Il linguaggio SQL

Unità didattica

A4

Prerequisiti

- Terminologia e concetti introduttivi sulle basi di dati.
- La metodologia per la progettazione di basi di dati non a oggetti.
- I modelli concettuali.
- Il modello ER.
- Il modello relazionale.

Obiettivi

- Creare lo schema di una base di dati, utilizzando istruzioni della parte DDL di SQL.
- Trattare i dati contenuti nelle tabelle, utilizzando le istruzioni della parte DML di SQL.
- Assicurare l'integrità dei dati tramite vincoli, viste e diritti di accesso, utilizzando istruzioni della parte DCL di SQL.
- Interfacciare SQL con un linguaggio di programmazione.

Conoscenze da apprendere

- Conoscere i tipi di dato supportati da SQL.
- Conoscere le istruzioni di base per creare tabelle a partire da uno schema relazionale.
- Conoscere la sintassi dell'istruzione SELECT per effettuare interrogazioni alla base di dati.
- Conoscere le funzioni di aggregazione, ordinamento e raggruppamento.

Competenze da acquisire

- Saper tradurre in SQL le operazioni dell'algebra relazionale.
- Saper impostare i vincoli interni ed esterni in una tabella.
- Saper impostare i vincoli tra più tabelle.
- Saper effettuare interrogazioni complesse, componendo interrogazioni più semplici.

1 Un linguaggio per le basi di dati relazionali

Il **linguaggio SQL** (*Structured Query Language*) è un linguaggio non procedurale o di tipo dichiarativo. È ormai da tempo uno degli standard tra i linguaggi per basi di dati relazionali.

Il linguaggio SQL è presente in diverse *versioni* o *dialetti*, che sono in genere aderenti agli standard internazionali **ANSI** (*X3,135*) e successivamente **ISO** (*9075*).

Le differenze tra tali versioni possono essere facilmente individuate attraverso la documentazione elettronica o cartacea fornita dai prodotti software che le implementano. Tutte si rifanno alla versione dello standard adottato nel 1992, detto **SQL2** o **SQL92**.

Una versione importante è l'**SQL3** (che è un'estensione dell'SQL2) che implementa nuove caratteristiche come la *ricorsione* e funzionalità per il *trattamento* degli oggetti.

In questa unità troveremo solo le linee guida per SQL2 (che da ora in poi chiameremo SQL). Per ulteriori dettagli e approfondimenti rimandiamo alla documentazione ufficiale.

Funzioni assolte dal linguaggio SQL

Il linguaggio SQL assolve alle **funzioni** di:

- **DDL** (*Data Definition Language*), che prevede le istruzioni per definire la struttura delle relazioni della base di dati. Serve quindi a creare tabelle, vincoli, viste e indici;
- **DML** (*Data Manipulation Language*), che prevede le istruzioni per manipolare i dati contenuti nelle diverse tabelle. In particolare permette inserimenti, cancellazioni e modifiche delle tuple;
- **DCL** (*Data Control Language*), che prevede istruzioni per controllare il modo in cui le operazioni vengono eseguite. Consente di gestire il controllo degli accessi per più utenti e i permessi per gli utenti autorizzati.

Modalità stand-alone/embedded

SQL può essere usato in:

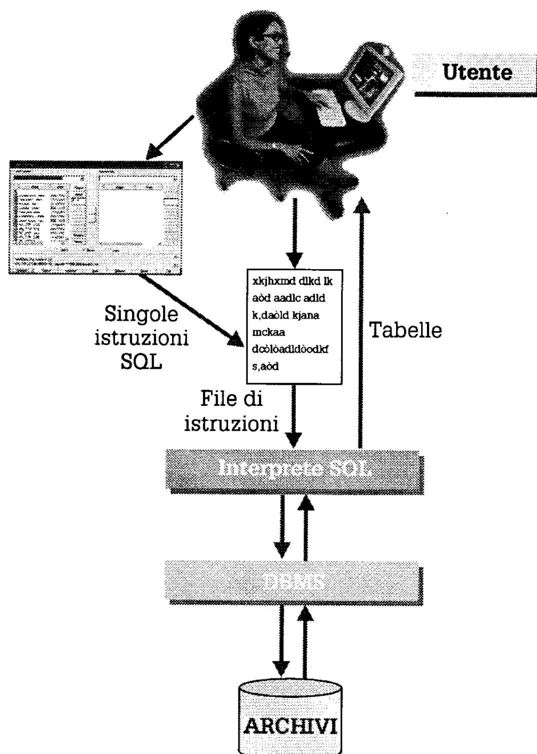
- modalità *stand-alone* o *a sé stante*;
- modalità *embedded* o *linguaggio ospite*.

1.1 SQL stand-alone

Modo interattivo e batch

Il linguaggio **SQL** nella modalità **stand-alone** può essere classificato come **query language interattivo**. I comandi possono essere inviati al sistema operativo in modo **interattivo** o **batch** (cioè da eseguire in gruppo). In modo *interattivo* si utilizza un'interfaccia grafica, che ricorre a menu, finestre e icone per guidare l'utente nella scelta dell'operazione da effettuare sui dati. In modo *batch* è possibile creare file di istruzioni pronte per essere eseguite. In entrambe le modalità per eseguire ogni istruzione viene invocato l'**interprete** SQL.

Le interrogazioni complesse vengono composte sfruttando le operazioni dell'algebra relazionale e ottenendo come risultato sempre una tabella rappresentante una relazione. La situazione è riassunta nella *Fig. A4.1*.



