Corso SQL e Database relazionali

Giuseppe ing. Grosso

Aeffegroup SRL

Via Dante Alighieri, 72

Indice generale

Introduzione

L’obiettivo del corso è quello di fornire agli allievi le competenze necessarie per poter progettare, implementare ed interrogare basi di dati relazionali utilizzando il linguaggio SQL; i concetti trattati sono applicabili ai più diffusi DBMS relazionali come Oracle, SQL Server, MySQL e PostgreSQL

Sintesi argomenti trattati:

- Introduzione ai database relazionali
- Approfondimento dei concetti principali: tabelle, campi, chiavi, relazioni, indici, vincoli, trigger ed integrità dei dati.
- Il disegno e l’implementazione di una base dati relazionale: regole e best practice.
- La sintassi del linguaggio SQL per la creazione di una base dati.
- La sintassi del linguaggio SQL per l’interrogazione di una base dati: particolare attenzione alle query con inner join, left join, full join.
- Tecniche avanzate per l’interrogazione di una base dati: ordinamento, raggruppamento, join tra tabelle.
- L’indicizzazione del database; analisi e miglioramento delle performance delle interrogazioni SQL.
- Introduzione alle transazioni.
- Introduzione all’SQL procedurale (PL/SQL, TSQL).

Requisiti ingresso

E’ richiesta la conoscenze delle nozioni di informatica di base.

Introduzione ai database relazionali.

Argomenti

- I Data base relazionali
- Tabelle e campi di un database
- Relazioni fra le tabelle

Base dati

- Una **base di dati** (BD) o *Data Base* (DB) è un sistema di archivi (file) organizzati per consentirne una facile consultazione
- DBMS (DB Management System) è il programma per gestire il DB
- Linguaggi per i DB
    - Data Definition Language (DDL)
    - Data Manipulation Language (DML)
    - Query Language (QL)
- Diversi tipi di organizzazione di DB:
    - Antichi: gerarchico, reticolare, a liste invertite, …
    - Attuali: relazionale

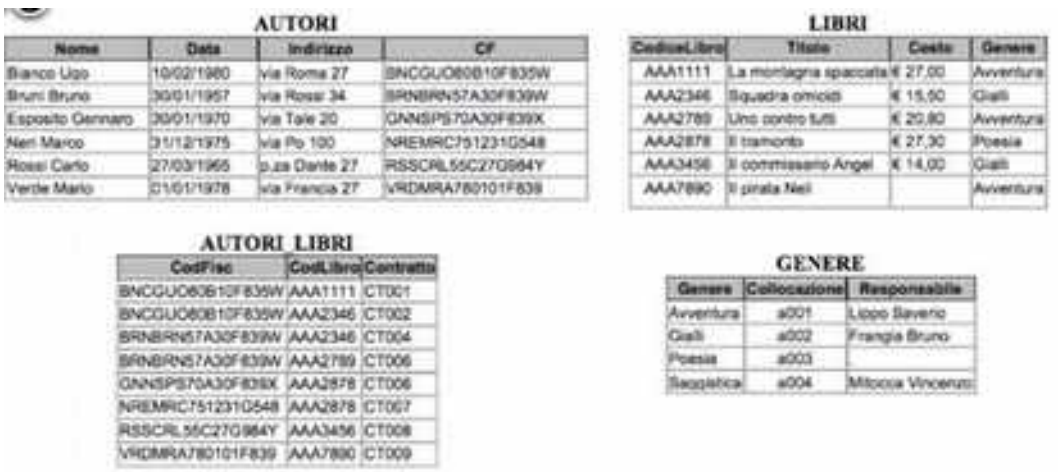
Relazioni e tabelle

- DB relazionale: l'organizzazione del DB basa sul concetto matematico di **"relazione"** che concretamente significa: **i dati sono organizzati in tabelle**

Ciclo di vita

- Progettazione
- Popolamento
- Interrogazione
- Aggiornamento dei dati
- Manutenzione

Es. di Database:



AUTORI

Nome	Data	Indirizzo	CF
Bianco Ugo	10/02/1980	via Roma 27	BNCGUO80B10F835W
Bruni Bruno	30/01/1957	via Rossi 34	BRNBRN57A30F839W
Esposito Gennaro	30/01/1970	via Tale 20	GNNSPS70A30F839X
Neri Marco	31/12/1975	via Po 100	NREMRC751231G548
Rossi Carlo	27/03/1965	p.za Dante 27	RSSCRL55C27G984Y
Verde Mario	01/01/1978	via Francia 27	VRDMRA780101F839

LIBRI

CodiceLibro	Titolo	Costo	Genere
AAA1111	La montagna spaccata	€ 27,00	Avventura
AAA2346	Squadra omicidi	€ 15,50	Gialli
AAA2789	Uno contro tutti	€ 20,80	Avventura
AAA2878	Il tramonto	€ 27,30	Poesia
AAA3456	Il commissario Angel	€ 14,00	Gialli
AAA7890	Il pirata Nel		Avventura

AUTORI_LIBRI

CodFisc	CodLibro	Contratto
BNCGUO80B10F835W	AAA1111	CT001
BNCGUO80B10F835W	AAA2346	CT002
BRNBRN57A30F839W	AAA2346	CT004
BRNBRN57A30F839W	AAA2789	CT006
GNNSPS70A30F839X	AAA2878	CT006
NREMRC751231G548	AAA2878	CT007
RSSCRL55C27G984Y	AAA3456	CT008
VRDMRA780101F839	AAA7890	CT009

GENERE

Genere	Collocazione	Responsabile
Avventura	a001	Lippo Saverio
Gialli	a002	Frangia Bruno
Poesia	a003	
Saggistica	a004	Milocca Vincenzo

Tabelle e campi di un Data Base

- Caratteristiche fondamentali del dato
  - Tipo: insieme di valori possibili e loro proprietà
  - Valore: il valore assunto
  - Attributo (nome): dà un significato al dato
- Esempi: tipo del “buon senso” e tipi del DBMS

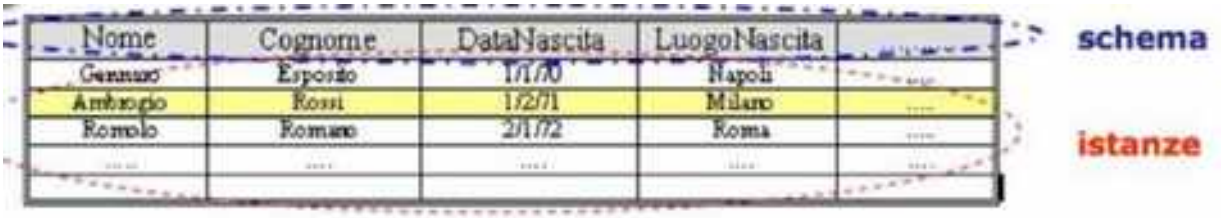
ATTRIBUTO	VALORE	TIPO	Nome
Radice dell'equazione	3.8	Numero reale	Radice
Numero di clienti	780	Numero intero	NumeroClienti
Cliente	Carlo Rossi	Nome di persona	Cliente
Prezzo in Euro	2970,00	Numero con 2 decimali	Prezzo
Codice del materiale	HP 29	Codice di magazzino	CODICE
Data di scadenza	31/12/2006	Data	SCADENZA
Sesso dell'impiegato	F	(M,F)	SESSO

Campi e record

La tabella come rappresentazione del concetto matematico di relazione fra insiemi

R ⊆ D1 × D2 × , … × Dn (Prodotto cartesiano)

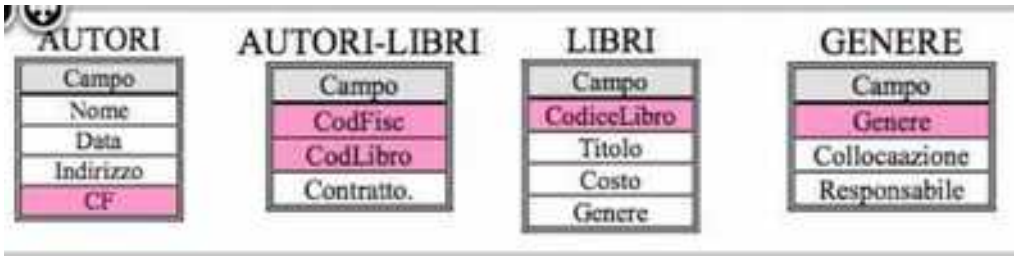
- Schema e istanza di una tabella
- Schema e istanze di una Base dati relazionale



Chiave primaria

- Valore Nullo. Un valore non definito di un campo
- Chiave primaria: Un valore che identifica univocamente un record
- Un valore-chiave

- Non può essere nullo
- Non può essere ripetuto
- Può essere “semplice” o “multiplo”



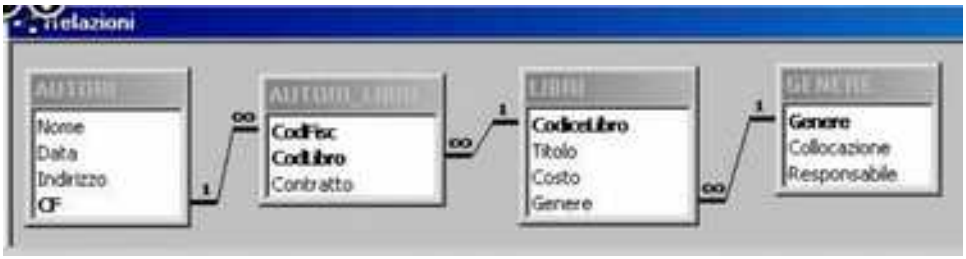
Viene colorato il campo chiave primario.

Campi-indice

- Un **indice** ordina “logicamente” i record di una tabella rispetto ad un campo, detto **campo-indice** o indicizzato.
- La tabella mantiene il suo ordinamento fisico, ma è associata ad un “indice” che mette in corrispondenza gli indirizzi dei record della tabella con i valori del campo-indice e viene ordinato rispetto a questo.
- L’indice velocizza la ricerca dei valori del campo.
- La chiave primaria è un indice, ma a questo se ne possono aggiungere altri.
- Possono essere indicizzati anche campi “non chiave”, cioè con valori duplicati ammessi (al contrario delle chiavi)
- Il vantaggio di velocità nella ricerca di un valore del campo-indice si paga con il tempo necessario per la generazione dell’indice. Indicizzare o no un campo è dunque una delicata scelta di progetto

Relazioni fra tabelle

- Relazioni 1 a molti: un record di una tabella contiene dati che interessano molti record di un’altra
- Relazioni 1 a 1: un record di una tabella contiene dati che interessano 1 record di un’altra (l’una è la “continuazione” dell’altra)
- Il lato “1″ della relazione è una chiave primaria
- Il lato “molti” (indicato con ∞ da ACCESS) si dice **Chiave esterna**



Integrità referenziale

- Le relazioni fra tabelle si “dichiarano” nel DBMS
  - o direttamente
  - o dichiarando le chiavi esterne

- Così, il DBMS può controllare il rispetto delle **Integrità referenziale**: **La chiave primaria di una relazione fra tabelle non si può cancellare**

Approfondimento dei concetti principali.

Tabelle.

