando un result-set con 3 colonne:**

- 1. OrderID della tabella orders**
- 2. CustomerName della tabella customers**
- 3. OrderDate della tabella orders**

# SQL: JOIN

Ci sono 4 tipi di join:

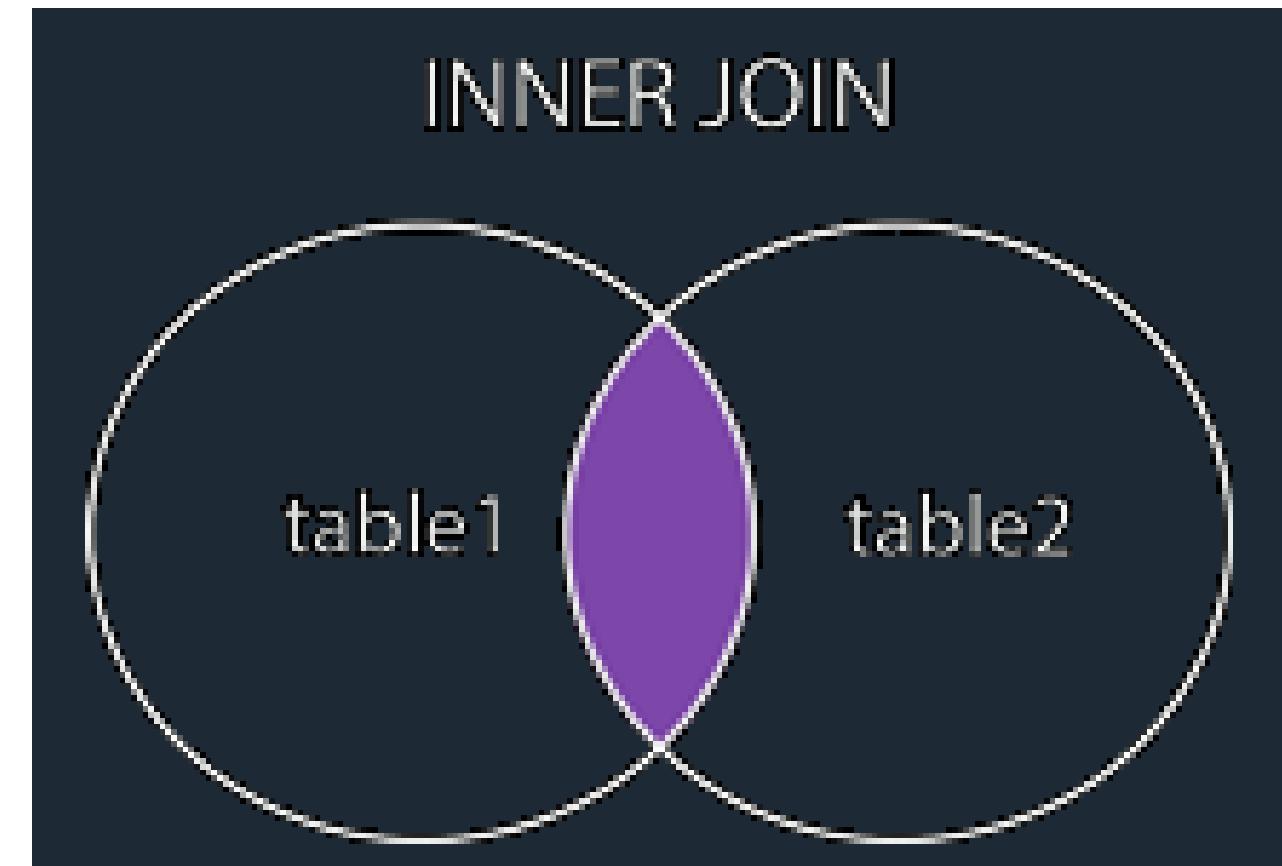
- **(INNER) JOIN:** restituisce record con valori in comune
- **LEFT (OUTER) JOIN:** restituisce tutti i record della tabella di sinistra e i record con valori in comune
- **RIGHT (OUTER) JOIN:** restituisce tutti i record della tabella di destra e i record con valori in comune
- **FULL (OUTER) JOIN:** restituisce tutti i record quando è presente una corrispondenza in uno dei due a sinistra o nella tabella a destra



# SQL: INNER JOIN

**INNER JOIN** seleziona i record con valori corrispondenti in entrambe le tabelle.

- 1. SELECT column\_name(s)**
- 2. FROM table1**
- 3. INNER JOIN table2**
- 4. ON table1.column\_name = table2.column\_name;**



# SQL: INNER JOIN

Esempio di utilizzo:

- 1. SELECT Orders.OrderID, Customers.CustomerName**
- 2. FROM Orders**
- 3. INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;**