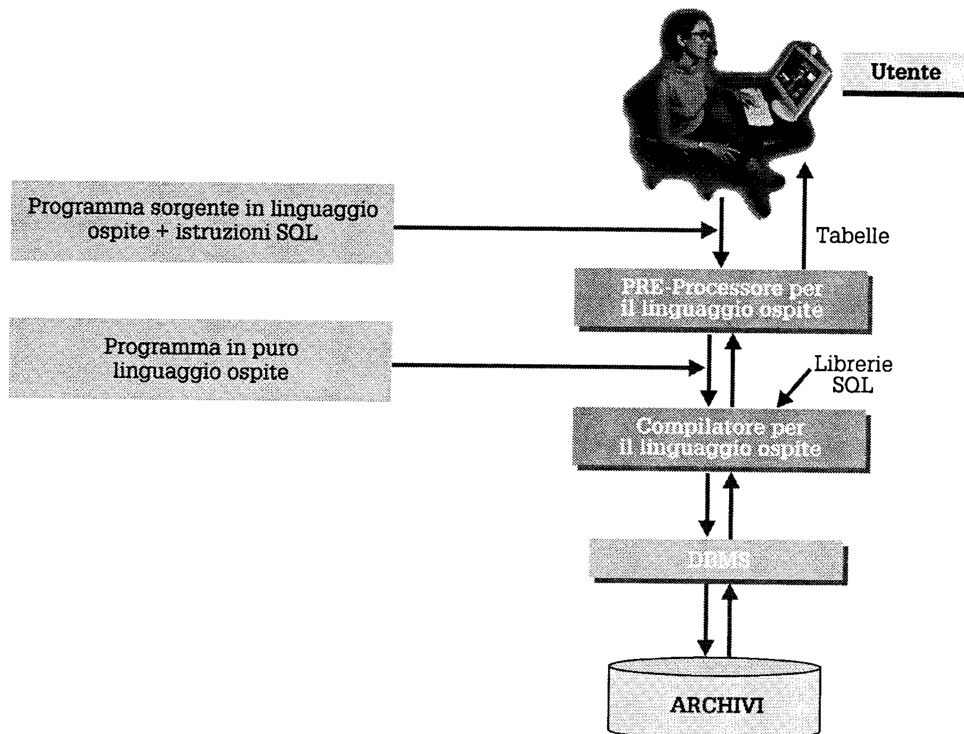
✦ Figura A4.1
Il linguaggio SQL in modalità stand-alone

1.2 SQL embedded

In questo caso, dato che il linguaggio ospite può trattare le tuple anche singolarmente, è necessario introdurre altri concetti tipici dei linguaggi procedurali o a oggetti. Ad esempio è possibile utilizzare comandi SQL all'interno di istruzioni **C**, **Java** o **C++**, che vengono chiamati, in questo caso, **linguaggi "ospite"**.

Per la modalità embedded i chiarimenti per l'interazione tra SQL e linguaggio ospite sono rimandati ai manuali di riferimento dei singoli linguaggi di programmazione. In linea di massima, possiamo dire che un programma scritto in un *linguaggio ospite compilato* (che

incorpora istruzioni SQL al suo interno) subirà un primo processo di **precompilazione**, seguito da una vera e propria compilazione, come mostrato nella figura seguente.



✦ Figura A4.2
Il linguaggio SQL in modalità embedded

In questa unità faremo riferimento alla versione stand-alone di SQL. In particolare, parleremo di **client SQL** per indicare l'interfaccia grafica che ci permette di interagire, tramite istruzioni (dette anche *comandi SQL in modalità stand-alone*), con la nostra base di dati.

Come già detto, rispetto ai linguaggi di programmazione procedurale, il linguaggio SQL è un **linguaggio dichiarativo**, nel senso che le istruzioni si limitano a descrivere "cosa" si vuol fare. Al "come" farlo ci pensa il sistema. Con i linguaggi dichiarativi occorre familiarizzare, ma sicuramente è un modo di programmare più semplice, intuitivo e conciso rispetto ai linguaggi non dichiarativi.

2 Identificatori e tipi di dati

Le istruzioni possono essere scritte *utilizzando indifferentemente* caratteri maiuscoli o minuscoli. Generalmente vengono separate dal ";" ma ciò non accade per tutte le versioni. In questa unità utilizzeremo caratteri maiuscoli per le parole chiave.

Identificatori

Gli **identificatori** utilizzati per i nomi delle tabelle e degli attributi devono:

- avere *lunghezza massima* pari a 18 caratteri;
- *iniziare* con una lettera;
- *contenere* come unico carattere speciale l'underscore: "_".

Terminologia SQL

Nella terminologia SQL le *relazioni* sono chiamate **tabelle**, le *tuple* **righe** o **registrazioni** e gli *attributi* sono le **colonne** delle tabelle.

Per riferirsi a un attributo di una tabella si utilizza la seguente dot-notation:

<NomeTabella>.<NomeAttributo>

Tipi di dato

I **tipi di dato** utilizzabili per gli attributi sono riassunti nella tabella seguente (anche se occorre precisare che in alcune versioni tali tipi potrebbero essere differenti).

TIPO	DESCRIZIONE	RANGE DI VARIABILITÀ
Character(n) abbreviato a *char(n)*	Stringa lunga n caratteri *char*(1) corrisponde a *char* o *character*	*n* varia da 1 a 15000
Integer(p) abbreviato a *int(p)*	Numero intero con precisione *p* (numero massimo di cifre che il numero può contenere) *p* varia da 1 a 45	dipende dalla precisione
Integer	Numero intero con precisione 10	da – 2147483648 a + 2147483647
Smallint	Numero intero con precisione 5	da – 32768 a + 32767
Decimal(p, d) Abbreviato a *dec(p, d)*	Numero reale in fixed point con precisione *p* (numero massimo di cifre prima del punto decimale) e *d* cifre decimali *p* varia da 1 a 45 *d* varia da 0 a *p*	dipende da *p* e *d*
Float(p)	Numero reale in floating point con precisione *p* per la mantissa *p* varia da 1 a 45	dipende da *p*
Float	Numero reale in floating point con precisione 15 per la mantissa	da 1E–38 a 1E+38
Real	Numero reale in floating point con precisione 7 per la mantissa	da 1E–38 a 1E–38
Date	Data nel formato AA/MM/GG	
Time	Ora nel formato HH:MM	

Costanti stringa

Operatori

Le **costanti stringa** sono rappresentabili utilizzando indifferentemente gli apici (‘ ’) o i doppi apici (“ ”). Nelle **espressioni** possono essere usati i seguenti **operatori**:

- aritmetici (+, –, /, *);
- relazionali (<, >, ≤, ≥);
- logici (**AND**, **OR**, **NOT**).