Una **tabella** è una collezione di dati correlati fra loro, consiste in una serie di colonne e di righe. Ogni colonna rappresenta i campi della tabella, questi sono definiti in sede di progettazione della tabella. Mentre le righe rappresentano i valori associati ai campi, questi ultimi vengono inseriti in numero variabile e rappresentano le varie righe di contenuto della tabella.
Creazione di un database in mySQL:

```
CREATE DATABASE nome_database
```

Definizione di una tabella in MySQL.

```
CREATE table nome_tabella (
    nome_colonna tipo_colonna [ clausola_default ] [ vincoli_di_colonna ]
    [ , nome_colonna tipo_colonna [ clausola_default ] [ vincoli_di_colonna ] … ]
    [ , [ vincolo_di tabella] … ] )
```

Campi.

nome_colonna: è il nome della colonna che compone la tabella. Sarebbe meglio non esagerare con la lunghezza degli identificatori di colonna, dal momento che l'SQL Entry Level prevede nomi non piu' lunghi di 18 caratteri. Si consulti comunque la documentazione dello specifico database. I nomi devono iniziare con un carattere alfabetico.
tipo_colonna: è l'indicazione del tipo di dato che la colonna potra' contenere. I principali tipi previsti dallo standard SQL sono:

- CHARACTER(n)
  Una stringa a lunghezza fissa di esattamente n caratteri. CHARACTER puo' essere abbreviato con CHAR
- CHARACTER VARYING(n)
  Una stringa a lunghezza variabile di al massimo n caratteri. CHARACTER VARYING puo' essere abbreviato con VARCHAR o CHAR VARYING.
- INTEGER
  Un numero intero con segno. Puo' essere abbreviato con INT. La precisione, cioe' la grandezza del numero intero che puo' essere memorizzato in una colonna di questo tipo, dipende dall'implementazione del particolare DBMS.
- SMALLINT
  Un numero intero con segno con precisione non superiore a INTEGER.
- FLOAT(p)
  Un numero a virgola mobile, con precisione p. Il valore massimo di p dipende dall'implementazione del DBMS. E' possibile usare FLOAT senza indicazione della

precisione, utilizzando quindi la precisione di default, anch'essa dipendente dall'implementazione. REAL e DOUBLE PRECISION sono dei sinonimi per un FLOAT con una particolare precisione. Anche in questo caso le precisioni dipendono dall'implementazione, con il vincolo che la precisione del primo non sia superiore a quella del secondo.

- DECIMAL(p,q)
  Un numero a virgola fissa di almeno p cifre e segno, con q cifre dopo la virgola. DEC e' un'abbreviazione per DECIMAL. DECIMAL(p) e' un'abbreviazione per DECIMAL(p,0). Il valore massimo di p dipende dall'implementazione.
- INTERVAL
  Un periodo di tempo (anni, mesi, giorni, ore, minuti, secondi e frazioni di secondo).
- DATE, TIME e TIMESTAMP
  Un preciso istante temporale. DATE permette di indicare l'anno, il mese e il giorno. Con TIME si possono specificare l'ora, i minuti e i secondi. TIMESTAMP e' la combinazione dei due precedenti. I secondi sono un numero con la virgola, permettendo cosi' di specificare anche frazioni di secondo.

clausola_default: indica il valore di default che assumerà la colonna se non gliene viene assegnato uno esplicitamente nel momento della creazione della riga. La sintassi da utilizzare e' la seguente:

DEFAULT { valore | NULL }

dove, valore e' un valore valido per il tipo con cui la colonna e' stata definita.

vincoli_di_colonna: sono vincoli di integrita' che vengono applicati al singolo attributo. Sono:

- NOT NULL, che indica che la colonna non puo' assumere il valore NULL.
- PRIMARY KEY, che indica che la colonna e' la chiave primaria della tabella.
- una definizione di riferimento, con cui si indica che la colonna e' una chiave esterna verso la tabella e i campi indicati nella definizione. La sintasi e' la seguente:

  REFERENCES nome_tabella [(colonna1 [, colonna2 …])]
  [ON DELETE { CASCADE | SET DEFAULT | SET NULL }]
  [ON UPDATE { CASCADE | SET DEFAULT | SET NULL }]

  Le clausole ON DELETE e ON UPDATE indicano quale azione deve essere compiuta nel caso in cui una tupla nella tabella referenziata venga eliminata o aggiornata. Infatti in tali casi nella colonna referenziante (che e' quella che si sta definendo) potrebbero esserci dei valori inconsistenti. Le azioni possono essere:

  - CASCADE: eliminare la tupla contenente la colonna referenziante (nel caso di ON DELETE) o aggiornare anche la colonna referenziante (nel caso di ON UPDATE).
  - SET DEFAULT: assegnare alla colonna referenziante il suo valore di default.
  - SET NULL: assegnare alla colonna referenziante il valore NULL.

un controllo di valore, con il quale si permette o meno l'assegnazione di un valore alla colonna, in base al risultato di un espressione. La sintassi da usare e':
CHECK (espressione_condizionale)

dove espressione_condizionale e' un'espressione che restituisce vero o falso.
Ad esempio, se stiamo definendo la colonna COLONNA1, definendo il seguente controllo:

CHECK (COLONNA1 < 1000)

in tale colonna potranno essere inseriti solo valori inferiori a 1000.

Chiavi.

vincolo_di_tabella: sono vincoli di integrita' che possono riferirsi a piu' colonne della tabella. Sono:

- la definizione della chiave primaria:

  PRIMARY KEY (colonna1 [, colonna2 …])

  Si noti che in questo caso, a differenza della definizione della chiave primaria come vincolo di colonna, essa puo' essere formata da piu' di un attributo.

Relazioni.

Mysql come altri database permette la definizioni delle chiavi esterne:

FOREIGN KEY (colonna1 [, colonna2 …]) definizione_di_riferimento

La definizione_di_riferimento ha la stessa sintassi e significato di quella che puo' comparire come vincolo di colonna.

- un controllo di valore, con la stessa sintassi e significato di quello che puo' essere usato come vincolo di colonna.

Indici.

La creazione di indici in un database MySQL permette di evitare che ogni ricerca sia preceduta da una scansione completa delle tabelle utilizzate (*full table scan*). L'indicizzazione è stata ideata appositamente per velocizzare l'esecuzione delle query di selezione e viene introdotta semplicemente utilizzando l'apposito comando, `CREATE INDEX`, seguito dal nome del campo o dei campi interessati alla generazione degli indici:

```
mysql> CREATE INDEX nome_cognome ON tbl (nome,cognome);
```

In alternativa è possibile creare un indice anche per alterazione della tabella:

```
mysql> ALTER TABLE tbl ADD INDEX nome_cognome (nome,cognome);
```

MySQL consente di creare fino a 16 indici all'interno di una stessa tabella, sono inoltre supportati indici su più colonne, indici multipli relativi a più colonne e indici per ricerche full-text. In pratica, con gli indici effettueremo le stesse operazioni che si svolgono comunemente in una biblioteca quando si ordinano i vari testi per titolo, autore, argomento ecc., evitando così di dover passare in rassegna tutti i libri ogni volta che si presenta la necessità di consultarne uno solo o una parte di essi.

Vincoli.

I **constraint** (o vincoli), sono delle regole che vengono stabilite all'interno di una tabella che servono ad evitare che l'utente possa inserire dati non conformi ad alcune regole. Queste regole possono preservare l'inserimento dei dati o l'integrità referenziale tra le tabelle.

Possono essere dichiarati sia in fase di creazione tabella (`CREATE TABLE`), sia in fase di modifica della tabella (`ALTER TABLE`). Se il constraint è applicato ad una singola colonna allora è detto constraint di colonna, altrimenti se è applicato a più di una colonna è detto constraint di tabella.

Durante la dichiarazione del vincolo possiamo associargli un nome (usando la clausola `CONSTRAINT`), in caso contrario sarà Oracle a definirne uno di sistema (inizierà sempre con il prefisso `SYS_`). I vincoli vengono memorizzati da Oracle nel dizionario dei dati.

Nota: In tutti gli esempi riportati in questo capitolo usiamo l'utente "myself" creato nelle lezioni precedenti.

Constraint NOT NULL

Può essere definito **esclusivamente su una singola colonna** della tabella. La sua funzione è quella di evitare che l'utente possa inserire valori nulli (`NULL`) nella colonna associata, pertanto questa dovrà sempre contenere un valore diverso da `NULL`.

Sinstassi del vincolo NOT NULL

<nome colonna> <tipo> CONSTRAINT <nome vincolo> NOT NULL

Sinstassi del vincolo NOT NULL (anonimo)

<nome colonna> <tipo> NOT NULL

Per esempio creiamo una tabella denominata "myTableNull" avente due colonne, entrambe "`NOT NULL`" e ne visualizziamo la struttura.

```
CREATE TABLE myTableNull
(
   Cognome VARCHAR2(15) CONSTRAINT nn_myTableNull_Cognome NOT NULL,
   Nome VARCHAR2(15) CONSTRAINT nn_myTableNull_Nome NOT NULL
);
```

trigger ed integrità dei dati.

Regole aziendali. Ulteriori vincoli, detti **vincoli di integrità generici** in quanto non legati al modello relazionale, sono quelli imposti dalle regole aziendali. Tali vincoli possono essere rappresentati in SQL in più modi: mediante il costrutto `check` nella definizione di una tabella, mediante le asserzioni, oppure attraverso l'uso di regole attive (*trigger*).

E' bene chiarire in anticipo che i vincoli di integrità generici rappresentano un argomento contrastato. A differenza dei vincoli relazionali, gli strumenti per specificare vincoli generici non sono stabilmente inseriti nello standard SQL (ad esempio i trigger sono stati aggiunti sono nell'ultima versione di SQL dopo essere stati disponibili per molto tempo nei DBMS). Di conseguenza, mentre i vincoli tipici del modello relazionale sono supportati efficientemente da tutti i DBMS relazionali, gli strumenti per specificare vincoli generici variano notevolmente tra i vari

DBMS disponibili e non sempre garantiscono l'efficienza del sistema risulante. E' quindi fortemente consigliato accertarsi di quali siano e di come funzionino gli strumenti per vincoli generici supportati dal DBMS prescelto.

Il costrutto `check` permette di specificare, mediante una condizione come quella che può apparire nella clausola `where` di una interrogazione SQL, vincoli generici a livello di di tabella o, mediante le asserzioni, a livello di schema di base di dati.

Si noti che un uso indiscriminato di questi vincoli appesantisce il carico del sistema in quanto, solitamente, i DBMS garantiscono una implementazione efficiente solo per i vincoli propri del modello relazionale.

Vediamo alcuni esempi. Supponiamo di avere le tabelle dipendente, teatro e lavoro. La tabella lavoro associa i dipendenti ai relativi teatri. Un teatro può avere più dipendenti e un dipendente può lavorare per più teatri. Supponiamo di voler esprimere un **vincolo massimo di partecipazione** per un dipendente rispetto alla relazione lavoro: un dipendente non può lavorare per più di due teatri. Questo vincolo può essere specificato sulla tabella lavoro nel seguente modo:

```
create table teatro
(
  nome             varchar(20) primary key,
  indirizzo        varchar(40) not null
)

create table dipendente
(
  cf                    char(16) primary key,
  nome                  varchar(20) not null,
  cognome               varchar(20) not null,
  dataDiNascita         date,
  luogoDiNascita        varchar(30),
  capo                  char(16),
  foreign key capo references dipendente(cf)
)

create table lavoro
(
  teatro              varchar(20),
  dipendente          char(16),
  primary key(teatro, dipendente),
  foreign key teatro references teatro(nome),
  foreign key dipendente references dipendente(cf),
  check(2 >= (select count(*)
              from lavoro L
              where dipendente = L.dipendente))
)
```

Il vincolo afferma che per ogni dipendente non ci possono essere più di due righe nella tabella lavoro, quindi più di due teatri per cui il dipendente lavoro. Nella query di definizione del vincolo si può usare il nome degli attributi sui quali si sta definenendo il vincolo (dipendente in questo caso).