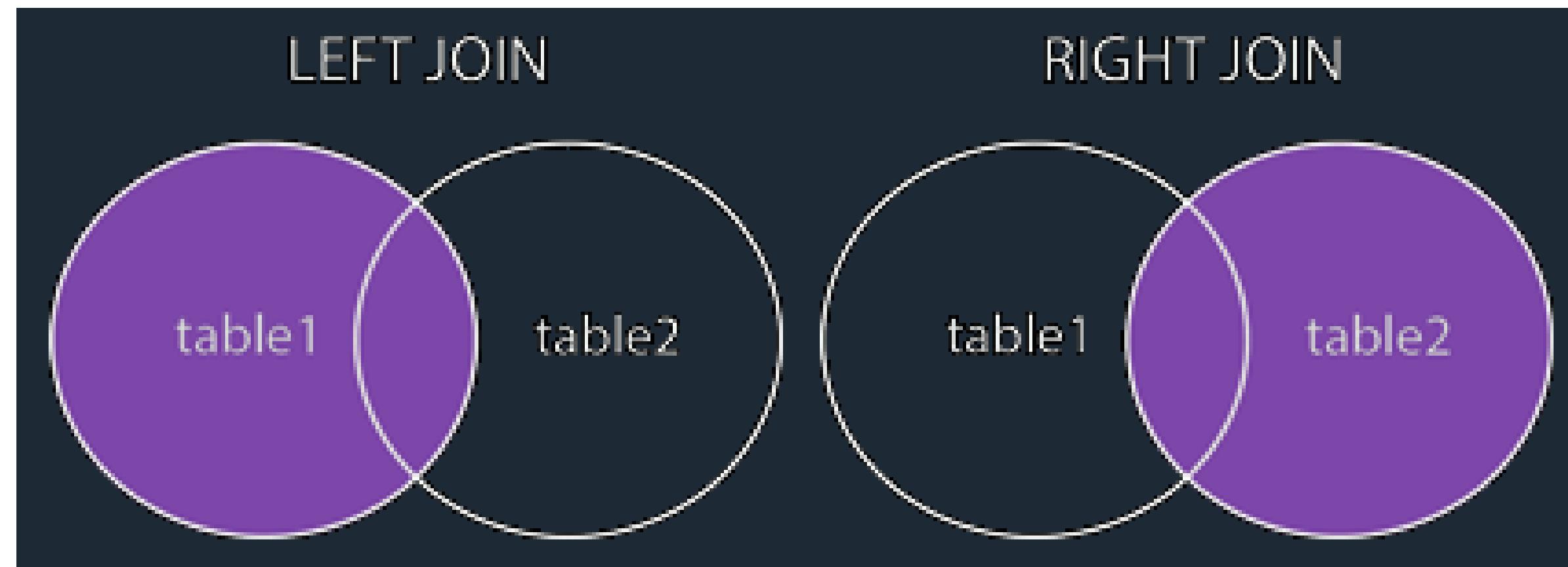
Esempio di utilizzo con più join (da notare le parentesi):

- 1. SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName**
- 2. FROM ((Orders**
- 3. INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID)**
- 4. INNER JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID);**

# SQL: LEFT JOIN - RIGHT JOIN

Questi join restituiscono i record della tabella di riferimento (Left o Right in base al tipo di join) insieme ai record che con i valori corrispondenti.

- 1. SELECT column\_name(s)**
- 2. FROM table1**
- 3. LEFT JOIN table2**
- 4. ON table1.column\_name = table2.column\_name;**



# SQL: LEFT JOIN - RIGHT JOIN

Un esempio dove recuperiamo e ordiniamo tutti i clienti in aggiunta recuperiamo anche l'eventuale id dell'ordine:

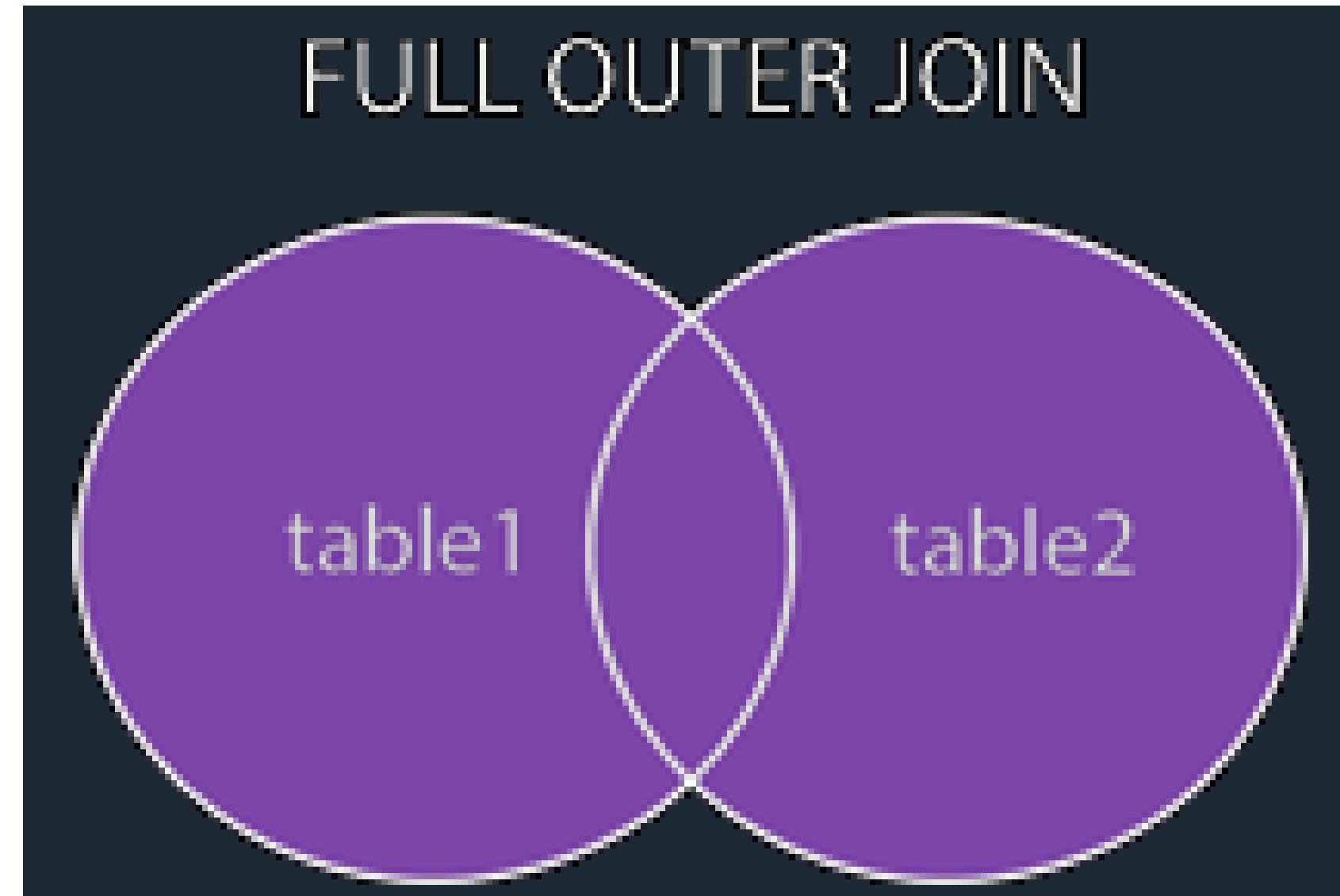
```
1. SELECT Orders.OrderID, Customers.CustomerName  
2. FROM Customers  
3. LEFT JOIN Orders  
4. ON Customers.CustomerID=Orders.CustomerID  
5. ORDER BY Customers.CustomerName;
```