I confronti tra dati numerici sono effettuati in accordo al loro *valore algebrico*.
I confronti tra dati alfanumerici sono effettuati in accordo al valore del corrispondente *codice ASCII* dei caratteri che li compongono, cominciando dal carattere più a sinistra (se si tratta di stringhe di più caratteri).

3 Istruzioni del DDL di SQL

DDL

Le istruzioni del **DDL** di SQL creano o modificano lo schema di una relazione. Vediamo i più importanti.

3.1 Creare una tabella e i vincoli di integrità

Vediamo come si può creare una tabella. Utilizzeremo la sintassi:

```
CREATE TABLE <NomeTabella>
        (<Attributo1> <Tipo1> [NOT NULL],
         <Attributo2> <Tipo2> [NOT NULL],
              ...,
         <AttributoN> <TipoN> [NOT NULL],
        [<Vincolo>]);
```

dove:

- NOT NULL (opzionale) sta a significare che il corrispondente attributo non può mai assumere il valore NULL (corrisponde a dichiarare l'attributo come obbligatorio);
- <Vincolo> esprime i possibili vincoli interni esistenti. Tali vincoli possono essere:
  - **PRIMARY KEY**(<ElencoColonne>), che indica le colonne facenti parte della chiave primaria, specificando un *vincolo di chiave primaria* per il modello relazionale;
  - **UNIQUE**(<ElencoColonne>), che indica le colonne facenti parte di una chiave candidata;
  - **FOREIGN KEY**(<ElencoColonne1>) **REFERENCES** <NomeTabella> (<ElencoColonne2>);
  - **<ELENCOCOLONNE1>** indica le colonne facenti parte di una chiave esterna che devono essere la chiave primaria <ElencoColonne2> di <NomeTabellaReferenziata>; si specifica così un *vincolo di integrità referenziale*;
  - **CHECK**(<Condizione>), che indica un vincolo interno alla tabella. Specifica un *vincolo intrarelazionale*.

NOT NULL

Vincoli interni

Esempio

Consideriamo le seguenti relazioni:

Azienda(CodAzienda, RagioneSociale, CodAttività, CodDip)
Dipendenti(CodDip, Cognome, Nome, DataAssunzione, Livello, StipendioLordo, Trattenute, StipendioNetto)
Categoria(CodCategoria, Nome)

Per creare le relative tabelle in SQL scriveremo:

```
CREATE TABLE Dipendenti
    (CodDip             char(6) NOT NULL,
    Cognome             char(30) NOT NULL,
    Nome                char(30) NOT NULL,
    DataAssunzione      Date,
    Livello             char(1)     NOT NULL,
    StipendioLordo      Decimal(8,3) NOT NULL,
    Trattenute          Decimal(8,3) NOT NULL,
    StipendioNetto      Decimal(8,3) NOT NULL,
        PRIMARY KEY (CodDip),
        CHECK (StipendioLordo > 0),
        CHECK (StipendioNetto > 0),
        CHECK (StipendioLordo – StipendioNetto = Trattenute),
        CHECK (Livello BETWEEN '0' AND '9'),
    );
CREATE TABLE Categoria
    (CodCategoria       char(4) NOT NULL,
    Nome                char(11),
        PRIMARY KEY (CodCategoria),
        CHECK (Nome = ANY('produzione', 'servizi', 'finanziamento', 'formazione', 'turismo')),
    );
CREATE TABLE Azienda
    (CodAzienda         char(5)  NOT NULL,
    RagioneSociale      char(30),
    CodAttivita         char(4),
    CodDip              char(6)  NOT NULL,
        PRIMARY KEY (CodAzienda),
        FOREIGN KEY (CodDip) REFERENCES Dipendenti,
        FOREIGN KEY (CodAttivita) REFERENCES Categoria (CodCategoria),
    );
```

L'attributo *CodDip* è la chiave primaria della tabella.

3.2 Creare un dominio

In SQL è possibile definire nuovi tipi di dato con l'istruzione **CREATE DOMAIN**, la cui sintassi è:

CREATE DOMAIN <NomeDominio> [**AS**] <Tipo> [**CHECK**(<Condizione>)]

dove si assegnano a <NomeDominio> tutti i valori di <Tipo> che verificano la <Condizione> (se presente, poiché è opzionale).

Consideriamo l'esempio del paragrafo precedente. Potremo indicare, per nostra comodità di scrittura, nelle definizioni future di altre tabelle i seguenti domini:

```
CREATE DOMAIN TipoCategoria AS char(10)
CHECK (VALUE IN ('produzione', 'servizi', 'finanziario', 'formazione', 'turismo'));

CREATE DOMAIN TipoLivello AS char(1)
CHECK (VALUE BETWEEN '0' AND '9');

CREATE DOMAIN Positivo AS Integer
CHECK (VALUE > 0);
```

Potremo così utilizzarli nella definizione delle tabelle precedenti, come, ad esempio, nella tabella *Categoria*:

```
CREATE TABLE Categoria
  (CodCategoria        char(4) NOT NULL,
  Nome                 TipoCategoria,
     PRIMARY KEY (CodCategoria),
  );
```

DROP DOMAIN e ALTER DOMAIN

Esistono anche istruzioni per:

- eliminare un dominio (**DROP DOMAIN**);
- modificare un dominio (**ALTER DOMAIN**).