Lo stesso vincolo può essere espresso mediante una **asserzione**. Solitamente viene scritta una interrogazione SQL che seleziona le righe della base di dati che violano il vincolo. La condizione dell'asserzione viene fomata mettendo tale interrogazione come argomento del predicano `not exists`. Dunque il vincolo di integrità specificato dall'asserzione è verificato che il risultato della interrogazione è vuoto, cioè se non esistono righe che violano il vincolo. Vediamo un esempio:

```
create assertion limitaImpieghi check (not exists(

  select dipendente
  from lavoro
  group by dipendente
  having count(*) > 2

))
```

Supponiamo ora di voler affermare il seguente **vincolo minimo di partecipazione** per un dipendente rispetto alla relazione lavoro: ogni dipendente deve essere assunto presso almeno un teatro. Possiamo imporre questo vincolo sull'attributo `cf` della tabella dipendente:

```
create table dipendente
(
  cf                    char(16) primary key,
  nome                  varchar(20) not null,
  cognome               varchar(20) not null,
  dataDiNascita         date,
  luogoDiNascita        varchar(30),
  capo                  varchar(20),
  foreign key capo references dipendente(cf),
  check (cf in (select dipendente from lavoro))
)
```

Lo stesso vincolo può essere espresso mediante una asserzione come segue:

```
create assertion disoccupato check (not exists (

  select cf
  from dipendente
  where cf not in (select dipendente from lavoro)

))
```

Vediamo un vincolo che coinvolge più attributi della stessa tabella. Ad esempio, supponiamo di voler affermare che i dipendenti nati a Milano devono essere nati prima del 1970. Possiamo riscrivere la definizione della tabella dipendente come segue:

```
create table dipendente
(
  cf                    char(16) primary key,
  nome                  varchar(20) not null,
  cognome               varchar(20) not null,
  dataDiNascita         date,
  luogoDiNascita        varchar(30),
  capo                  varchar(20),
  foreign key capo references dipendente(cf),
  check (luogoDiNascita <> "Milano" or
         dataDiNascita < '1970-01-01')
)
```

Vediamo un esempio di vincolo che coinvolge più tabelle. Supponiamo di voler specificare il seguente vincolo minimo di partecipazione: un teatro deve avere almeno 5 dipendenti. Possiamo scrivere la seguente asserzione:

```
create assertion vincoloDipendentiTeatro check (

  not exists (select nome
              from teatro
```

```
              where nome not in (select teatro from lavoro))

  and

  not exists (select count(*)
              from lavoro
              group by teatro
              having count(*) < 5)

)
```

Il vincolo asserisce che non esistono teatri privi di dipendenti e, tra quelli che hanno almeno un dipendente, non esistono teatri con meno di 5 dipendenti. Dunque tutti i teatri hanno almeno 5 dipendenti.

Inoltre, vediamo un vincolo sulla cardinalità di una tabella. La seguente asserzione afferma che ci devono essere almeno 3 teatri nella rete:

```
create assertion vincoloTeatriRete check (

  3 <= (select count(*) from teatro)

)
```

Le **regole attive** (*trigger*) permettono di *gestire* i vincoli di integrità. La differenza rispetto agli strumenti fin ora introdotti per specificare vincoli di integrità (relazionali o generici) è la seguente: un trigger specifica una azione da intraprendere qualora in vincolo non sia soddisfatto, solitamente una azione riparatrice della integrità violata.

Un trigger segue il **paradigma evento-condizione-azione**: se un certo evento si verifica, la relativa condizione viene controllata e, se soddisfatta, l'azione viene intrapresa. Un evento è solitamento un aggiornamento della base di dati (insert, update, delete). Una condizione è un predicato espresso in SQL. Una azione è una interrogazione SQL (solitamente di aggiornamento della base di dati) oppure una eccezione che annulla gli effetti dell'operazione che ha attivato il trigger riportando la base di dati allo stato precedente a tale operazione (*rollback*). Il trigger può essere attivato prima o dopo l'evento.

Si noti che ci possono essere più trigger associati ad un evento. L'**ordine di esecuzione** dei trigger in tal caso è gestito dal sistema e generalmente tiene conto dell'ordine di creazione dei trigger. Un trigger che come azione aggiorna lo stato della base di dati può a sua volta innescare altri trigger, che a loro volta possono attivare altri trigger, con la possibilità di avere **reazioni a catena infinite**. Inoltre, l'azione di un trigger può violare vincoli di integrità. La violazione di un vincolo di integrità di chiave esterna può causare, come conseguenza delle politiche di gestione di tali vincoli, ulteriori modifiche alla base di dati che al loro volta possono scatenare altri trigger, oppure violare altri vincoli di integrità. Si badi bene che la violazione di un vincolo di integrità non gestito, a qualsiasi livello della catena di attivazione, produce un **annullamento degli effetti** di tutte le operazioni innescate dalla primitiva madre che ha generato la catena di trigger, compresi gli effetti della primitiva madre stessa. I trigger sono dunque strumenti semplici da scrivere in modo indipendente ma difficili da gestire in modo integrato.

Supponiamo di voler specificare un trigger per il vincolo che afferma che lo stipendio di un dipendente non può essere incrementato più del 20%:

```
create trigger LimitaIncrementoStipendio
```

```
after update of stipendio on dipendente
for each row
when (New.stipendio > Old.Stipendio * 1.2)
update dipendente
set New.stipendio = Old.Stipendio * 1.2
where cf = New.cf
```

Il trigger LimitaIncrementoStipendio viene attivato dall'evento modifica (update) dello stipendio di un dipendende. Per ogni riga modificata, se il nuovo stipendio è stato incrementato più del 20% rispetto al vecchio (condizione when), allora lo stipendio viene incrementato del massimo possibile senza violare il vincolo di integrità. Si noti che è possibile usare le variabili di tupla `New` e `Old` per riferirsi, rispettivamente, alla tupla dopo e prima la modifica. Per gli eventi di inserimento, solo New è accessibile, per gli eventi di cancellazione, solo Old è accessibile.

Vediamo un altro esempio. Vogliamo modellare la regola che dice che una prenotazione per uno spettacolo può essere effettuata solo se vi sono ancora posti a disposizione in sala. Usiamo i seguenti quattro trigger:

```
create trigger disponibilità-1
after insert on messaInScena
for each row
update messaInScena
set postiDisponibili = (select capienza
                        from spazio
                        where nome = New.spazio)
where (data = New.data and
       ora = New.ora
       spazio = New.spazio)

create trigger disponibilità-2
after insert on prenotazione
for each row
update messaInScena
set postiDisponibili = postiDisponibili - 1
where (data = New.dataSpettacolo and
       ora = New.oraSpettacolo
       spazio = New.spazioSpettacolo)


create trigger disponibilità-3
after delete on prenotazione
for each row
update messaInScena
set postiDisponibili = postiDisponibili + 1
where (data = Old.dataSpettacolo and
       ora = Old.oraSpettacolo
       spazio = Old.spazioSpettacolo)

create trigger disponibilità-4
before insert on prenotazione
for each row
when (0 = (select postiDisponibili
           from messaInScena
           where (data = New.dataSpettacolo and
                  ora = New.oraSpettacolo
                  spazio = New.spazioSpettacolo)))
rollback("Posti esauriti")
```

Per specificare la regola aziendale sui posti disponibili abbiamo usato i seguenti trigger:

1. disponibilità-1, che imposta il numero di posti disponibili alla capienza dello spazio teatrale quando uno spettacolo viene inserito;
2. disponibilità-2, che decrementa di uno i posti disponibili quando una prenotazione viene inserita;
3. disponibilità-3, che incrementa di uno i posti disponibili quando una prenotazione viene cancellata;
4. disponibilità-4, che controlla, *prima* dell'inserimento della prenotazione nella base di dati, se esistono posti disponibili. Se non ne esistono, esso annulla l'operazione di inserimento e avvisa l'utente che i posti sono esauriti.

Si noti che la soluzione funziona assumendo che disponibilità-4 venga eseguito prima di disponibilità-2 (di solito è così in quanto disponibilità-4 è di tipo before e disponibilità-2 è di tipo after).

I trigger sono anche utili per specificare le regole di calcolo degli **attributi calcolati**. Supponiamo che il prezzo ridotto di uno spettacolo debba essere scontato del 20% rispetto a quello intero. Dunque l'attributo prezzo ridotto è calcolato rispetto al prezzo intero. I trigger per gestire questo vincolo seguono:

```
create trigger CalcolaPrezzoRidottoInsert
after insert on messaInScena
for each row
update messaInScena
set New.prezzoRidotto = New.prezzoIntero * 0.8
where codice = New.codice

create trigger CalcolaPrezzoRidottoUpdate
after update of prezzoIntero on messaInScena
for each row
update messaInScena
set New.prezzoRidotto = New.prezzoIntero * 0.8
where codice = New.codice
```

Non tutti i vincoli di integrità possono essere descritti a livello di schema in SQL. Solitamente, quando un vincolo non è descrivibile in SQL, esso viene catturato a livello di applicazione implementandolo in qualche linguaggio di programmazione. E' bene che tutti i vincoli esprimibili in SQL vengano definiti a livello di schema in modo da renderli condivisi da tutte le applicazioni invece che replicare il vincolo per ogni applicazione. In tal modo le modifiche di un vincolo sono gestite a livello di schema senza modificare le applicazioni. Si parla in tal caso di **indipendenza dalla conoscenza**, dove per conoscenza si intende l'insieme delle regole codificate nei vincoli che regolano l'integrità della base.

E' possibile aggiungere e rimuovere vincoli di integrità definiti su una tabella mediante il comando `alter`. Per rimuovere un vincolo occorre averlo definito per nome mediante il costrutto `constraint`. Ad esempio:

```
create table dipendente
(
  cf                    char(16) primary key,
  nome                  varchar(20) not null,
  cognome               varchar(20) not null,
  indirizzo             varchar(30),
  constraint chiaveCandidata unique(nome, cognome)
)
```

```
alter table dipendente drop constraint chiaveCandidata
alter table dipendente add constraint chiaveCandidata unique(indirizzo)
```

Per rimuovere una asserzione o un trigger occorre usare il comando `drop` seguito dal nome del costrutto.

Concludiamo la parte sulla definizione dei dati in SQL parlando brevemente del **catalogo dei dati**. Il catalogo dei dati è una base relazionale per archiviare lo schema fisico di una base di dati; tale base contiene una descrizione dei dati e non i dati veri e propri. Ad esempio, il catalogo dei dati contiene una tabella per gli attributi delle tabelle di uno schema fisico. Ogni riga della tabella specifica, tra l'altro, il nome dell'attributo, la tabella di appartenenza, il suo valore di default e l'obligatorietà.