OrderID	CustomerName
<i>null</i>	Alfreds Futterkiste
10308	Ana Trujillo Emparedados y helados
10365	Antonio Moreno Taquería

# SQL: CROSS JOIN

**CROSS JOIN** restituisce tutti i record da entrambe le tabelle.

- 1. SELECT column\_name(s)**
- 2. FROM table1**
- 3. CROSS JOIN table2;**



# Esercizio

**Si considerino le seguenti due tabelle con 20 dati l'una:**

**1.Clienti (**  
2. **id INT,**  
3. **nome VARCHAR(100),**  
4. **città VARCHAR(100)**  
**5.)**

**1.Ordini (**  
2. **id INT,**  
3. **id\_cliente INT,**  
4. **data\_ordine DATE,**  
5. **importo DECIMAL(7,2)**  
**6.)**

**Le due tabelle sono collegate dalla relazione tra Clienti.id e Ordini.id\_cliente.**

**Esercizio 1 – INNER JOIN Obiettivo:**

**Visualizza l'elenco dei clienti che hanno effettuato almeno un ordine.**

**Per ciascuno, mostra: nome del cliente, data dell'ordine e importo.**

**Esercizio 2 – LEFT JOIN Obiettivo:**

**Visualizza tutti i clienti, inclusi quelli che non hanno mai effettuato ordini.**

**Mostra per ciascuno: nome del cliente, data dell'ordine (se presente) e importo (se presente).**

**Esercizio 3 – RIGHT JOIN Obiettivo:**

**Visualizza tutti gli ordini, anche quelli che non hanno un cliente associato (caso anomalo).**

**Mostra per ciascuno: nome del cliente (se esiste), data dell'ordine e importo.**

# Esercizio

Devi realizzare un report completo per il reparto vendite, che soddisfi tutte le seguenti condizioni usando correttamente e separatamente i tre tipi di JOIN:

- Elenca i clienti attivi, cioè quelli che hanno effettuato almeno un ordine, mostrando per ciascuno:
  - Nome del cliente
  - Totale ordini effettuati
  - Somma totale degli importi spesi
- Elenca i clienti inattivi, cioè quelli che non hanno mai effettuato ordini, mostrando solo:
  - Nome del cliente
  - Città di residenza
- Individua gli ordini orfani, cioè ordini presenti in tabella ma senza un cliente valido associato (es. cliente cancellato), e mostra:
  - ID dell'ordine
  - Data dell'ordine
  - Importo
  - (Cliente = NULL)

**Requisiti tecnici:**

- Per il punto 1: usa INNER JOIN.
- Per il punto 2: usa LEFT JOIN con condizione su IS NULL.
- Per il punto 3: usa RIGHT JOIN con condizione su IS NULL.

# SQL: UNION

**L'operatore UNION viene utilizzato per combinare il result-set di risultati distinti di due o più istruzioni. Vanno però rispettate queste condizioni:**

- L'istruzione SELECT all'interno deve avere lo stesso numero di colonne;
- Le colonne devono inoltre avere tipi di dati simili;
- Le colonne devono inoltre essere nello stesso ordine.

Sintassi:

1. **SELECT column\_name(s) FROM table1**
2. **UNION**
3. **SELECT column\_name(s) FROM table2;**

# SQL: UNION ALL

È identico all'operatore UNION solo che include tutti i valori anche i duplicati.

- 1. SELECT column\_name(s) FROM table1**
- 2. UNION ALL**
- 3. SELECT column\_name(s) FROM table2;**

Esempio:

- **SELECT City FROM Customers**
- **UNION ALL**
- **SELECT City FROM Suppliers**
- **ORDER BY City;**

NOTA: potete utilizzare USE e omettere il nome della tabella per le colonne quando possibile

**Esercizi:**

- 1) Si vogliono recuperare dal database "world" la lingua e la nazione di ogni città.**
- 2) Si vuole recuperare il numero di città per nazione dal database "world" mostrando anche il nome della nazione e ordinarli in base al numero di città.**
- 3) Si vuole conoscere la lista di repubbliche con aspettativa di vita maggiore dei 70 anni, inoltre si vuole visualizzare anche la lingua parlata.**

# SQL: HAVING

**HAVING è una clausola che opera sui raggruppamenti invece che sui campi della tabella.**

- 1. SELECT column\_name(s)**
- 2. FROM table\_name**
- 3. WHERE condition**
- 4. GROUP BY column\_name(s)**
- 5. HAVING condition**
- 6. ORDER BY column\_name(s);**

**NOTA: deve essere usato prima dell'ORDER By.**

# SQL: EXISTS

L'operatore **EXISTS** viene utilizzato per verificare l'esistenza di qualsiasi record in una sottoquery, restituisce **TRUE** se la sottoquery restituisce uno o più record.

- 1. SELECT column\_name(s)**
- 2. FROM table\_name**
- 3. WHERE EXISTS**
- 4. (SELECT column\_name FROM table\_name WHERE condition);**

L'esempio seguente restituisce **TRUE** ed elenca i fornitori con un prezzo del prodotto inferiore a 20:

- 1. SELECT SupplierName**
- 2. FROM Suppliers**
- 3. WHERE EXISTS**
- 4. (SELECT ProductName FROM Products WHERE Products.SupplierID =**
- 5. Suppliers.supplierID AND Price < 20);**

# SQL: ANY

**L'operatore ANY restituisce TRUE se UNO qualsiasi dei valori della sottoquery soddisfa la condizione dell'operatore.**

- 1. SELECT column\_name(s)**
- 2. FROM table\_name**
- 3. WHERE column\_name operator ANY**
- 4. (SELECT column\_name FROM table\_name WHERE condition);**

**L'esempio seguente restituisce TRUE se qualsiasi prodotto ha ordini superiori a 90:**

- 1. SELECT ProductName**
- 2. FROM Products**
- 3. WHERE ProductID = ANY**
- 4. (SELECT ProductID FROM OrderDetails WHERE Quantity > 90);**

# SQL: ALL

L'operatore **ALL** restituisce TRUE se TUTTI i valori della sottoquery soddisfa la condizione dell'operatore, si può utilizzare in: **SELECT, WHERE, HAVING**.