3.3 Modificare la struttura di una tabella

ALTER TABLE

Una volta creata la struttura di una tabella, la si può successivamente modificare con l'istruzione ALTER TABLE, la cui sintassi è:

ALTER TABLE <NomeTabella>
ADD <NomeColonna1> <Tipo> | <NomeDominio>
[**BEFORE** <NomeColonna2>];

oppure:

ALTER TABLE <NomeTabella>
DROP COLUMN <NomeColonna3>;

oppure:

ALTER TABLE <NomeTabella>
MODIFY (<NomeColonna4> <NuovoTipoColonna>);

dove:

ADD, DROP e MODIFY

- l'istruzione **ADD** è utilizzata per aggiungere la nuova colonna <NomeColonna1> di <Tipo> o <Dominio> specificato, alla tabella <NomeTabella>. Eventualmente, si specifica la posizione in cui inserirla, ad esempio prima (**BEFORE**) della colonna <NomeColonna2>;
- l'istruzione **DROP COLUMN** viene utilizzata per eliminare la colonna <NomeColonna3> dalla tabella <NomeTabella>;
- l'istruzione **MODIFY** viene utilizzata per modificare solo il tipo di una colonna e non il suo nome.

Ad esempio, sempre riferendoci all'esempio del paragrafo precedente, possiamo scrivere:

```
ALTER TABLE Dipendenti
   ADD DataNascita date;

ALTER TABLE Dipendenti
   DROP COLUMN DataAssunzione;
```

3.4 Eliminare una tabella

Per cancellare completamente una tabella dalla base di dati si utilizza l'istruzione **DROP**, la cui sintassi è:

DROP

DROP TABLE<NomeTabella> **[RESTRICT | CASCADE | SET NULL]**;

La cancellazione di una tabella può provocare inconsistenze dovute al fatto che possono esistere tabelle collegate tramite chiavi esterne o, più in generale, tramite vincoli di integrità. Per far fronte a tali situazioni si utilizzano:

RESTRICT, CASCADE e SET NULL

- l'istruzione **RESTRICT**, che non permette la cancellazione se la tabella da cancellare è legata tramite vincoli ad altre tabelle;
- l'istruzione **CASCADE**, che dà luogo a una cancellazione ricorsiva in cascata su tutte le tabelle collegate;
- l'istruzione **SET NULL**, che pone a *Null* tutti i valori delle chiavi interessate.

Ad esempio, per cancellare la tabella *Categoria* scriveremo:

```
DROP TABLE Categoria;
```

Se scrivessimo:

```
DROP TABLE Categoria RESTRICT;
```

l'operazione non verrebbe consentita se ci fossero chiavi in *CodAttività* di *Azienda* che si riferiscono a tuple di *Categoria*. Scrivendo invece:

```
DROP TABLE Categoria CASCADE;
```

verrebbe cancellata la tabella *Categoria* e, con essa, tutti i riferimenti alle chiavi di *Categoria* presenti nell'attributo *CodAttività* di *Azienda*.

3.5 I vincoli di integrità e le politiche di violazione

Possiamo riassumere quanto abbiamo a disposizione finora per creare e gestire i vincoli in SQL. Per esprimere i **vincoli interni** o **intrarelazionali**, utilizziamo durante la creazione di una tabella:

Vincoli interni o intrarelazionali

- l'istruzione **NOT NULL**, per esprimere vincoli sul valore di un *attributo*;
- l'istruzione **PRIMARY KEY**, per esprimere vincoli sulle *chiavi*;
- l'istruzione **CHECK**, per esprimere vincoli di *ennupla* che coinvolgono diversi attributi;
- l'istruzione **VALUE BETWEEN** e l'istruzione **VALUE IN**, per esprimere vincoli di *dominio*.

Per esprimere invece **vincoli di integrità referenziale**, utilizziamo durante la creazione di una tabella:

Vincoli di integrità referenziale

- l'istruzione **FOREIGN KEY ... REFERENCES**, per indicare le chiavi esterne;
- l'istruzione **RESTRICT**, **CASCADE** e **SET NULL** per implementare una delle tre politiche:
  - *rifiuto* delle modifiche che violano un vincolo;
  - *propagazione* a cascata della modifica;
  - *impostazione* a *Null* delle chiavi delle tabelle interessate.

Violazione di un vincolo

RESTRICT, CASCADE e SET NULL sono clausole molto importanti perché chiariscono cosa fare in caso di **violazione di un vincolo**.

La **politica di default** è la RESTRICT, cioè rifiutare le modifiche che violano un vincolo. Tale politica può essere modificata utilizzando CASCADE o SET NULL.

ON DELETE e ON UPDATE

Poiché tali politiche devono essere utilizzate al momento di una cancellazione o di una modifica di un attributo interessato da un vincolo referenziale, è possibile impostarle direttamente al momento della creazione della tabella, utilizzando le clausole **ON DELETE** e **ON UPDATE**.

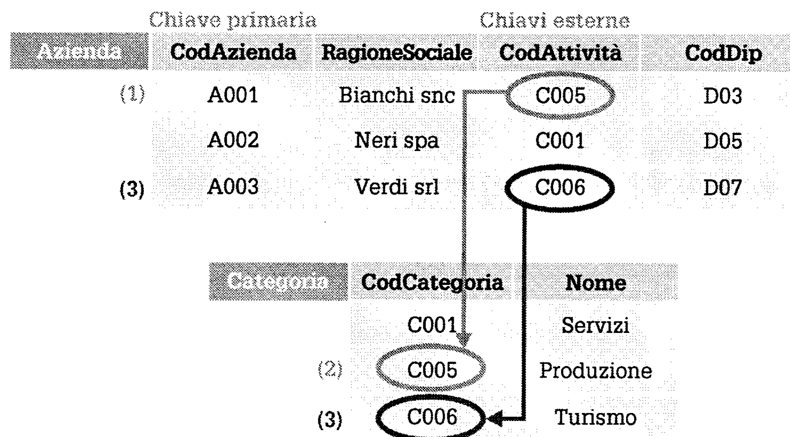
Ad esempio, per impostare la politica SET NULL per la cancellazione e la politica CASCADE per la modifica di un attributo di una tupla della tabella *Azienda*, scriveremo:

```
CREATE TABLE Azienda
   (CodAzienda        char(5) NOT NULL,
   RagioneSociale     char(30),
   CodAttività        char(4),
   CodDip             char(6) NOT NULL,
      PRIMARY KEY (CodAzienda),
      FOREIGN KEY (CodDip)         REFERENCES Dipendenti (CodDip),
      FOREIGN KEY (CodAttività)    REFERENCES Categoria (CodCategoria)
         ON DELETE SET NULL,
         ON UPDATE CASCADE
   );
```

In base a quanto dichiarato, se viene cancellato un valore dell'attributo *CodCategoria* di una tupla della tabella *Categoria*, automaticamente il sistema assegnerà il valore NULL ai corrispondenti valori dell'attributo *CodAttività* delle tuple della tabella *Azienda*.