Il catalogo dei dati viene solitamente mantenuto dal DBMS e non deve essere creato o modificato dall'utente. Il catalogo dei dati può però essere interrogato dall'utente. Questo offre la possibilità interessante di costruire interrogazioni che accedano sia ai dati che ai metadati. E' bene che i dati e i metadati vengano organizzati nel medesimo modello dei dati (relazionale, ad oggetti, XML). In questo modo è possibile archiviare dati e metadati con le stesse strutture e interrogarli con lo stesso linguaggio. Questa caratteristica prende il nome di **riflessività**.

Il disegno e l’implementazione di una base dati relazionale: regole e best practice.

Non esistono soluzioni universali per la progettazione di un database, l’organizzazione di una banca dati è infatti una procedura dipendente dal tipo di applicazione che si desidera realizzare. Vi sono delle semplici regole che è possibile seguire al fine di strutturare basi di dati che consentano di archiviare e manipolare dati in modo semplice ed efficiente.

Nella progettazione di un database è fondamentale tenere presente alcune caratteristiche basilari:

- il numero delle tabelle che verranno coinvolte dalle interrogazioni lanciate tramite le applicazioni;
- il numero e il tipo di campi che verranno da creare all’interno delle tabelle;
- le relazioni interne ed esterne tra i campi.

Le relazioni sono il punto saliente della fase di progettazione, maggiore sarà il numero di relazioni che si sarà in grado di stabilire, più efficiente sarà la struttura del database, più breve sarà il codice da digitare per le applicazioni e inferiore sarà l’esigenza di generare nuovi campi o di memorizzare più volte gli stesso dati.

La logica della progettazione.

Progettare un database, in particolare nel caso dei database relazioni come quelli gestiti da un DBMS SQL server, significa definirne la struttura, le caratteristiche e i contenuti gestiti cercando di prevedere quelle che saranno le esigenze dell’applicazione che avrà accesso ai dati; già da questa prima affermazione è possibile introdurre una regola di valore generale: è buona norma progettare il database prima dell’applicazione e non il contrario, la motivazione di questa affermazione è

semplice: l’applicazione andrà a gestire i dati contenuti nella base di dati, quindi, migliore sarà la struttura della banca dati più efficiente sarà l’applicazione.

In pratica, progettare un database consiste nella **costruzione di un modello di realtà**, una base di dati non è altro che un archivio a cui è possibile inviare, attraverso un DBMS, delle interrogazioni sotto forma di query; le query non sono altro che delle “domande”, un database dotato di una struttura lineare sarà in grado di rispondere in modo semplice a domande semplici, a livello tecnico la possibilità di inviare semplici domande ottenendo il massimo risultato significa creare applicazioni dotate di codici meno complessi.

La fase di progettazione di un database non può naturalmente prescindere dal **tipo di applicazione** che si desidera realizzare, le applicazioni vengono scritte per degli scopi (ad esempio: un database manager che gestisca i post di un sito Web), quindi anche i database a cui esse si interfacciano dovranno perseguire gli stessi scopi (conservare i dati relativi alle news); gli scopi per cui si progetta un database possono essere suddivisi in due categorie:

1. scopi correlati ad esigenze operative: archiviare, mantenere e amministrare informazioni su entità animate (individui accomunati da specifiche caratteristiche), inanimate (oggetti, come dei prodotti) o astratte (ad esempio i prezzi dei prodotti);
2. scopi correlati ad esigenze decisionali: le decisioni possono essere prese esclusivamente sulla base delle informazioni disponibili, migliore sarà la struttura del database più immediato sarà l’accesso alle informazioni.

La progettazione di una base di dati necessita generalmente di più fasi la cui finalità è quella di generare un modello (o astrazione) in grado di fornire la rappresentazione delle informazioni gestite e delle relazioni esistenti tra di esse.

Le fasi della progettazione

La progettazione di un database relazione si basa su un **modello chiamato E\R (Entità\Relazione)**, dove con il termine di entità si identificano delle persone, degli oggetti (anche astratti) o eventi dotati di determinati attributi; gli attributi hanno il compito di definire quelle che sono le proprietà delle entità mentre le relazioni non sono altro che le correlazioni esistenti tra le entità. In un modello di rappresentazione della realtà sotto forma di database, le entità saranno indicate da tabelle e gli attributi da campi, i record (cioè le informazioni gestite) consentiranno di stabilire le relazioni esistenti sulla base di valori.

Con la prima fase della progettazione di un database secondo il modello E\R, si ha la cosiddetta “**analisi dei requisiti**“, i requisiti possono essere distinti in quattro categorie:

1. analisi delle caratteristiche dei dati;
2. analisi delle operazioni da effettuare sui dati;
3. analisi degli eventi che possono influenzare i dati;
4. analisi dei vincoli d’integrità, cioè delle proprietà relative a informazioni ed eventi.

Alla **fase descritta** deve seguire quella più attinente al modello E\R che consiste nell’elaborazione di una struttura che sia in grado di fornire una rappresentazione della realtà di cui il database sarà modello; le informazioni relative ai dati, alle entità, ai loro attributi e agli eventi che le coinvolgono

saranno già state raccolte nella fase precedente, la seconda fase impone di porre in relazione quanto analizzato.

A questo punto sarà possibile passare alla fase in cui sarà cioè necessario definire quali tabelle dovranno essere create, quali relazioni dovranno sussistere tra di esse e internamente ad esse e a quali vincoli saranno sottoposte le relazioni.

Fatto questo, si passerà alla fase della **realizzazione del database** attraverso il DBMS, i comandi inviati durante questa fase dipenderanno fondamentalmente dalle decisioni prese durante le fasi precedenti.

L'unico modo possibile per **mettere in relazione delle entità differenti** è quello di stabilire la presenza in essi di specifici elementi, si ha così un riferimento a quelle che sono le "regole di integrità" che sanciscono l'obbligo di prevedere elementi comuni in entità distinte; si pensi per esempio al Campionato di Calcio di Seria A, tutti i giocatori che vi partecipano sono calciatori di prima divisione, è però possibile suddividerli in gruppi sulla base di una relazione dovuta alla squadra di appartenenza.

Inoltre, sempre secondo le regole di integrità, deve essere possibile l'**accesso ai dati senza ambiguità**, quindi ogni singolo valore deve essere indirizzabile specificando il nome della relativa tabella, della colonna e della "chiave primaria" del record corrispondente; una chiave primaria non è altro che l'attributo di un'entità, in grado di identificare in modo univoco un elemento appartenete all'entità stessa. La targa attribuita ad un veicolo dalla motorizzazione civile è per esempio un identificativo univoco grazie al quale distinguere un'auto dal resto del parco macchine nazionale.

Esempio pratico di progettazione basata sul modello relazione

Fondamentalmente il modello E\R prevede tre tipi di relazioni:

1. uno ad uno: ad esempio il codice fiscale che identifica univocamente un cittadino rispetto all'Agenzia delle Entrate;
2. uno a molti: una stessa azienda emette più fatture per diversi clienti;
3. molti a molti: il calendario di un campionato di Calcio prevede che tutte le squadre giochino tra di loro.

Un **possibile esempio di relazioni tra entità** è quello della costruzione di un database per gestire i prestiti in una biblioteca, l'analisi dei requisiti farà emergere l'esigenza di creare un sistema che permetta la gestione di entità (libri, iscritti e prestiti) sulla base di eventi; il riferimento al modello E\R consentirà di stabilire una struttura in cui per i diversi libri posseduti sia definibile una relazione interna (tutti i libri in prestito sono in relazione così come lo sono quelli ancora disponibili) ed esterna (quella tra i libri e gli iscritti che li hanno presi in prestito o restituiti).

La prima tabella da creare sarà quindi quella relativa ai libri posseduti:

id_libro	titolo	autore	stato
1	L'amore del bandito	Massimo Carlotto	Buono
2	Il lupo e il filosofo	Mark Rowlands	Ottimo

L’identificativo univoco è quindi quello relativo al valore del campo “id_libro” che non potrà essere ripetuto, ma all’interno della tabella esistono anche altre relazioni: sarà possibile ricercare i libri relativi ad un determinato autore così come tutti quelli che sono conservati nello stesso stato.

Si passi ora alla tabella degli iscritti:

id_iscritto	nome	cognome	indirizzo	email
1	Giuseppe	Garibaldi	Via Teano	g.garibaldi@imille.it
2	Linus	Torvalds	Via Kernel	tor@valds.it

Anche in questo caso i record relativi agli iscritti potranno essere distinti in modo univoco grazie al relativo identificativo, in questo modo sarà possibile evitare ambiguità dovute ad eventuali omonimie.

Si passi ora alla tabella relativa ai prestiti e alle restituzioni:

id_prestito	Id_libro	id_iscritto	data_prestito	data_restituzione	reso
1	2	2	2009-09-08	2009-09-28	0
2	1	1	2009-09-05	2009-09-25	1

Essa permette di **mettere in relazione i diversi libri** con gli iscritti che li hanno presi in prestito, consente quindi di stabilire una relazione tra i record presenti nella tabella dei libri con quelli della tabella degli iscritti; inoltre essa consente di stabilire delle relazioni interne, sarà infatti possibile selezionare e contare:

- tutti i libri prestati ad un determinato iscritto;
- tutti gli iscritti che hanno preso in prestito un determinato libro o soltanto quelli che lo hanno restituito o solo quelli che non lo hanno ancora riportato;
- tutti i libri prestati o restituiti in una determinata data;
- tutti i libri in prestito così come quelli disponibili.

Quelli proposti sono soltanto alcuni esempi di relazioni concepibili nella progettazione di un database, la struttura di una banca dati può naturalmente essere implementata o semplificata sulla base dell’applicazione che si desidera realizzare.

Conclusioni

La progettazione di un database non è di per sé una procedura particolarmente complessa, ma può variare molto a seconda dei progetti del programmatore e delle esigenze del software che si dovrà interfacciare al DBMS per la manipolazione dei dati; in questa breve trattazione sono state esposte alcune regole di base per realizzare una banca dati con attenzione a quelle che sono le indicazioni del modello relazionale.

La sintassi del linguaggio SQL per la creazione di una base dati.

SQL (Structured Query Language, linguaggio di interrogazione strutturato) è un linguaggio per computer mirato all'archiviazione, alla modifica e al recupero dei dati che sono memorizzati all'interno di database relazionali. La prima apparizione dell'SQL risale al 1974, quando un gruppo di ricercatori IBM sviluppò il primo prototipo di database relazionale. La prima versione commerciale di database relazionale fu rilasciata da Relational Software che, successivamente, prese il nome di Oracle.

Per il linguaggio SQL esistono delle norme standard. Tuttavia, oggigiorno l'SQL può essere utilizzato sui principali RDBMS (Sistema Relazionale per la Gestione dei Database, Relational Database Management System) in diversi modi. Principalmente, ciò è dovuto a due motivi: 1) lo standard SQL e piuttosto complicato e l'implementazione dell'intero standard non è pratico e, 2) ogni fornitore di database vuole poter definire un modo in base al quale poter distinguere il suo prodotto da quello degli altri fornitori. In questo corso, tali differenze verranno segnalate dove appropriato.