- 1. SELECT ALL column\_name(s)**
- 2. FROM table\_name**
- 3. WHERE column\_name operator ALL**
- 4. (SELECT column\_name FROM table\_name WHERE condition);**

L'esempio seguente elenca ProductName se TUTTI i record nella tabella OrderDetails ha Quantity uguale a 10:

- 1. SELECT ProductName**
- 2. FROM Products**
- 3. WHERE ProductID = ALL (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);**

**ESERCIZIO 1:**

**Si vuole recuperare dal database WORLD le lingue parlate per nazione con la rispettiva percentuale di utilizzo;**

**ESERCIZIO 2:**

**Si vuole recuperare dal database WORLD le nazioni e la percentuale della lingua più parlata;**

**ESERCIZIO 3:**

**Unire gli esercizi 1e2 facendole diventare subQuery per mostrare la lingua più parlata di una nazione con la percentuale;**

# SQL: SCHEMA RELAZIONALE

È uno schema che ci riassume le varie tabelle con le PK (chiave primaria) e le FK\* (chiave esterna):

- ATTORI (CodAttore, Nome, AnnoNascita, Nazionalità);
- RECITA (CodAttore\*, CodFilm\*)
- FILM (CodFilm, Titolo, AnnoProduzione, Nazionalità, Regista, Genere)
- PROIEZIONI (CodProiezione, CodFilm\*, CodSala\*, Incasso, DataProiezione)
- SALE (CodSala, Posti, Nome, Città)

# **SQL: INSERT INTO SELECT**

**INSERT INTO con SELECT copia i record da una tabella e li aggiunge ad un'altra tabella.**

**L'operazione richiede che i tipi di dati delle colonne nelle tabelle di origine e di destinazione corrispondono.**

- 1. INSERT INTO table2 (column1, column2, column3, ...)**
- 2. SELECT column1, column2, column3, ...**
- 3. FROM table1**
- 4. WHERE condition;**

# SQL: CASE

**CASE passa attraverso dei WHEN restituisce con THEN un valore quando trova la prima condizione vera, quindi, una volta che una condizione è vera, si fermerà la lettura del CASE e restituirà un valore.**

**Se nessuna condizione è vera, restituisce il valore indicato nella clausola ELSE.**

- 1. CASE**
- 2. WHEN condition1 THEN result1**
- 3. WHEN condition2 THEN result2**
- 4. WHEN conditionN THEN resultN**
- 5. ELSE result**
- 6. END;**

**Possiamo quindi considerarlo un if-elseif-else oppure uno switch di java**

# SQL: CASE

**Un esempio che passa un valore in base alla condizione vera trovata:**

- 1. SELECT OrderID, Quantity,**
- 2. CASE**
- 3. WHEN Quantity > 30 THEN 'The quantity is greater than 30'**
- 4. WHEN Quantity = 30 THEN 'The quantity is 30'**
- 5. ELSE 'The quantity is under 30'**
- 6. END AS QuantityText**
- 7. FROM OrderDetails;**

# SQL: CASE

**Un esempio che sceglie in base alla condizione per quale colonna ordinare:**

```
1. SELECT CustomerName, City, Country  
2. FROM Customers  
3. ORDER BY  
4. (CASE  
5.   WHEN City IS NULL THEN Country  
6.   ELSE City  
7. END);
```

# **SQL: IFNULL() - COALESCE()**

**IFNULL() è una funzione che restituisce un valore alternativo se l'espressione è NULL:**

**1. IFNULL(expression, alt\_value)**

**COALESCE() è una funzione che restituisce il primo valore non NULL:**

**1. COALESCE(val1, val2, ...., val\_n)**

**Esercizio1:**  
**scrivere e mostrare con JDBC una query che vada a prendere le nazioni mostrando come risposta se hanno o meno una superficie di 100.000 e se hanno registrato un IndepYear, se è presente l'anno va mostrato altrimenti si indica che non è presente;**

**Esercizio2:**  
**recuperare e mostrare le città con il codice della nazione che indica l'utente. Inoltre l'utente può decidere:**

- **l'ordine con cui ordinare le città in maniera decrescente o meno;**
- **se filtrare le città per un minimo di popolazione;**
- **se mostrare il nome della nazione a cui fa riferimento il code;**

# **SQL: CREATE/DROP DATABASE**

**Sono delle operazioni che richiedono i privilegi di un amministratore.**

**CREATE DATABASE ci permette di creare un nuovo database:**

**1. CREATE DATABASE databasename;**

**DROP DATABASE ci permette di eliminare un database eliminando tutto il suo contenuto:**

**1. DROP DATABASE databasename;**

# SQL: CREATE TABLE

**CREATE TABLE** ci permette di creare una nuova tabella nel database database specificando il nome delle colonne con le loro caratteristiche:

```
1. CREATE TABLE table_name (  
2.   column1 datatype,  
3.   column2 datatype,  
4.   column3 datatype,  
5.   ....  
6. );
```

Ci sono numerosi datatype che possiamo dare alle colonne, i più usati sono il VARCHAR(nMax di char) e l'INT.