Se invece viene modificato un valore dell'attributo *CodCategoria* di una tupla della tabella *Categoria*, verranno aggiornati con tale nuovo valore anche i corrispondenti valori dell'attributo *CodAttività* delle tuple della tabella *Azienda*.

Vediamo un esempio concreto di attuazione di queste politiche a seguito della violazione di un vincolo di integrità referenziale. La situazione prima di una cancellazione e di una modifica è illustrata nel seguente schema:

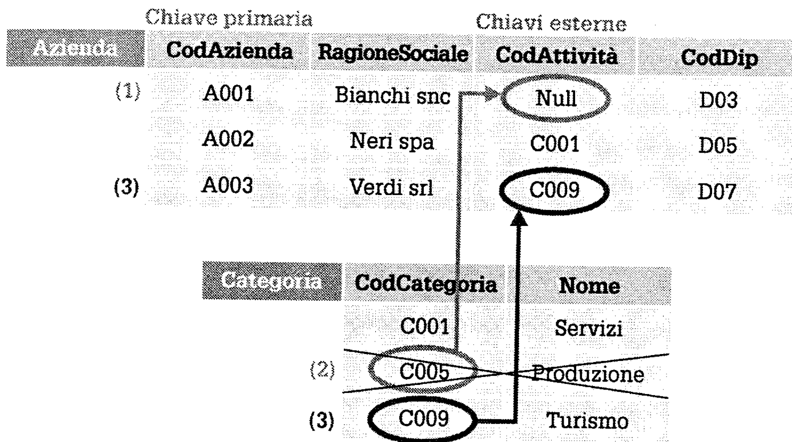


✦ Figura A4.3 Rappresentazione di due vincoli referenziali

Supponiamo ora che:

- la tupla (2) di *Categoria* contenente il valore "C005" dell'attributo *CodCategoria* venga cancellata. Nella tupla (1) di *Azienda*, in questo caso, verrà inserito il valore **Null**;
- la tupla (3) di *Categoria* contenente il valore "C006" dell'attributo *CodCategoria* venga modificata nel nuovo valore "C009"; in tal caso, nella tupla (3) di *Azienda* verrà inserito questo nuovo valore.

La nuova situazione sarà la seguente:



✦ Figura A4.4
La situazione dopo la violazione di due vincoli referenziali

4 Istruzioni DML di SQL

Una volta creato lo schema relazionale tramite le istruzioni del paragrafo precedente, occorre poter riempire, modificare e cancellare i valori delle righe delle tabelle. Vediamo le più importanti istruzioni della parte **DML** di SQL che effettuano tali operazioni.

DML

4.1 Inserire i valori in una tabella

I valori delle righe possono essere inseriti utilizzando l'istruzione **INSERT INTO**, la cui sintassi è:

INSERT

```
INSERT INTO <NomeTabella>[(<Attributo1>, <Attributo2>, ..., <AttributoN>)]
VALUES (<Valore1>, <Valore2>, ..., <ValoreN>);
```

Se non è presente la lista degli attributi, si intende che i valori specificati devono corrispondere in ordine, tipo e numero a quelli specificati nella dichiarazione di <NomeTabella>.

Se invece si specifica una lista di attributi di <NomeTabella>, l'ordine e il tipo dei valori dovrà rispettare questa lista. Gli attributi omessi vengono assunti come NULL.

Esempio

Inizializziamo la tabella *Categoria* inserendo la seguente riga:

```
INSERT INTO Categoria
VALUES ('C001','Servizi');
```

4.2 Modificare i valori delle righe di una tabella

Per aggiornare una o più righe di una tabella utilizziamo l'istruzione **UPDATE**, la cui sintassi è:

UPDATE

```
UPDATE <NomeTabella>
SET    <Attributo1> = <Espressione1>,
       <Attributo2> = <Espressione2>,
       ...,
       <AttributoN> = <EspressioneN>
[WHERE <Condizione>];
```

dove gli attributi di <NomeTabella> (specificati nella clausola SET) vengono aggiornati con i valori delle corrispondenti espressioni in tutte le righe che soddisfano la <Condizione>.

Esempio

Per cambiare la ragione sociale dell'azienda A001, scriveremo:

```
UPDATE Azienda
SET RagioneSociale = 'NuovaElettronica 3000'
WHERE CodAzienda = 'A001';
```

In un altro caso, per incrementare di 100 euro il valore dello stipendio dei dipendenti, scriveremo:

```
UPDATE Dipendente
SET StipendioLordo = StipendioLordo + 100.0;
```

4.3 Cancellare le righe di una tabella

DELETE

Per cancellare una o più righe di una tabella utilizziamo l'istruzione **DELETE**, la cui sintassi è:

```
DELETE FROM <NomeTabella>
[WHERE <Condizione>];
```

In questo modo si eliminano tutte le righe di <NomeTabella> che soddisfano la <Condizione>.

Esempio

Per cancellare i dipendenti assunti prima del 31 dicembre 1990, scriveremo:

```
DELETE FROM Dipendente
WHERE Data <= '901231';
```

Da notare che la data è stata espressa nella forma "anno-mese-giorno".

4.4 Istruzioni per il reperimento dei dati: SELECT

SELECT

Per il reperimento dei dati, il linguaggio SQL prevede il comando **SELECT**, la cui potenza ed espressività sono alla base del successo dell'SQL. Il risultato è sempre una tabella, che viene normalmente rappresentata a video, ma che può anche essere stampata, oppure assegnata a una variabile strutturata.