In questo sito del corso SQL vengono elencati i comandi SQL utilizzati più frequentemente. È diviso nei seguenti paragrafi:

- **Comandi SQL**: le istruzioni SQL fondamentali per l'archiviazione, il recupero e la modifica dei dati all'interno di un database relazionale.
- **Modifica delle tabelle**: il modo in cui vengono utilizzate le istruzioni SQL per gestire le tabelle presenti nel database.
- **SQL avanzato**: i comandi SQL più avanzati.
- **Sintassi SQL**: una singola pagina in cui sono elencate le sintassi di tutti i comandi SQL presenti in questo corso.

Per ogni comando, la sintassi SQL verrà prima presentata, quindi spiegata e illustrata mediante un esempio. Alla fine del corso, si disporrà di una buona comprensione generale della sintassi SQL. Inoltre, si sarà in grado di scrivere le interrogazioni SQL utilizzando la sintassi corretta. La comprensione dei fondamenti dell'SQL risulta più semplice rispetto al controllo di tutte le difficoltà che caratterizzano questo linguaggio per database. Siamo convinti che anche l'utente arriverà alla stessa conclusione.

DDL(data definition Language).

Le tabelle rappresentano la struttura di base in cui vengono memorizzati i dati nel database. Nella maggior parte dei casi, nessun fornitore di database saprà mai in anticipo quali sono le esigenze di archiviazione dei dati dei suoi clienti. È quindi probabile che sia necessario creare le tabelle del database autonomamente. Molti strumenti per database consentono di creare le tabelle senza dover scrivere in SQL. Considerando, però, che le tabelle rappresentano dei contenitori in cui vengono inseriti tutti i dati, è importante includere la sintassi **CREATE TABLE**.

Prima di occuparsi completamente della sintassi SQL per **CREATE TABLE**, è consigliato approfondire meglio la comprensione della tabella. Le tabelle sono divise in righe e colonne. Ogni riga rappresenta una porzione dei dati e ogni colonna può essere vista come un rappresentante di un

componente di quella porzione di dati. Se, ad esempio, si dispone di una tabella per la registrazione delle informazioni dei clienti, nelle colonne possono essere contenute informazioni come nome, cognome, indirizzo, città, paese, data di nascita e così via. Come risultato, quando viene specificata una tabella, vengono inclusi i titoli delle colonne e il tipo di dati di una determinata colonna.

Varietà dei tipi di dati. Generalmente, i dati arrivano in diverse forme. Possono, infatti, essere numeri interi (come 1), numeri reali (come 0,55), una stringa (come “sql”), un’espressione data/ora (come “2000-GEN-25 03:22:22”) oppure possono essere espressi in formato binario. Quando viene specificata una tabella, è necessario specificare il tipo di dati associato a ciascuna colonna, vale a dire che, nell‘esempio riportato precedentemente, se il “nome” è di tipo char(50) ciò significa che rappresenta una stringa di 50 caratteri. Si noti che database relazionali diversi consentono tipi di dati diversi, quindi è sempre consigliato consultare prima un riferimento specifico del database.

Creazione delle tabelle.

La sintassi del linguaggio SQL per **CREATE TABLE** è

```
CREATE TABLE "nome_della_tabella"
("colonna_1" "tipo_di_dati_per_la_colonna_1",
"colonna_2" "tipo_di_dati_per_la_colonna_2",
... );
```

Quindi, se si sta creando una tabella clienti come quella specificata precedentemente, è necessario digitare:

```
CREATE TABLE Customer
(First_Name char(50),
Last_Name char(50),
Address char(50),
City char(50),
Country char(25),
Birth_Date datetime);
```

A volte, si può desiderare di fornire un valore predefinito a ogni colonna. Un valore predefinito viene utilizzato quando, al momento di inserire i dati nella tabella, non è stato specificato un valore della colonna. Per specificare un valore predefinito, aggiungere "Default [value]" dopo la dichiarazione del tipo di dati. Nell’esempio precedente, se alla colonna “Address” e “City” si desidera fornire un valore predefinito pari a, rispettivamente, “Unknown” e “Milan”, è necessario digitare:

```
CREATE TABLE Customer
(First_Name char(50),
Last_Name char(50),
Address char(50) default 'Unknown',
City char(50) default 'Milan',
Country char(25),
Birth_Date datetime);
```

È inoltre possibile limitare il tipo di informazioni che possono essere contenute in una tabella o in una colonna. Per realizzare questa operazione viene utilizzata la parola chiave CONSTRAINT.

Constraints.

Attraverso i vincoli è possibile limitare il tipo di dati che possono essere inseriti in una tabella. Tali vincoli possono essere specificati al momento della creazione della tabella mediante l'istruzione CREATE TABLE oppure, una volta che questa è già stata realizzata, mediante l'istruzione ALTER TABLE.

I tipi più comuni di vincoli sono i seguenti:

- **NOT NULL CONSTRAINT**: consente di assicurarsi che una colonna non può avere un valore NULL.
- **DEFAULT CONSTRAINT**: mediante questo vincolo è possibile stabilire un valore predefinito per una colonna, qualora non ne sia stato specificato uno.
- **UNIQUE CONSTRAINT**: consente di assicurarsi che tutti i valori presenti in una colonna siano diversi.
- **CHECK CONSTRAINT**: consente di assicurarsi che tutti i valori presenti in una colonna soddisfino determinati criteri.
- **PRIMARY KEY CONSTRAINT**: viene utilizzato solo per individuare una riga della tabella.
- **FOREIGN KEY CONSTRAINT**: viene utilizzato per garantire l'integrità referenziale dei dati.

NOT NULL CONSTRAINT.

Per impostazione predefinita, una colonna può contenere il valore NULL. Se si desidera evitare che una colonna contenga il valore NULL, è necessario posizionare un limite su questa colonna in cui sia specificato che NULL adesso non è un valore consentito.

Ad esempio, nella seguente istruzione:

```
CREATE TABLE Customer
(SID integer NOT NULL,
Last_Name varchar (30) NOT NULL,
First_Name varchar(30));
```

Nelle colonne "SID" e "Last_Name" non può essere incluso il valore NULL, mentre può essere incluso nella colonna "First_Name".

Un tentativo di eseguire la seguente istruzione SQL,

```
INSERT INTO Customer (Last_Name, First_Name) VALUES ('Romano', 'Stefano');
```

restituirà un errore poiché questa operazione, conducendo alla colonna "SID" che è NULL, viola il vincolo NOT NULL impostato per la colonna.

DEFAULT.

Quando l'istruzione INSERT INTO non fornisce un valore specifico, il vincolo **DEFAULT** consente di fornire un valore predefinito a una colonna. Ad esempio, se si crea una tabella come la seguente:

CREATE TABLE Student
(Student_ID integer Unique,
Last_Name varchar (30),
First_Name varchar (30),
Score DEFAULT 80);

e si esegue la seguente istruzione SQL,

INSERT INTO Student (Student_ID, Last_Name, First_Name) VALUES (10, 'Johnson', 'Rick');

La tabella verrà visualizzata nel seguente modo:

Student_ID	Last_Name	First_Name	Score
10	Johnson	Rick	80

Sebbene nell'istruzione **INSERT INTO** non è stato specificato un valore per la colonna "Score", viene assegnato un valore predefinito pari a 80 in quanto tale valore è stato già impostato come predefinito per questa colonna.

Il vincolo UNIQUE garantisce che tutti i valori presenti in una colonna siano diversi.

Ad esempio, nella seguente istruzione:

CREATE TABLE Customer
(SID integer Unique,
Last_Name varchar (30),
First_Name varchar(30));

la colonna "SID" dispone di un vincolo unique e non possono, quindi, esservi inclusi valori duplicati. Tale vincolo non viene conservato nelle colonne "Last_Name" e "First_Name". Quindi, se la tabella contiene già le seguenti righe:

SID	Last_Name	First_Name
1	Mancini	Stella
2	Costa	Mario
3	Ferrari	Paolo

l'esecuzione della seguente istruzione SQL,

INSERT INTO Customer VALUES ('3', 'Russo', 'Sara');

restituirà un errore in quanto "3" è già esistente nella colonna SID e il tentativo di inserire un'altra riga con quel valore viola il vincolo **UNIQUE**.

Si noti che una colonna che è specificata come chiave primaria deve anche essere univoca. Allo stesso tempo, una colonna che è univoca non necessariamente deve essere una chiave primaria. Inoltre, su una tabella possono essere definiti più vincoli **UNIQUE**.

Il vincolo **CHECK** assicura che tutti i valori presenti in una colonna soddisfino determinate condizioni. Una volta definito, se il nuovo valore soddisfa il vincolo **CHECK**, il database inserirà solo una nuova riga o ne aggiornerà una esistente. Il vincolo **CHECK** viene utilizzato per assicurare la qualità dei dati.

Ad esempio, nella seguente istruzione CREATE TABLE,

```
CREATE TABLE Customer
(SID integer CHECK (SID > 0),
Last_Name varchar (30),
First_Name varchar(30));
```

la colonna “SID” dispone di un vincolo. Nel suo valore devono solo essere inclusi numeri interi maggiori di 0. In questo modo, il tentativo di eseguire la seguente istruzione:

```
INSERT INTO Customer VALUES (-3, 'Prada', 'Lynn');
```

restituirà un errore poiché i valori di SID devono essere maggiori di 0.

Si noti che, questa volta, il vincolo **CHECK** non viene applicato da MySQL.

Una chiave primaria viene utilizzata solo per individuare ogni riga presente in una tabella. Essa può far parte dello stesso record corrente oppure essere un campo artificiale e non avere niente a che fare con il record corrente. Una chiave primaria può consistere di uno o più campi presenti in una tabella. Quando vengono utilizzati più campi come chiave primaria, viene assegnato loro il nome di chiave composta.

Le chiavi primarie possono essere specificate durante la creazione della tabella, mediante **CREATE TABLE** oppure modificando la struttura della tabella esistente, utilizzando ALTER TABLE.

A seguire vengono illustrati degli esempi in cui viene specificata una chiave primaria durante la creazione di una tabella:

MySQL:

```
CREATE TABLE Customer
(SID integer,
Last_Name varchar(30),
First_Name varchar(30),
PRIMARY KEY (SID));
```

Oracle:

```
CREATE TABLE Customer
(SID integer PRIMARY KEY,
Last_Name varchar(30),
First_Name varchar(30));
```

SQL Server:

```
CREATE TABLE Customer
(SID integer PRIMARY KEY,
Last_Name varchar(30),
First_Name varchar(30));
```

A seguire vengono illustrati degli esempi per specificare una chiave primaria mediante la modifica di una tabella:

MySQL:

ALTER TABLE Customer ADD PRIMARY KEY (SID);

Oracle:

ALTER TABLE Customer ADD PRIMARY KEY (SID);

SQL Server:

ALTER TABLE Customer ADD PRIMARY KEY (SID);

Nota: prima di utilizzare il comando **ALTER TABLE** per aggiungere una chiave primaria, è necessario assicurarsi che il campo sia definito come **NOT NULL**, vale a dire che **NULL** non può essere un valore accettato per quel campo.

Una chiave esterna rappresenta uno o più campi che fanno riferimento alla chiave primaria di un'altra tabella. Lo scopo della chiave esterna è garantire l'integrità referenziale dei dati. Cioè, sono consentiti solo i valori che si ritiene debbano apparire nel database.