# SQL: CREATE TABLE

Esempio:

```
1. CREATE TABLE Persons (
2.   PersonID int,
3.   LastName varchar(255),
4.   FirstName varchar(255),
5.   Address varchar(255),
6.   City varchar(255)
7.);
```

PersonID	LastName	FirstName	Address	City

# SQL: CREATE TABLE

**CREATE TABLE** ci permette di creare una nuova tabella anche da un'altra tabella, copiando le caratteristiche delle tabelle selezionate e i suoi record:

1. **CREATE TABLE new\_table\_name**
2. **SELECT column1, column2,...**
3. **FROM existing\_table\_name**
4. **WHERE ....;**

Un esempio di sintassi:

1. **CREATE TABLE TestTable**
2. **SELECT customername, contactname**
3. **FROM customers;**

# SQL: DATATYPE

In MySQL ci sono tre tipi di dati principali: stringa, numerico, data e ora.

Qui elencheremo i principali.

I principali tipi sono:

- **CHAR(size): una stringa di grandezza fissa(max 255);**
- **VARCHAR(max\_size): una stringa a grandezza variabile fino al massimo indicato(max 65535);**
- **MEDIUMTEXT: una stringa lunga fino a 16,777,215 caratteri;**
- **LONGTEXT: una stringa lunga fino a 4,294,967,295 caratteri;**
- **ENUM(val1, val2, val3, ...): una lista di valori possibili;**
- **BOOL: un valore booleano, 0 per false e 1 per true;**
- **INT: un valore numerico intero;**
- **FLOAT: un valore numero con virgola mobile;**
- **DATE: data in formato YYYY-MM-DD;**

# **SQL: DROP/TRUNCATE TABLE**

**DROP TABLE** ci permette di eliminare una tabella con tutto il suo contenuto:

**1. `DROP TABLE table_name`**

**TRUNCATE TABLE** ci permette di eliminare tutti i record di una tabella senza eliminare la tabella:

**1. `TRUNCATE TABLE table_name`**

# SQL: ALTER TABLE

**ALTER TABLE** ci permette di aggiungere, eliminare o modificare colonne in una tabella esistente con:

- **ADD:** per aggiungere una colonna;
- **DROP COLUMN:** per eliminare una colonna
- **MODIFY COLUMN:** per modificare una colonna;

**1. ALTER TABLE table\_name  
2. ADD column\_name datatype;**

**NOTA:** è possibile modificare anche i vincoli di una tabella.

# SQL: CONSTRAINT

**Per ogni colonna si possono specificare dei vincoli sul tipo di dato.**

```
1. CREATE TABLE table_name (  
2.   column1 datatype constraint,  
3.   .... );
```

**I vincoli più utilizzati sono:**

- **NOT NULL** - non ci può essere un valore **NULL**;
- **UNIQUE** - tutti i valori in una colonna sono **differenti**;
- **PRIMARY KEY** - identifica in modo univoco ogni riga di una tabella;
- **FOREIGN KEY** - identifica una chiave esterna;
- **CHECK** - i valori in una colonna devono soddisfare una condizione;
- **DEFAULT** - imposta un valore di **default**;
- **CREATE INDEX** - crea un index per recuperare dati più velocemente.

# **SQL: CONSTRAINT**

**NOT NULL esempio:**

```
1. CREATE TABLE Persons (
2.   ID int NOT NULL,
3.   LastName varchar(255) NOT NULL,
4.   FirstName varchar(255) NOT NULL,
5.   Age int
6. );
```

**Per poi modificare in successiva Age:**

```
1. ALTER TABLE Persons
2. MODIFY Age int NOT NULL;
```

# SQL: CONSTRAINT

**UNIQUE esempio:**

- 1. CREATE TABLE Persons (**
- 2. ID int NOT NULL,**
- 3. LastName varchar(255) NOT NULL,**
- 4. FirstName varchar(255),**
- 5. Age int,**
- 6. UNIQUE (ID) );**

**È possibile indicare più colonne UNIQUE modificando la sintassi indicando il nome del vincolo:**

- 1. CONSTRAINT ctnt\_name UNIQUE (ID,LastName)**

**Per poi eliminare un vincolo bisogna fare:**

- 1. ALTER TABLE table\_name**
- 2. DROP INDEX ctnt\_name;**

# SQL: CONSTRAINT

**PRIMARY KEY esempio:**

```
1. CREATE TABLE Persons (
2.   ID int NOT NULL,
3.   LastName varchar(255) NOT NULL,
4.   FirstName varchar(255),
5.   Age int,
6.   PRIMARY KEY (ID));
```

**È possibile indicare una PRIMARY KEY su più colonne con un CONSTRAINT:**

```
1. CONSTRAINT PK_name PRIMARY KEY (column1, column2)
```

**Per poi eliminare una PRIMARY KEY bisogna fare:**

```
1. ALTER TABLE table_name
2. DROP PRIMARY KEY;
```

# SQL: CONSTRAINT

**FOREIGN KEY esempio:**

```
1. CREATE TABLE Orders (
2.   OrderID int NOT NULL,
3.   OrderNumber int NOT NULL,
4.   PersonID int,
5.   PRIMARY KEY (OrderID),
6.   FOREIGN KEY (PersonID) REFERENCES Persons(PersonID) );
```

È possibile indicare una **FOREIGN KEY** con un **CONSTRAINT**:

```
1. CONSTRAINT FK_name FOREIGN KEY (column) REFERENCES ref_table(column)
```

Per poi eliminare una **FOREIGN KEY** bisogna fare:

```
1. ALTER TABLE table_name
2. DROP FOREIGN KEY FK_name;
```

# SQL: CONSTRAINT

**CHECK esempio:**

```
1. CREATE TABLE Persons (
2.   ID int NOT NULL,
3.   LastName varchar(255) NOT NULL,
4.   FirstName varchar(255),
5.   Age int,
6.   CHECK (Age>=18);
```

È possibile indicare un **CHECK** con un **CONSTRAINT**:

```
1. CONSTRAINT CHK_name CHECK (con1 AND con2)
```

Per poi eliminare un **CHECK** bisogna fare:

```
1. ALTER TABLE table_name
2. DROP CHECK CHK_name;
```

# SQL: CONSTRAINT

**DEFAULT esempio:**

```
1. CREATE TABLE Persons (
2.   ID int NOT NULL,
3.   LastName varchar(255) NOT NULL,
4.   FirstName varchar(255),
5.   Age int,
6.   City varchar(255) DEFAULT 'Sandnes' );
```