La sintassi del comando SELECT è molto complessa. Vediamo in questo paragrafo una sua forma semplificata:

```
SELECT [DISTINCT] <Attributo1>, ..., <AttributoN>
FROM <Tabella1>, <Tabella2>, ..., <TabellaK>
[WHERE <Condizione>];
```

SELECT restituisce una tabella formata dagli attributi:

<Attributo1>, <Attributo2>, ..., <AttributoN>

del prodotto delle tabelle:

<Tabella1>, <Tabella2>, ..., <TabellaK>

ristretto alle righe che soddisfano la <Condizione>.

Precisiamo che:

- se è assente la clausola **WHERE**, la condizione si assume sempre vera;
- se è presente l'opzione **DISTINCT**, il risultato viene fornito privo di righe duplicate.

WHERE e DISTINCT

Se si vogliono visualizzare tutti gli attributi presenti nel prodotto delle tabelle, è possibile utilizzare il simbolo "*" il cui significato sarà "tutti gli attributi del prodotto delle tabelle". La condizione inoltre può essere composta da più condizioni semplici combinate con gli operatori logici **AND**, **NOT**, **OR**. Vediamo alcuni esempi.

Esempio

Per visualizzare il cognome e il nome di tutti i dipendenti scriveremo:

```
SELECT Cognome, Nome FROM Dipendente;
```

Per visualizzare tutti gli attributi dei dipendenti scriveremo:

```
SELECT * FROM Dipendente;
```

Per visualizzare il cognome e il nome dei dipendenti che guadagnano più di 2000 euro scriveremo:

```
SELECT Cognome, Nome FROM Dipendenti WHERE StipendioNetto > 2000.00;
```

4.4.1 Intestare le colonne della tabella risultato

La tabella risultato di una SELECT ha (per default) come intestazione delle colonne il nome degli attributi della tabella. Se si vuole dare un diverso nome a ogni colonna del risultato, si deve utilizzare la clausola **AS** chiamata **alias**.

AS

Esempio

Per visualizzare il nome, il cognome e lo stipendio netto dei dipendenti, quest'ultimo con intestazione di colonna "*AttualeStipendio*", scriveremo:

```
SELECT Cognome, Nome, StipendioNetto AS AttualeStipendio FROM Dipendente;
```

4.4.2 Eseguire calcoli senza modificare il contenuto delle tabelle

È possibile far eseguire all'istruzione SELECT il calcolo di un'espressione sugli attributi. Il risultato viene visualizzato a video in una nuova colonna intestata con la clausola AS. Il calcolo viene eseguito esternamente alla tabella non vengono modificati quindi i dati in essa contenuti.

Esempio

Se vogliamo visualizzare una variazione del 10% degli stipendi dei dipendenti scriveremo:

```
SELECT Cognome, Nome, StipendioNetto * 1,10 AS NuovoStipendio FROM Dipendente;
```

4.5 Le operazioni relazionali in SQL

Le operazioni di *restrizione* (RESTRICT), *proiezione* (PROJECT) e *giunzione* (JOIN) su una base dati relazionale vengono realizzate attraverso l'istruzione SELECT, secondo le diverse forme consentite dalla sintassi di questa istruzione.

La restrizione e la proiezione sono già state utilizzate negli esempi di uso del comando SELECT nella sezione precedente. Rivediamole in dettaglio, assieme alle altre elencate nell'unità precedente.

4.5.1 L'operazione di restrizione

L'operazione di **restrizione**, che consente di ricavare da una relazione un'altra relazione contenente solo le righe che soddisfano una certa condizione, viene realizzata nel linguaggio SQL utilizzando la clausola WHERE nel comando SELECT.

Restrizione

Esempio

Per ottenere l'elenco di tutti i dipendenti che hanno *StipendioNetto* minore o uguale a 1000.0 euro, si opera una selezione sulla tabella *Dipendente*.
Con lo pseudocodice utilizzato per le operazioni dell'algebra relazionale scriveremo:

restrict Dipendente where StipendioNetto ≤ 1000,0

In SQL la precedente interrogazione viene riscritta:

SELECT * FROM Dipendente WHERE StipendioNetto < = 1000.0;

4.5.2 L'operazione di proiezione

Proiezione

L'operazione di **proiezione**, che permette di ottenere una relazione contenente solo alcuni attributi della relazione di partenza, si realizza indicando accanto alla parola SELECT l'elenco degli attributi richiesti.

Esempio

Al fine di ottenere l'elenco dei dipendenti visualizzando soltanto *Cognome*, *Nome* e *LivelloRetributivo*, si deve effettuare una proiezione sulla tabella *Dipendente* estraendo soltanto le colonne corrispondenti.
Con lo pseudocodice utilizzato per le operazioni dell'algebra relazionale scriveremo:

project Dipendente on Cognome, Nome, Livello

In SQL la precedente interrogazione viene riscritta:

SELECT Cognome, Nome, Livello **FROM Dipendente**;

4.5.3 L'operazione di giunzione

Giunzione naturale

Abbiamo visto che il comando SELECT può operare il prodotto su più tabelle, indicandone i nomi (separati da virgola) dopo la parola chiave FROM; scrivendo dopo la parola WHERE i nomi degli attributi che corrispondono nelle due tabelle (legati tra loro dal segno "="), si realizza in pratica l'operazione di **giunzione naturale join** (chiamata anche **inner-join** o **equi-join)**.

Qualora ci sia l'esigenza di maggiore chiarezza nella descrizione del comando, oppure ci siano due attributi con lo stesso nome in due tabelle diverse, è opportuno, come già sappiamo, specificare il nome della tabella e il nome dell'attributo in dot-notation.