Ad esempio, si supponga di disporre di due tabelle: una tabella ***CUSTOMER***, in cui sono inclusi tutti i dati dei clienti e una tabella ***ORDERS***, in cui sono contenuti tutti gli ordini dei clienti. Il vincolo impostato stabilisce che tutti gli ordini devono essere associati a un cliente presente nella tabella ***CUSTOMER***. In questo caso, verrà posizionata una chiave esterna sulla tabella ***ORDERS*** che sia in relazione con la chiave primaria della tabella ***CUSTOMER***. In questo modo, è possibile garantire che tutti gli ordini della tabella ***ORDERS*** sono correlati a un cliente presente nella tabella ***CUSTOMER***. Cioè, nella tabella ***ORDERS*** non possono essere contenute informazioni relative a un cliente che non è incluso nella tabella ***CUSTOMER***.

La struttura di queste due tabelle è la seguente:

Tabella ***CUSTOMER***

Nome di Colonna	Caratteristica
SID	Chiave Primaria
Last_Name	
First_Name	

Tabella ***ORDERS***

Nome di Colonna	Caratteristica
Order_ID	Chiave Primaria
Order_Date	
Customer_SID	Chiave Esterna
Amount	

Nell'esempio precedente, la colonna Customer_SID contenuta nella tabella ***ORDERS*** rappresenta una chiave esterna facente riferimento alla colonna SID della tabella ***CUSTOMER***.

A seguire, vengono illustrati degli esempi nei quali viene mostrato come specificare una chiave esterna durante la creazione della tabella ***ORDERS***:

MySQL:

```
CREATE TABLE ORDERS
(Order_ID integer,
Order_Date date,
Customer_SID integer,
Amount double,
PRIMARY KEY (Order_ID),
FOREIGN KEY (Customer_SID) REFERENCES CUSTOMER (SID));
```

Oracle:

```
CREATE TABLE ORDERS
(Order_ID integer PRIMARY KEY,
Order_Date date,
Customer_SID integer REFERENCES CUSTOMER (SID),
Amount double);
```

SQL Server:

```
CREATE TABLE ORDERS
(Order_ID integer PRIMARY KEY,
Order_Date datetime,
Customer_SID integer REFERENCES CUSTOMER (SID),
Amount double);
```

A seguire, vengono illustrati degli esempi per specificare una chiave esterna mediante la modifica di una tabella. Si supponga che la tabella ***ORDERS*** sia stata creata e che la chiave esterna non sia stata ancora inserita:

MySQL:

```
ALTER TABLE ORDERS
ADD FOREIGN KEY (Customer_SID) REFERENCES CUSTOMER (SID);
```

Oracle:

```
ALTER TABLE ORDERS
ADD (CONSTRAINT fk_orders1) FOREIGN KEY (Customer_SID) REFERENCES
CUSTOMER (SID);
```

SQL Server:

```
ALTER TABLE ORDERS
ADD FOREIGN KEY (Customer_SID) REFERENCES CUSTOMER (SID);
```

Le viste possono essere considerate come tabelle virtuali. Generalmente, una tabella è caratterizzata da una serie di definizioni ed è il luogo in cui vengono fisicamente memorizzati i dati. Anche una vista dispone di una serie di definizioni, generata nella parte superiore di una o più tabelle o viste, ma non rappresenta il luogo in cui i dati vengono fisicamente memorizzati.

La sintassi per creare una vista è la seguente:

```
CREATE VIEW "nome_di_vista" AS "istruzione_SQL ";
```

"istruzione_SQL" può essere una qualsiasi delle istruzioni SQL che sono state affrontate finora in questo corso.

Si veda questo semplice esempio. Si supponga di avere la seguente tabella:

Tabella ***Customer***

Nome di Colonna	Tipo di Dati
First_Name	char(50)
Last_Name	char(50)
Address	char(50)
City	char(50)
Country	char(25)
Birth_Date	datetime

e che si desideri creare una vista denominata ***V_Customer***, in cui sono contenute solo le colonne First_Name, Last_Name e Country da questa tabella. Digitare:

```
CREATE VIEW V_Customer
AS SELECT First_Name, Last_Name, Country
FROM Customer;
```

Adesso, si dispone di una vista denominata ***V_Customer*** caratterizzata dalla seguente struttura:

View ***V_Customer***

nome_di_colonna	Tipo di Dati
First_Name	char(50)
Last_Name	char(50)
Country	char(25)

Una vista può anche essere utilizzata per applicare le unioni a due tabelle. In questo caso, gli utenti visualizzano solo una vista anziché due tabelle e l'istruzione SQL che devono utilizzare diviene molto più semplice. Si supponga di disporre delle seguenti due tabelle:

Tabella ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

Tabella ***Geography***

Region_Name	Store_Name
East	Boston
East	New York
West	Los Angeles
West	San Diego

e che si desideri creare una vista in cui sono visualizzate le informazioni relative alle vendite per regione. In tal caso è necessario rilasciare la seguente istruzione SQL:

```
CREATE VIEW V_REGION_SALES
AS SELECT A1.Region_Bame REGION, SUM(A2.Sales) SALES
FROM Geography A1, Store_Information A2
WHERE A1.Store_Name = A2.Store_Name
GROUP BY A1.Region_Name;
```

In questo modo si dispone di una vista, ***V_REGION_SALES***, definita per memorizzare i record delle vendite realizzate in base alla regione. Se si desidera trovare il contenuto di questa vista, digitare:

```
SELECT * FROM V_REGION_SALES;
```

Risultato:

REGION	SALES
East	700
West	2050

Attraverso gli indici è possibile recuperare più velocemente i dati contenuti in una tabella. Si veda il seguente esempio in cui viene illustrato questo concetto. Si supponga di essere interessati alla lettura di un libro di giardinaggio per sapere come coltivare dei peperoni. Invece di leggere il libro intero dall'inizio fino al paragrafo in cui viene descritto il procedimento per la coltivazione dei peperoni, risulta molto più semplice consultare l'indice presente alla fine del libro, individuare le pagine contenenti le informazioni sui peperoni e passare direttamente alla lettura di quelle pagine. La consultazione dell'indice consente di risparmiare tempo ed è, senza alcun dubbio, il modo migliore per individuare le informazioni necessarie.

Lo stesso principio vale per il recupero dei dati presenti in una tabella di database. In assenza di un indice, per individuare le informazioni desiderate il sistema del database è costretto a leggere l'intera tabella (questo processo viene denominato "scansione della tabella"). Con un indice appropriato, invece, il sistema del database può prima consultarlo per trovare la posizione in cui poter recuperare i dati, quindi andare direttamente in quella posizione e ottenere i dati desiderati. In questo modo il recupero dei dati è certamente più veloce.

Per questo motivo è sempre consigliato creare un indice nelle tabelle. Un indice può interessare una o più colonne. La sintassi generale per creare un indice è:

```
CREATE INDEX "nome_di_indice" ON "nome_della_tabella" (nome_di_colonna);
```

Si supponga di disporre della seguente tabella:

Tabella ***Customer***

Nome di Colonna	Tipo di Dati
First_Name	char(50)
Last_Name	char(50)
Address	char(50)
City	char(50)
Country	char(25)

Birth_Date	datetime

e che si desideri creare un indice nella colonna Last_Name. Digitare:

```
CREATE INDEX IDX_CUSTOMER_LAST_NAME
ON Customer (Last_Name);
```

Se si desidera creare un indice su entrambe le colonne City e Country, digitare:

```
CREATE INDEX IDX_CUSTOMER_LOCATION
ON Customer (City, Country);
```

Non esiste una regola precisa relativa alla denominazione di un indice. Il metodo comunemente accettato è posizionare un prefisso, come “IDX_”, prima di un nome di indice per evitare di creare confusione con altri oggetti del database. Può anche risultare utile fornire informazioni relative alla tabella e alle colonne su cui viene utilizzato l’indice.

Si noti che la sintassi esatta per **CREATE INDEX** può variare in base ai database utilizzati. Per conoscere la sintassi precisa, è consigliato consultare il manuale di riferimento.

Una volta create una tabella, esistono diverse occasioni per cui potrebbe essere possibile voler modificare la struttura della tabella. I casi più tipici sono i seguenti:

- Aggiungere una colonna
- Rimuovere una colonna
- Modificare il nome di una colonna
- Modificare il tipo di dati di una colonna

Si consideri che quanto riportato sopra non è un elenco esaustivo. Esistono altre istanze in cui **ALTER TABLE** viene utilizzato per modificare la struttura di una tabella, come la modifica delle specifiche di una chiave primaria o l'aggiunta di un vincolo univoco a una colonna.

La sintassi del linguaggio SQL per **ALTER TABLE** è

```
ALTER TABLE "nome della tabella"
[alteri la specifica];
```

[alteri la specifica] dipende dal tipo di modifica che si desidera apportare. Per i casi citati precedentemente, le istruzioni [alteri la specifica] sono:

- Aggiungere una colonna: ADD "colonna 1" "tipo di dati per la colonna 1"
- Rimuovere una colonna: DROP "colonna 1"
- Modificare il nome di una colonna: CHANGE "vecchio nome di colonna" "nuovo nome di colonna" "tipo di dati per la nuova colonna"
- Modificare il tipo di dati di una colonna: MODIFY "colonna 1" "nuovo tipo di dati"

Di seguito alcuni esempi relativi a ciascun caso elencato sopra:

Tabella ***Customer***

Nome di Colonna	Tipo di Dati
First_Name	char(50)
Last_Name	char(50)
Address	char(50)
City	char(50)

Country	char(25)
Birth_Date	datetime

La prima operazione che si desidera realizzare è aggiungere a questa tabella una colonna denominata “Gender”. Per realizzare questa operazione, digitare:

ALTER TABLE Customer ADD Gender char(1);

Struttura della tabella risultante:

Tabella ***Customer***

Nome di Colonna	Tipo di Dati
First_Name	char(50)
Last_Name	char(50)
Address	char(50)
City	char(50)
Country	char(25)
Birth_Date	datetime
Gender	char(1)

Successivamente, si desidera modificare il nome di “Address” in “Addr”. Per realizzare questa operazione, digitare:

ALTER TABLE Customer CHANGE Address Addr char(50);

Struttura della tabella risultante:

Tabella ***Customer***

Nome di Colonna	Tipo di Dati
First_Name	char(50)
Last_Name	char(50)
Addr	char(50)
City	char(50)
Country	char(25)
Birth_Date	datetime
Gender	char(1)

Quindi, si desidera modificare il tipo di dati di “Addr” in char(30). Per realizzare questa operazione, digitare:

ALTER TABLE Customer MODIFY Addr char(30);

Struttura della tabella risultante:

Tabella ***Customer***

Nome di Colonna	Tipo di Dati
First_Name	char(50)
Last_Name	char(50)
Addr	char(30)
City	char(50)
Country	char(25)
Birth_Date	datetime

Gender	char(1)

Infine, si desidera eliminare la colonna “Gender”. Per realizzare questa operazione, digitare:

```
ALTER TABLE Customer DROP Gender;
```

Struttura della tabella risultante:

Tabella ***customer***

Nome di Colonna	Tipo di Dati
First_Name	char(50)
Last_Name	char(50)
Addr	char(30)
City	char(50)
Country	char(25)
Birth_Date	datetime

Per vari motivi, a volte, può essere necessario eliminare una tabella del database. In caso contrario, si potrebbero creare non pochi problemi rispetto alla manutenzione del DBA. Fortunatamente, il linguaggio SQL consente di realizzare questa operazione mediante il comando **DROP TABLE**. La sintassi per **DROP TABLE** è:

```
DROP TABLE "nome_della_tabella";
```

In questo modo, se si sceglie di eliminare la tabelle denominata customer, creata precedentemente nel corso del paragrafo **CREATE TABLE**, è sufficiente digitare:

```
DROP TABLE Customer;
```

Può essere necessario, a volte, eliminare tutti i dati presenti in una tabella. Come descritto in precedenza, è possibile realizzare questa operazione mediante il comando **DROP TABLE**. Diversamente, si vedrà adesso come eliminare i dati senza per questo eliminare l’intera tabella. Per realizzare questa operazione, viene utilizzato il comando **TRUNCATE TABLE**. La sintassi per **TRUNCATE TABLE** è:

```
TRUNCATE TABLE "nome della tabella";
```

In questo modo, se si sceglie di tagliare la tabella denominata customer, creata in SQL CREATE TABLE, è sufficiente digitare:

```
TRUNCATE TABLE Customer;
```

Nei paragrafi precedenti, è stato visto come recuperare le informazioni da una tabella. È arrivato il momento di sapere come vengono inserite queste righe di dati in una tabella. Nei prossimi due paragrafi vengono descritte le istruzioni **INSERT** e **UPDATE**.