**È possibile indicare un valore con una function:**

```
1. OrderDate date DEFAULT CURRENT_DATE()
```

**Per poi eliminare o modificare un DEFAULT bisogna fare:**

```
1. ALTER TABLE table_name
2. ALTER column DROP DEFAULT; / ALTER column SET DEFAULT value;
```

# SQL: CONSTRAINT

**AUTO\_INCREMENT esempio:**

```
1. CREATE TABLE Persons (
2.   Personid int NOT NULL AUTO_INCREMENT,
3.   LastName varchar(255) NOT NULL,
4.   FirstName varchar(255),
5.   Age int,
6.   PRIMARY KEY (Personid) );
```

È possibile indicare un valore con cui far iniziare la sequenza:

```
1. ALTER TABLE Persons AUTO_INCREMENT=100;
```

**NOTE:** la colonna con il valore **AUTO\_INCREMENT** non permette di inserirgli valori.

# **SQL: DATE**

**MySQL fornisce diversi tipi di DATE per la memorizzazione di una data o di un valore di data / ora nel database:**

- **DATE - formato AAAA-MM-DD ('2038-01-19')**
- **DATETIME - formato: AAAA-MM-GG HH:MI:SS ('2038-01-19 03:14:07.999999')**
- **TIMESTAMP - formato: AAAA-MM-GG HH:MI:SS ('2038-01-19 03:14:07.999999')**
- **YEAR - formato AAAA o YY ('2038', '00' - '69' per il 2000, '70' - '99' per il 1900)**

**NOTE: è possibile confrontare con operatori di uguaglianza solo DATE senza valori temporali (data, ora e minuti).**

# SQL: VIEW

**Una VIEW è un tabella virtuale basata sul result-set di una QUERY, contiene righe e colonne come una tabella. I campi di una VIEW sono campi di una o più tabelle presenti nel database.**

- 1. CREATE VIEW view\_name AS**
- 2. SELECT column1, column2, ...**
- 3. FROM table\_name**
- 4. WHERE condition;**

**Nota: una VIEW mostra sempre dati aggiornati, il di database ricrea la VIEW ad ogni QUERY.**

- 1. CREATE VIEW [Brazil Customers] AS**
- 2. SELECT CustomerName, ContactName**
- 3. FROM Customers**
- 4. WHERE Country = 'Brazil';**

# **SQL: VIEW**

**Una VIEW si può aggiornare modificandola con il comando CREATE OR REPLACE VIEW:**

- 1. CREATE OR REPLACE VIEW view\_name AS**
- 2. SELECT column1, column2, ...**
- 3. FROM table\_name**
- 4. WHERE condition;**