Esempio

Per la *giunzione naturale* tra dipendenti e ditte, in algebra relazionale scriveremo:

Dipendente.CodDip **join** Azienda.CodDip

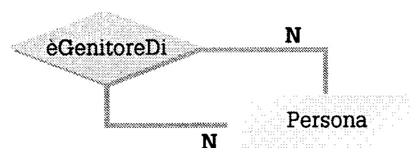
In SQL scriveremo:

SELECT * FROM Dipendente, Azienda
WHERE Dipendente.CodDip = **Azienda**.CodDip;

Self-join

Utilizzando gli alias (cioè la clausola **AS**) è possibile effettuare operazioni di **self-join**. Consideriamo il seguente esempio, in cui nel modello ER abbiamo l'associazione *èGenitoreDi* sulla stessa tabella *Persona*.

✦ Figura A4.5
Un'associazione sulla stessa tabella



Persona(<u>CodPers</u>, Cognome, Nome)

ÈGenitoreDi(<u>CodPers1</u>, <u>CodPers2</u>)

Esempio

Se in SQL vogliamo avere una *tabella risultato* con il cognome e il nome delle persone accanto al cognome e nome dei genitori, dobbiamo scrivere:

```
SELECT Tab1.Cognome, Tab1.Nome, Tab2.Cognome, Tab2.Nome
FROM Persona AS Tab1, Persona AS Tab2
WHERE Tab1.CodPers1 = Tab2.CodPers2;
```

Come possiamo notare, abbiamo dato i ruoli di genitore e figlio rispettivamente alle chiavi *CodPers1* e *Codpers2*.

È da sottolineare che non tutti i dialetti SQL implementano le operazioni relazionali di *left-join*, *right-join* o *outer-join* viste nell'unità precedente.

4.5.4 Le operazioni di unione, intersezione e differenza

Per tradurre le operazioni dell'algebra relazionale di *unione*, *intersezione* e *differenza*, SQL utilizza tre operatori che si applicano ai risultati delle interrogazioni.

Tali operatori sono **UNION**, **INTERSECT**, **MINUS** rispettivamente per: *unione*, *intersezione* e *differenza simmetrica*.

UNION, INTERSECT e MINUS

Esempio

Consideriamo le seguenti relazioni:

Regista (<u>CodRegista</u>, Cognome, Nome)
Attore (<u>CodAttore</u>, Cognome, Nome)

Per ottenere i registi che sono stati anche attori scriveremo:

```
(SELECT Cognome, Nome FROM Regista)
   INTERSECT
(SELECT Cognome, Nome FROM Attore);
```

Le parentesi sono obbligatorie.

Per ottenere i registi che non sono mai stati attori scriveremo:

```
(SELECT Cognome, Nome FROM Regista)
   MINUS
(SELECT Cognome, Nome FROM Attore);
```

Per ottenere tutti i registi e tutti gli attori scriveremo:

```
(SELECT Cognome, Nome FROM Regista)
   UNION
(SELECT Cognome, Nome FROM Attore);
```

Nei tre casi precedenti le due relazioni *risultato parziale* delle SELECT devono essere compatibili per poter essere utilizzate come operandi degli operatori di *intersezione*, *unione* e *differenza*.

4.6 Interrogazioni parametriche

Consideriamo la seguente query:

```
SELECT * FROM Clienti
WHERE NomeCliente = "Rossi" AND DataNascita = '26/07/89';
```

Per visualizzare i dati relativi al cliente "Bianchi" dovremmo riscrivere la query.

È possibile parametrizzare una query in modo da scriverla una sola volta e sfruttarla per diversi valori del parametro. Per fare questo, è sufficiente far precedere la query dalla seguente dichiarazione:

```
PARAMETERS <Parametro1>[:<TipoParametro1>], ..., <ParametroN>
                                                [:<TipoParametroN];
```

dove, al momento dell'interrogazione, verrà chiesto di immettere il valore per il

parametro e, se il tipo del parametro è presente, verrà controllato il tipo del valore immesso, altrimenti il controllo non verrà effettuato.

Esempio

L'esempio precedente può allora essere riscritto nel seguente modo:

```
PARAMETERS Nome, Data: Date;
SELECT * FROM Clienti
WHERE NomeCliente = Nome AND DataNascita = Data;
```

Verrà così chiesto di immettere:

- il *valore* del nome del cliente, sul quale il sistema non effettuerà alcun controllo;
- la *data* di nascita del cliente, sul cui valore il sistema effettuerà un controllo di tipo.

4.7 Le asserzioni

Dopo aver visto i vincoli interni e quelli referenziali, occorre introdurre per completezza un ultimo tipo di vincolo che è possibile specificare in SQL: le **asserzioni**.

Gli unici vincoli interrelazionali finora visti sono stati quelli referenziali.

Vincolo interrelazionale

Vediamo ora come può essere espresso un generico **vincolo interrelazionale**, che non sia necessariamente referenziale. I vincoli finora visti, inoltre, sono stati associati a elementi dello schema inserendoli all'interno di altre istruzioni SQL. Le *asserzioni* sono esse stesse elementi dello schema e si utilizzano apposite istruzioni SQL per crearle.

Per creare un'asserzione utilizzeremo la seguente sintassi:

```
CREATE ASSERTION <NomeAsserzione> CHECK(<Condizione>);
```

dove <Condizione> deve essere sempre verificata (cioè sempre vera) e qualsiasi modifica alle relazioni che la renda falsa deve essere rifiutata.

Esempio

Modifichiamo l'esempio degli attori e dei registi già osservato nel paragrafo precedente, introducendo le seguenti relazioni:

Regista (<u>CodRegista</u>, Cognome, Nome, CompensoMinimo)
Film (<u>CodFilm</u>, Regia, Titolo, Budget)

Vogliamo che sia sempre verificato il seguente vincolo: *"non devono esistere registi che abbiano un compenso minimo superiore a 100 000 euro per film il cui budget è inferiore a 5 000 000 euro"*. Questo vincolo coinvolge più relazioni e può essere definito nel seguente modo:

```
CREATE ASSERTION ControllaBudget CHECK
(NOT EXISTS
   (SELECT * FROM Film, Regista
      WHERE Film.Regia = Regista.CodRegista
         AND
            Regista.CompensoMinimo > 100000
         AND
            Film.Budget ≤ 5000000
   )
);
```

Ricordiamo ancora che l'introduzione di un alto numero di vincoli aumenta la sicurezza di una base di dati per quanto riguarda la sua integrità e congruenza interna. Bisogna però dire che rallenta anche le operazioni di *modifica*, *inserimento* e *interrogazione*. È necessario fissare l'obiettivo principale: ottenere una maggiore velocità di esecuzione o una assoluta rigorosità nelle modifiche.