Nel linguaggio SQL, esistono fondamentalmente due modi per l’inserimento dei dati in una tabella: Uno, contempla l’inserimento dei dati in una riga alla volta, l'altro, in più righe alla volta. Di seguito viene descritto come inserire i dati in una riga alla volta mediante il comando **INSERT**.

La sintassi per l’inserimento dei dati in una tabella una riga alla volta è la seguente:

```
INSERT INTO "nome_della_tabella" ("colonna_1", "colonna_2", ...)
VALUES ("valore_1", "valore_2", ...);
```

Si supponga di disporre di una tabella caratterizzata dalla seguente struttura:

Tabella ***Store_Information***

Nome di Colonna	Tipo di Dati
Store_Name	char(50)
Sales	float
Txn_Date	datetime

e che adesso si desideri inserire un'altra riga nella tabella in cui siano rappresentati i dati delle vendite realizzate a Los Angeles il 10 gennaio 1999. In quella data, questo negozio ha realizzato 900 € di vendite. Si utilizzerà, quindi, il seguente script SQL:

```
INSERT INTO Store_Information (Store_Name, Sales, Txn_Date)
VALUES ('Los Angeles', 900, '10-Jan-1999');
```

Con il secondo metodo di inserimento dei dati, **INSERT INTO**, è possibile inserire più righe in una tabella. Diversamente dall'esempio precedente, in cui veniva inserita una sola riga specificandone il valore per tutte le colonne, adesso, mediante l'istruzione **SELECT**, è possibile specificare i dati che si desiderano immettere nella tabella. Ciò significa che si stanno utilizzando le informazioni presenti in un'altra tabella. La sintassi è la seguente:

```
INSERT INTO "tabella 1" ("colonna 1", "colonna 2", ...)
SELECT "colonna 3", "colonna 4", ...
FROM "tabella 2";
```

Si noti che questa rappresenta la forma più semplice. L'intera istruzione può facilmente contenere le clausole **WHERE**, **GROUP BY**, e **HAVING** così come unioni di tabella e alias.

In questo modo, se, ad esempio, si desidera disporre di una tabella ***Store_Information***in cui sono raccolte tutte le informazioni relative alle vendite del 1998 e si è già a conoscenza che i dati di origine risiedono nella tabella ***Sales_Information***, è necessario digitare:

```
INSERT INTO Store_Information (Store_Name, Sales, Txn_Date)
SELECT Store_Name, Sales, Txn_Date
FROM Sales_Information
WHERE Year (Txn_Date) = 1998;
```

In questo caso, per estrarre le informazioni relative all'anno di una data è stata utilizzata la sintassi di SQL Server. In altri database relazionali sono necessarie sintassi diverse. Ad esempio, in Oracle, viene utilizzata la sintassi **TO_CHAR (Txn_Date, 'yyyy') = 1998**.

Una volta presenti i dati in una tabella, potrebbe essere necessario modificarli. Per realizzare tale operazione, è possibile utilizzare il comando **UPDATE**. La sintassi per questa istruzione è:

```
UPDATE "nome_della_tabella"
SET "colonna 1" = [nuovo valore]
WHERE "condizionale";
```

Ad esempio, si supponga di avere la seguente tabella:

Tabella ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999

Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

e che si noti che le vendite relative a Los Angeles il 01/08/1999 corrispondono effettivamente a 500 € anziché 300 €. In questo caso, è necessario aggiornare quella particolare voce. Per realizzare questa operazione si utilizza la seguente istruzione SQL:

```
UPDATE Store_Information
SET Sales = 500
WHERE Store_Name = "Los Angeles"
AND Txn_Date = "08-Jan-1999";
```

La tabella risultante è:

Tabella ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	500	08-Jan-1999
Boston	700	08-Jan-1999

In questo caso, solo una riga soddisfa la condizione relativa alla clausola **WHERE**. Se sono presenti più righe che soddisfano la condizione, verranno tutte modificate.

Mediante il commando **UPDATE**, è possibile anche aggiornare più colonne contemporaneamente. La sintassi necessaria per questa operazione è la seguente:

```
UPDATE "nome_della_tabella"
SET colonna 1 = [valore 1], colonna 2 = [valore 2]
WHERE "condizionale";
```

A volte, può essere necessario eliminare i record presenti in una tabella. Per realizzare tale operazione, è possibile utilizzare il comando **DELETE FROM**. La sintassi per questa istruzione è:

```
DELETE FROM "nome_della_tabella"
WHERE "condizionale";
```

L'utilizzo di un esempio ne faciliterà la comprensione. Si supponga di avere la seguente tabella:

Tabella ***Store_Information***

Store_Name	Sales	Txn_Date
Los Angeles	1500	05-Jan-1999
San Diego	250	07-Jan-1999
Los Angeles	300	08-Jan-1999
Boston	700	08-Jan-1999

e che si decida di non voler più conservare le informazioni relative a Los Angeles presenti in questa tabella. Per realizzare questa operazione, è necessario digitare la seguente istruzione SQL:

```
DELETE FROM Store_Information
WHERE Store_Name = "Los Angeles";
```

Adesso, il contenuto della tabella visualizzato è:

Tabella ***Store_Information***

Store_Name	Sales	Txn_Date
San Diego	250	07-Jan-1999
Boston	700	08-Jan-1999

E’ inoltre possibile con l’utilizzo del linguaggio SQL definire una serie di funzioni avanzate:

- SQL Union
- SQL Union ALL
- SQL Intersect
- SQL Minus
- SQL Sottoquery
- SQL EXISTS
- SQL CASE
- SQL NULL
- SQL ISNULL
- SQL IFNULL
- SQL NVL
- SQL COALESCE
- SQL NULLIF

La sintassi del linguaggio SQL per l’interrogazione di una base dati.

SQL SELECT

SQL DISTINCT

SQL WHERE

SQL AND OR

SQL IN

SQL BETWEEN

SQL LIKE

SQL ORDER BY

SQL Funzioni

SQL COUNT

SQL GROUP BY

SQL HAVING

SQL ALIAS

SQL Join

SQL Outer Join

SQL CONCATENATE

SQL SUBSTRING

SQL TRIM

SQL LENGTH

SQL REPLACE

SQL DATEADD

SQL DATEDIFF

SQL DATEPART

SQL GETDATE

SQL SYSDATE

SQL CREATE TABLE

SQL CONSTRAINT

SQL NOT NULL

SQL DEFAULT

SQL UNIQUE

SQL CHECK

SQL Chiave Primaria

SQL Chiave Esterna

SQL CREATE VIEW

SQL CREATE INDEX

SQL ALTER TABLE

SQL DROP TABLE

SQL TRUNCATE TABLE

SQL INSERT INTO

SQL UPDATE

SQL DELETE FROM

SQL Avanzato

SQL UNION

SQL UNION ALL

SQL INTERSECT

SQL MINUS

SQL Sottoquery

SQL EXISTS

SQL CASE

SQL NULL

SQL ISNULL

SQL IFNULL

SQL NVL

SQL COALESCE

SQL NULLIF

Tecniche avanzate per l’interrogazione di una base dati: ordinamento, raggruppamento, join tra tabelle.

Le join sono utilizzate per combinare righe di due o più tabelle, si basa sulle relazioni fra di loro, ad esempio la tabella ordini e la tabella clienti.

Ordini: OrdineId, ClientiId, dataOrdine,

Clienti: ClientiId, NominativoCliente, NomeContatto, Nazione

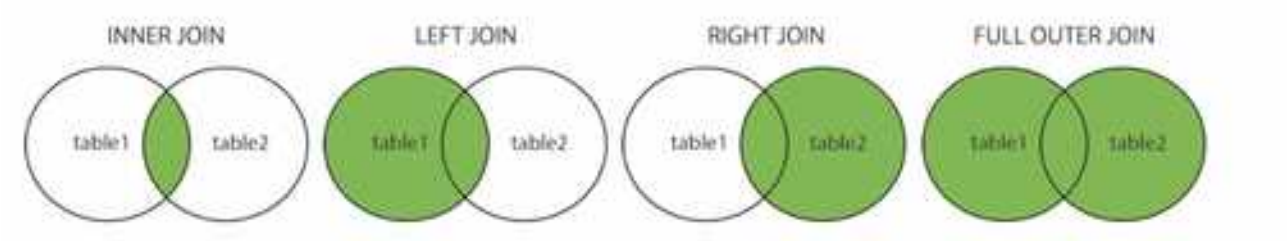
Possiamo effettuare un comando sql per ottenere i record della tabella clienti e ordini, in maniera tale da avere la lista degli ordini per cliente.

es.

```
SELECT Ordini.OrdineId, Clienti.NominativoCliente, Ordini.dataOrdine
FROM Ordini
INNER JOIN Clienti ON Ordini.ClienteId=Clienti.ClienteId;
```

Questa produce una lista di ordini dei vari clienti, come si può notare vengono recuperati tutti gli ordini con i dati dei clienti relativi, nel caso in cui un cliente non sia presente, quindi un ordine sia orfano la query non estrae il record relativo a tale ordine.

Vediamo nel dettaglio i tipi di join come in figura sottostante.



Tipi di Join:

1. Inner join: ritorna i record che hanno matching in entrambe le tabelle

2. Left join: ritorna i record della tabella di partenza che hanno i relativi record nella tabella di join

3. Right join: ritorna i record della tabella in join e i record che hanno matching sulla tabella di partenza.

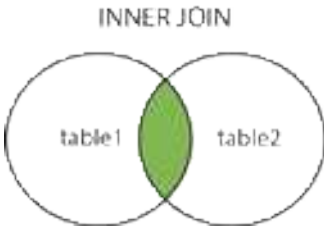
4. Full outer join: ritorna tutti i record che hanno match in entrambi i versi (LEFT + RIGHT)

Inner join.

L'INNER JOIN seleziona i records che hanno un matching in entrambe le tabelle.

Es. di sintassi

SELECT *column_name(s)*
FROM *table1*
INNER JOIN *table2* ON *table1.column_name = table2.column_name*;



Database di DEMO

In questo tutorial utilizziamo un database di demo con dei dati a campione.