**Per eliminare una VIEW invece bisogna usare DROP VIEW:**

- 1. DROP VIEW view\_name;**

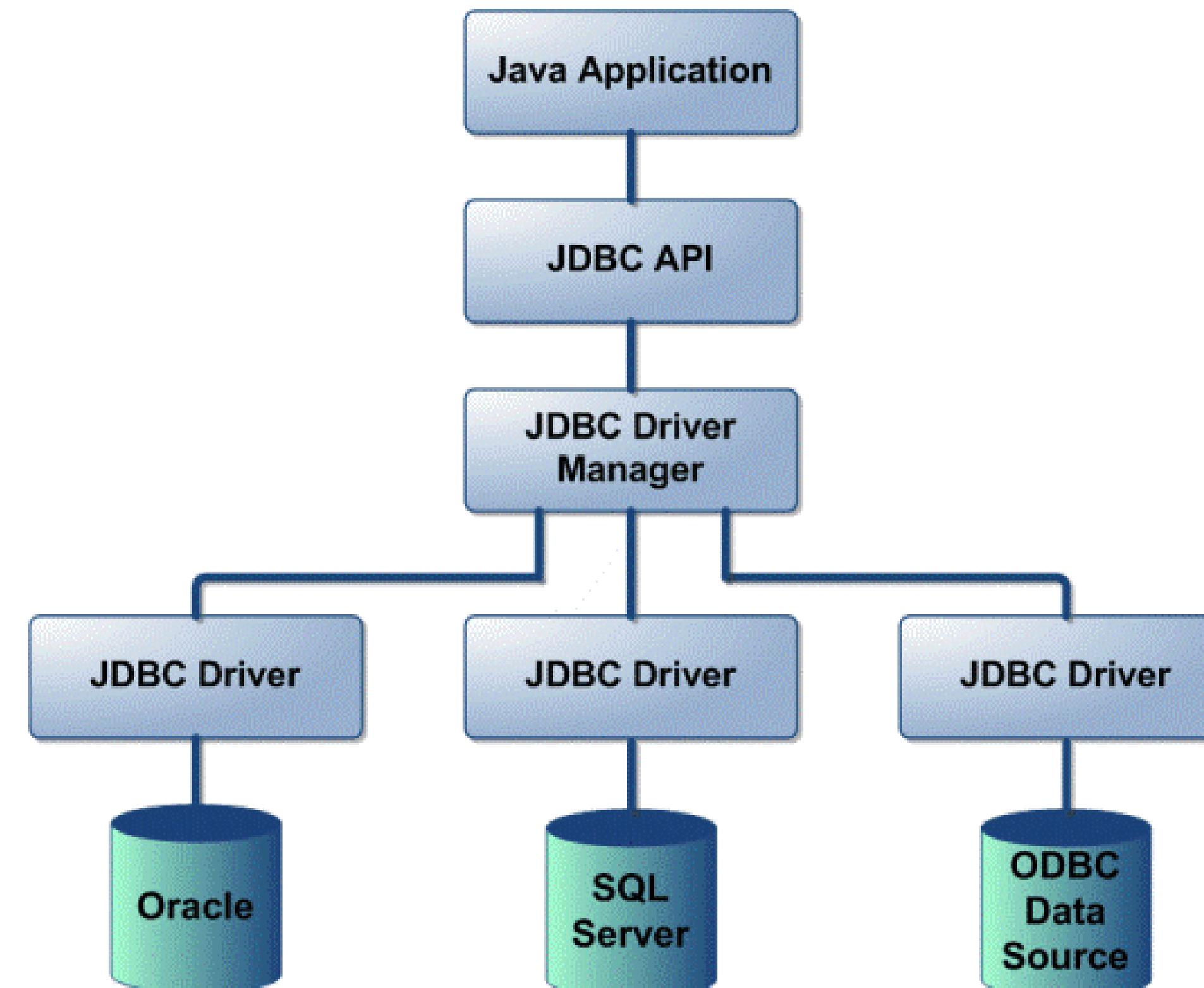
## **Esercizio:**

**Creare una view di city del database world su una query che mostra le città italiane.**

**Su questa VIEW eseguire una query che mostra solo le città con una popolazione inferiore ai 100k.**

# JDBC

**NOTA: ODBC è un driver attraverso un'API standard per la connessione dal client al DBMS**



# JDBC:

**In informatica JDBC (Java DataBase Connectivity), è un connettore e un driver per database che consente l'accesso e la gestione della persistenza dei dati sulle basi di dati da qualsiasi programma scritto con il linguaggio di programmazione Java, indipendentemente dal tipo di DBMS utilizzato.**

**È costituito da un'API object oriented orientata ai database relazionali, raggruppata nel package `java.sql`, che serve ai client per connettersi a un database fornendo i metodi per interrogare e modificare i dati.**

**L'architettura di JDBC, prevede l'utilizzo di un "driver manager", che espone alle applicazioni un insieme di interfacce standard e si occupa di caricare a "run-time" i driver opportuni per "pilotare" gli specifici DBMS.**

**Le applicazioni Java utilizzano le "JDBC API" per parlare con il JDBC driver manager, mentre il driver manager usa le JDBC driver API per parlare con i singoli driver che pilotano i DBMS specifici.**

- JDBC:** L'API JDBC è costituita dai seguenti componenti principali:
- **JDBC Drivers:** una raccolta di classi e interfacce per comunicare con il database;
  - **Connections:** stabilisce una connessione con il database;
  - **Statements:** dichiarazioni che servono per avviare le QUERY;
  - **ResultSets:** gli elementi restituiti da una QUERY, La classe **ResultSet** fornisce un cursore che punta alla riga corrente nel result-set fornito dalla query;

**I passaggi per connettere il database utilizzando JDBC sono:**

- 1. Caricare il driver JDBC;**
- 2. Connessione;**
- 3. Statements;**
- 4. Esecuzione statements;**
- 5. Chiudere la connessione al database;**

# JDBC: DRIVERS & CONNECTION

I driver da caricare cambiano in base al DBMS che utilizziamo.

Vengono caricati a runTime grazie alla classe **DriverManager** con l'url di connessione al DB formattato secondo lo standard **jdbc** e varia in base al DBMS.

In MySQL l'url è: **jdbc:mysql://localhost:3306/nomeSchema**

Possiamo scomporlo in 3 punti:

1. **jdbc:mysql://** >> prefisso standard per le connessioni MySQL;
2. **localhost:3306** >> nome del server e porta che ospita il DB;
3. **nomeSchema** >> nome dello schema a cui fare riferimento;

Una volta recuperato l'url di connessione è possibile effettuare la connessione al DB con le credenziali di accesso al DB.

Questo ci permetterà di ricavare l'oggetto **Connection** grazie al **DriverManager**:

**1. Connection conn = DriverManager.getConnection(url,username,password);**

# JDBC: DRIVERS & CONNECTION

**Per far funzionare la connessione abbiamo bisogno di importare il Driver "mysql-connector-j" per MySql nel progetto, questo permetterà al DriverManager di recuperare il driver necessario per la comunicazione con il DB.**

**Potrebbe essere necessario chiamare manualmente la registrazione del driver:**

- 1. String DB\_DRIVER = "com.mysql.jdbc.Driver";**
- 2. Class.forName(DB\_DRIVER);**

**Per utilizzare invece il DriverManager e Connection abbiamo bisogno di importare due librerie:**

- 1. import java.sql.Connection;**
- 2. import java.sql.DriverManager;**

# JDBC: DRIVERS & CONNECTION

Un esempio di sintassi con anche chiamata manuale del Driver:

```
1. String DB_DRIVER = "com.mysql.jdbc.Driver";
2. String DB_URL = "jdbc:mysql://localhost:3306/world";
3. String DB_USERNAME = "root";
4. String DB_PASSWORD = "root";
5. Connection conn = null;
6. try {
7.     // Register the JDBC driver
8.     Class.forName(DB_DRIVER);
9.     // Open the connection
10.    conn = DriverManager.getConnection(DB_URL, DB_USERNAME, DB_PASSWORD);
11.    if (conn != null)
12.        System.out.println("Successfully connected.");
13.    else
14.        System.out.println("Failed to connect.");
15. } catch (Exception e) {
16.     e.printStackTrace(); }
```

# JDBC: STATEMENT & RESULT-SET

**Le interfacce Statement definiscono i metodi e le proprietà che consentono di inviare comandi SQL e ricevere dati dal database. Ci sono tre tipi di statement:**

- **Statement** - non è in grado di accettare parametri;
- **PreparedStatement** - accetta parametri di input;
- **CallableStatement** - accetta parametri di input a runtime;

**La differenza principale tra Prepared e Callable è che PreparedStatement viene gestito in Java mentre CallableStatement viene recuperato dal database.**

**Quindi Statement viene usato per query fisse, statiche. Invece PreparedStatement ci permette di settare dei valori prima di eseguire la query.**

- 1. Statement stmt = conn.createStatement();**
- 2. ResultSet rs = stmt.executeQuery(QUERY);**

# JDBC: STATEMENT & RESULT-SET

**Una volta creato lo statement bisogna eseguirlo per recuperare un oggetto di ResultSet:**

- 1. PreparedStatement stmt = conn.prepareStatement(QUERY);**
- 2. ResultSet rs = stmt.executeQuery();**