Di seguito la tabella degli ordini:

OrdineId	ClientiId	impiegatiId	dataOrdine	consegnaId
10308	2	7	1996-09-18	3
10309	37	3	1996-09-19	1
10310	77	8	1996-09-20	2

Tabella dei clienti:

ClienteId	Nominativo Cliente	NomeContatto	Indirizzo	Citta	CodicePostale	Nazione
1	Paolino Paperino			Paperopoli		Immaginaria
2	Paperone			Paperopoli		Immaginaria
3	Topolino			Paperopoli		Immaginaria

Esempio di SQL inner join

The following SQL statement selects all orders with customer information:

Example

```
SELECT Orders.OrderID, Customers.CustomerName
FROM Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```

Note: The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns. If there are records in the "Orders" table that do not have matches in "Customers", these orders will not be shown!

Join con tre tabelle.

Il seguente join recupera tutti gli ordini con le informazioni dei clienti e delle spedizioni:

Esempio

```
SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName
FROM ((Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID)
INNER JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID);
```

L’indicizzazione del database.

Gli indici sono delle tabelle speciali associate alle tabelle dati e diventano utili nel momento in cui vengono eseguite delle query.

Infatti, quando un database crea un record in una tabella, tale record seguirà un ordine che è quello di inserimento. Questo equivale a dire che, in assenza di un indice, ogni operazione che tenta il recupero di dati da qualsiasi tabella del database, costringe il database stesso a leggere l’intera tabella, eseguendo quello che in gergo viene definito scansione della tabella (Table scan). Il famigerato Table Scan, infatti, di norma è sinonimo di crollo delle prestazioni.

Con un indice appropriato, invece, il database è capace di recuperare i dati necessari direttamente consultando l'indice, per identificare la posizione esatta occupata dalle informazioni sul database stesso.

In questo modo, si evita il Table Scan, il recupero dei dati su cui le query devono lavorare avviene in modo più veloce e la query stessa è più performante.

Messa in questi termini, si potrebbe pensare di associare un indice a qualsiasi campo di qualsiasi tabella dati contenuta nel database, credendo che in questo modo le performance tendano a migliorare.

In realtà, questa conclusione è errata e viene a essere confutata dall'esperienza d'uso. Infatti, quando con il linguaggio SQL si creano degli indici, SQL memorizza tanto i dati della tabella, quanto i dati degli indici. Inoltre, a differenza di altri linguaggi progettati per la gestione dei file, il linguaggio SQL permette di creare più indici sulla stessa tabella.

Entrambe queste caratteristiche delle strutture degli indici fanno sì che, qualora si modifichino spesso i dati a cui gli indici sono associati, a queste variazioni ne conseguano altrettante relative ai puntatori che puntano alle tuple della tabella stessa.

Questa situazione, insieme al fatto che gli indici occupano spazio su disco, porta a una brusca caduta delle performance dell'intero database, qualora gli indici siano troppi o siano associati a tipologie di query, di esecuzioni e dati in modo non opportuno.

Per questo, nell'uso degli indici, è opportuno tenere a mente tutta una serie di *tips* e *reminders*, utili a capire come gestire questa tipologia di struttura.

Prima di tutto, bisogna sapere che:

- gli indici sono indicati nelle query di SELECT, in quelle con condizioni WHERE o negli ordinamenti di tipo ORDER BY; sono da evitare nei comandi di INSERT e UPDATE per quanto detto fino a ora;
- gli indici velocizzano le query a livello computazionale, garantendo un accesso più veloce ai dati coinvolti nell'interrogazione, ma occupano molto spazio su disco;
- le query possono essere ottimizzate tramite l'uso degli indici solo se lavorano su una quantità di dati che non superi il 30 percento dei dati totali. Nel caso in cui si lavori con una quantità di dati superiore, allora gli indici non migliorano la velocità di lettura delle query;
- gli indici non dovrebbero comunque essere usati sulle tabelle piccole e con poche tuple, perché non migliorerebbero i tempi di accesso, ma produrrebbero l'effetto contrario;
- i migliori risultati nell'uso degli indici si ottengono quando questi lavorano su un numero consigliato di quattro o cinque colonne con importanti quantità di dati e con pochi valori NULL;
- gli indici non dovrebbero essere usati su dati che richiedono modifiche frequenti e di questa regola bisognerebbe tenerne conto, quando si effettuano molti aggiornamenti. Se si deve procedere con un aggiornamento totale dei dati e vi sono degli indici, le buone norme impongono che prima si proceda alla distruzione degli indici, poi si aggiornino i dati e infine si ricreino gli indici necessari;
- se si usano delle funzioni sugli attributi, allora è opportuno non indicizzarli;
- l'indice non si sfrutta se si usa l'operatore di diseguaglianza (!=);

- l’uso degli indici è sconsigliato nei confronti testuali con l’operatore LIKE e, comunque, in questi casi le wildcard vanno messe in fondo alla direttiva;
- nelle query su indici combinati, bisogna mantenere l’ordine per garantire migliori performance.

Quali tipologie di indici esistono

Nell’ultima raccomandazione, si è parlato di indici combinati. In realtà, con l’aggettivo combinato si indica una tipologia di indice, fra diversi disponibili.

In generale, si può dire che un indice può essere di tipo semplice o combinato, clustered o unclusterd, univoco o non univoco.

Senza scendere troppo nei dettagli, un indice viene detto semplice quando contiene una sola colonna, mentre viene definito composto quando è dichiarato su due o più colonne di dati.

Un indice cluster, invece, definisce l’ordinamento della tabella. In pratica, questo tipo di indice non esiste fisicamente, ma definisce le colonne (o attributi) rispetto ai quali ordinare, in modo sequenziale, i dati memorizzati nella tabella. Ovviamente, può essere definito un solo indice clustered per ogni tabella ed è il più performante dal punto di vista delle query di richiesta. Allo stesso modo, è anche il più esoso dal punto di vista delle risorse, qualora venga usato nelle query di aggiornamento e inserimento. Il più delle volte, l’indice clustered coincide con l’identificatore della tupla di dati, perché è immodificabile.

Un indice non clustered non memorizza i dati della tabella, ma solo i puntatori ai dati in una struttura a tabella a parte contenuta sempre nel database.

Infine, un indice può essere univoco qualora i dati a cui l’indice fa riferimento non possano essere duplicati all’interno della tabella, mentre, al contrario, quello non univoco permette di inserire nella tabella più tuple con gli stessi valori delle colonne definite come indici.

Come si implementano e si distruggono gli indici

Dal punto di vista applicativo, la creazione di un indice avviene in modo molto semplice, con un apposito comando del linguaggio SQL.

La sintassi di base prevede che si usi una query del tipo:

CREATE INDEX index_name ON table_name;

Per quanto riguarda il nome dell’indice, è prassi comune usare il prefisso idx_, prima del nome dell’indice, di solito formato dalle informazioni relative alla tabella e alla colonna su cui l’indice viene creato. In questo modo, si evita di confondere l’indice con altre strutture del database.

Per creare un indice su singola colonna, usare:

CREATE INDEX index_name ON table_name (column_name);

mentre per creare un indice composito, bisogna usare:

CREATE INDEX index_name ON table_name (column1, column2);

Qualora sia necessario eliminare un indice creato in precedenza, bisogna usare:

DROP INDEX index_name;

Analisi e miglioramento delle performance delle interrogazioni SQL.

Guida di riferimento (https://docs.microsoft.com/it-it/sql/relational-databases/performance/start-and-use-the-database-engine-tuning-advisor?view=sql-server-2017)

Introduzione alle transazioni.

Le singole istruzioni `INSERT`, `UPDATE` e `DELETE` sono atomiche nel senso che o hanno successo totale (su tutte le righe coinvolte) o falliscono totalmente senza alcun effetto sul database.

È possibile combinare più istruzioni in una singola **transazione atomica**.

```
START TRANSACTION;
  DELETE FROM Book WHERE Publisher = 1;
  DELETE FROM Publisher WHERE ID = 1
COMMIT;
```

Con questo script viene eliminato l'editore con ID pari a 1, ma prima vengono eliminati i libri pubblicati da esso. Racchiudendo il blocco tra `START TRANSACTION` e `COMMIT TRANSACTION`, si fa in modo di rendere tutto il blocco atomico: o avvengono con successo entrambe le istruzioni oppure in caso di errore tutto resta invariato. Con l'istruzione `ROLLBACK`, invece, si forza il fallimento della transazione, lasciando il database allo stato consistente.

Livello di isolamento delle transazioni

A proposito delle transazioni, soprattutto se esse sono lunghe, è importante sapere quale **livello di isolamento** stiamo usando. Il livello di isolamento stabilisce come transazioni contemporanee si comportano rispetto ai dati. Ogni RDBMS ha un livello di isolamento di default e generalmente si può stabilire un livello differente per sessione o per transazione. Lo standard ANSI/ISO stabilisce quattro livelli di isolamento, generalmente implementati da tutti i RDBMS:

1. **Serializable**. È il massimo livello di isolamento: ogni transazione, dall'inizio alla fine, non vede le modifiche fatte ai dati acceduti. Il vantaggio è che la transazione può lavorare sul database assumendo di essere la sola transazione in corso sul database. Un modo di realizzare questo livello è l'approccio ottimistico: ogni transazione lavora in isolamento, poi, se accadono problemi di concorrenza, la transazione che tenta di agire su un dato modificato da altre transazioni fallirà con un errore e con conseguente rollback. Lo svantaggio è che ci possono essere molte scritture fallite se ci sono tante transazioni che interessano gli stessi dati.
2. **Repeatable Read**. Con questo livello si fa in modo che i dati letti durante la transazione in corso non possono essere modificati da altre transazioni per tutta la durata della transazione in corso. I vari RDBMS gestiscono questo livello utilizzando i **lock in lettura** sulle righe

lette durante la transazione. L'unico problema che può succedere con questo livello consiste nel verificarsi delle cosiddette **letture fantasma**: se rieseguo la stessa query durante la transazione, potrei trovare righe in più di quelle che ho letto in precedenza, ma mai in meno o modificate. Lo svantaggio di questo livello è una penalizzazione delle prestazioni se ci sono molte transazioni concorrenti che agiscono sulle stesse tabelle.

3. **Read Committed**. Utilizzando questo livello, invece, si evitano i lock in lettura sulle tabelle che sono molto onerosi dal punto di vista prestazionale. Lo svantaggio è che, oltre al fenomeno delle letture fantasma, si verifica anche quello delle **letture non ripetibili**: in pratica, rieseguendo due volte la stessa `SELECT` nel corso di una transazione, potrei ottenere dati diversi se altre transazioni sono terminate nel tempo intercorso tra le due letture. Questo è il livello di default per Oracle e per Microsoft Sql Server.
4. **Read Uncommitted**. Questo è il livello più basso, in pratica nessun isolamento. Con questo livello si possono avere **letture sporche**: nella transazione corrente si possono leggere dati che qualche altra transazione sta scrivendo in quel momento senza aver ancora fatto `COMMIT`, quindi può capitare di leggere chiavi violate, dati inconsistenti, eccetera.

Ovviamente non esiste il livello migliore di isolamento, generalmente il livello di default è valido nella maggior parte dei contesti. Gli RDBMS supportano gli altri livelli per gestire casi particolari di utilizzo in concorrenza.

Introduzione all'SQL procedurale (PL/SQL, TSQL).