**Una volta recuperato il result-set è possibile accedervi come se fosse una matrice sfruttando un ciclo while per far avanzare il cursore del result-set su tutte le righe:**

- 1. while(rs.next())**
- 2. {**
- 3.     System.out.println(rs.getString(1)); //First Column**
- 4.     System.out.println(rs.getString(2)); //Second Column**
- 5.     System.out.println(rs.getString(3)); //Third Column**
- 6.     System.out.println(rs.getString(4)); //Fourth Column**
- 7. }**

# JDBC: RESULT-SET

**È possibile anche indicare il nome della colonna nei getter dei valori per le colonne:**

```
1. System.out.println(rs.getString("column_name"));
```

**È possibile recuperare anche i METADATI dal result-set, ovvero tutte le informazioni riguardanti la struttura della tabella come ad esempio il tipo di una colonna o il numero di colonne o il nome della tabella ecc ecc...**

```
1. ResultSetMetaData metaData = rs.getMetaData();
2. Integer columnCount = metaData.getColumnCount();
```

**Nell'esempio andiamo a recuperare il numero di colonne del result-set.**

**NOTA: è necessaria la libreria java.sql.ResultSetMetaData.**

# JDBC: RESULT-SET

**Le principali funzioni della classe ResultSetMetaData sono:**

- **getColumnName()** - restituisce il numero di colonne;
- **getColumnName(int columnNumber)** - restituisce il nome della colonna;
- **getColumnLabel(int columnNumber)** - restituisce l'alias assegnato nella QUERY alla colonna;
- **getTableName(int columnNumber)** - restituisce il nome della tabella;
- **getColumnType(int columnNumber)** - restituisce il tipo di dato della colonna;
- **isAutoIncrement(int columnNumber)** - indica se la colonna è auto-incrementata;
- **isCaseSensitive(int columnNumber)** - indica se la colonna è case sensitive;
- **isSearchable(int columnNumber)** - indica se possiamo usare la colonna nel WHERE;
- **isNullable(int columnNumber):**
  - **restituisce:**
    - 0 se può avere NULL;
    - 1 se non può avere NULL;
    - 2 se è sconosciuto;

# JDBC: RESULT-SET

**Il result-set ottenuto è navigabile grazie ad un cursore, che di default avanza alle righe successive.**

**La navigazione del cursore è modificabile tramite dei valori da passare al momento della creazione dello statement (se è supportato dal DBMS).**

**Possiamo usare diverse funzioni per far scorrere il cursore:**

- **next() - passa alla riga successiva;**
- **previous() - passa alla riga precedente;**
- **first() - passa alla prima riga del ResultSet;**
- **last() - passa all'ultima riga;**
- **beforeFirst() - si sposta all'inizio, prima della prima riga;**
- **afterLast() - si sposta alla fine, dopo l'ultima riga;**
- **relative(int numRows) - si sposta del numero di righe indicate (anche in negativo);**
- **absolute(int rowNumber) - passa alla riga specificata;**
- **getRow() - recupera il numero della riga;**

# JDBC: RESULT-SET

**Di default il result-set è di sola lettura, per renderlo modificabile è necessario indicare alla creazione dello statement che può essere modificato:**

- 1. PreparedStatement pstmt = dbConnection.prepareStatement(**
- 2. QUERY, // la query da eseguire**
- 3. ResultSet.TYPE\_SCROLL\_SENSITIVE, // indichiamo il tipo di scorrimento del cursore**
- 4. ResultSet.CONCUR\_UPDATABLE); // indichiamo che il result-set è aggiornabile**
- 5. ResultSet rs = pstmt.executeQuery();**

**Si possono inserire e modificare i record del result-set ottenuto:**

- 1. rs.updateDouble("salary", 1100.0);**
- 2. // possiamo passare anche il numero della colonna**
- 3. rs.updateRow();**
- 4. // aggiorniamo il database**

# JDBC: RESULT-SET

**Per aggiungere una o più righe dobbiamo prima spostarci con il cursore:**

**1. rs.moveToInsertRow(); // sposta il cursore per inserire una riga**

**Ora possiamo aggiornare i valori della nuova riga:**

**1. rs.updateString("name", "Venkat"); //aggiorna il tipo string  
2. rs.updateString("position", "DBA"); //aggiorna il tipo string  
3. rs.updateDouble("salary", 925.0); // aggiorna il tipo double**

**Infine possiamo inserire la nuova riga al database:**

**1. rs.insertRow(); // aggiorna il database con la nuova riga**

**Adesso possiamo spostarci alla riga a cui ci trovavamo prima del metodo moveToInsertRow():**

**1. rs.moveToCurrentRow(); // sposta il cursore a prima di moveToInsertRow()**

# JDBC: RESULT-SET

**Per rimuovere una riga invece bisogna posizionarsi con il cursore alla riga interessata e poi eliminarla:**

```
1.rs.absolute(2); // sposta il cursore alla riga indicata  
2.rs.deleteRow();
```

**Il recupero dei record di un result-set da una QUERY avviene tutto contemporaneamente in memoria.**

**Possiamo indicare il numero di righe massimo da caricare in memoria contemporaneamente:**

```
1.PreparedStatement pstmt = dbConnection.prepareStatement(QUERY);  
2.pstmt.setFetchSize(10);  
3.ResultSet rs = pstmt.executeQuery();
```

**NOTA: ad esempio un risultato di 100 righe verrà visualizzato con 10 cicli da 10 righe di lettura al database. È possibile usare la stessa istruzione per indicare un limite di memoria al result-set.**

**Esercizio:**

**Usando jdbc recuperare e stampare una view che mostri le città italiane presenti nel database world, stampando la view sfruttando le informazioni ricavate dai metadati (quindi con un metodo di stampa generalizzato).**

**Esercizio:**

**Aggiungere tramite JDBC 10 città italiane non presenti nella tabella city di world con anche inserendo i corrispettivi dati.**

**Da svolgere senza le query ma usando i metodi del ResultSet (JDBC).**