Parleremo ancora del predicato NOT EXIST. Tale predicato (utilizzato nelle condizioni) restituisce falso se la tabella restituita dalla SELECT possiede qualche riga.

Riassumendo quanto esposto sui vincoli:

TIPO DEL VINCOLO RELAZIONALE	CON CHE COSA È IMPLEMENTATO	DOVE È DICHIARATO	QUANDO È ATTIVATO
Di attributo NON NULLO	NOT NULL	CREATE TABLE	Inserimento/modifica di una tupla
Di dominio	Clausola CHECK	CREATE DOMAIN CREATE TABLE Ad es: CHECK (età < 120)	In caso di inserimento in una relazione o aggiornamento dell'attributo
Di singola ennupla	Clausola CHECK	CREATE TABLE	Inserimento/modifica di una tupla
Intrarelazionale su più ennuple	Clausola CHECK	CREATE TABLE	Inserimento di una tupla
Interrelazionale sui valori qualsiasi di attributi	Clausola CHECK nelle asserzioni	CREATE ASSERTION	In caso di modifica su una qualsiasi delle relazioni menzionate
Referenziale di chiave primaria	Clausola PRIMARY KEY	CREATE TABLE	Inserimento/modifica di una tupla
Referenziale su chiave esterna	Clausola FOREIGN KEY...REFERENCES	CREATE TABLE	Cancellazione/modifica di una chiave esterna

4.8 Le funzioni di aggregazione

SQL possiede alcune funzioni predefinite, utilissime in molte circostanze in cui occorre effettuare conteggi, somme, calcoli di medie o altro ancora. Tali funzioni si applicano a una colonna di una tabella. La loro sintassi è:

<FunzioneDiAggregazione> ([**DISTINCT**] <Attributo>)

dove <FunzioneDiAggregazione> può essere:

- l'istruzione **COUNT**, che conteggia il *numero di elementi* della colonna specificata in <Attributo>;
- l'istruzione **MIN**, che restituisce il *valore minimo* della colonna specificata in <Attributo>;
- l'istruzione **MAX**, che restituisce il *valore massimo* della colonna specificata in <Attributo>;
- l'istruzione **SUM**, che restituisce la *somma degli elementi* della colonna specificata in <Attributo>;
- l'istruzione **AVG**, che restituisce la *media aritmetica* degli elementi della colonna specificata in <Attributo>.

Esempio

Consideriamo la seguente relazione:

Dipendente (CodDip, Cognome, Nome, Livello, DataStipendio, Stipendio)

Per conoscere il numero di dipendenti con stipendio maggiore di 2000 euro, scriveremo:

```
SELECT COUNT (Stipendio)
FROM Dipendente WHERE Stipendio > 2000;
```

Per conoscere l'esborso totale per gli stipendi dei dipendenti scriveremo:

```
SELECT SUM (Stipendio)
FROM Dipendente;
```

Per conoscere il valore massimo degli stipendi dei dipendenti, scriveremo:

```
SELECT MAX (Stipendio)
FROM Dipendente;
```

Le funzioni MIN e MAX operano anche sugli attributi di tipo carattere.

Esempio

Per conoscere l'ultimo cognome dei dipendenti scriveremo:

```
SELECT MAX (Cognome)
FROM Dipendente;
```

4.9 Ordinamenti

Finora non abbiamo fatto alcuna ipotesi sull'ordine in cui possono apparire le righe di una *tabella risultato* di una query.

ORDER BY

In SQL è possibile ordinare tali righe utilizzando alcune clausole, che vediamo di seguito, dell'istruzione SELECT:

ORDER BY <Attributo1>[**ASC** | **DESC**], ..., <AttributoN>[**ASC** | **DESC**]

dove **ASC** e **DESC** stanno per ordine *crescente* e *decrescente*. Per default l'ordinamento è crescente.
L'ordinamento viene eseguito prima sull'*Attributo1*, poi, a parità di ordinamento sull'*Attributo1*, si ordina sulla base dell'*Attributo2* e così via.

Esempio

Per ordinare i dipendenti in *ordine alfabetico* scriveremo:

```
SELECT *
FROM Dipendente
ORDER BY Cognome, Nome;
```

Per ordinare i dipendenti con stipendi maggiori di 3000 euro in *ordine decrescente*, cioè dallo stipendio più alto a quello più basso (ma sempre maggiore di 3000 euro), scriveremo:

```
SELECT Cognome, Nome, Stipendio
FROM Dipendente
WHERE Stipendio > 3000
ORDER BY Stipendio DESC;
```

4.10 Raggruppamenti

Clausole di raggruppamento

Le funzioni di aggregazione sono generalmente abbinate alle clausole di raggruppamento, la cui sintassi è:

GROUP BY <Attributo1>, ..., <AttributoN> [**HAVING** <CondizioneGruppo>]

Con questa clausola il significato della *select* è il seguente:

- viene eseguito il *prodotto* delle tabelle presenti nella clausola WHERE;
- su tale prodotto si effettua una *restrizione* in base alla clausola WHERE;
- la *tabella risultante* viene logicamente partizionata in gruppi di righe. Due righe appartengono allo stesso gruppo se hanno gli stessi valori per gli attributi elencati nella clausola GROUP BY;
- tutti i gruppi che *non soddisfano* la clausola HAVING vengono eliminati.

Esempio

Per raggruppare i dipendenti in base al loro livello e conoscere lo stipendio medio per livello, possiamo scrivere:

```
SELECT Livello, AVG(Stipendio)
FROM Dipendente
GROUP BY Livello;
```

A video verrà visualizzato:

Livello	AVG(Stipendio)
5	1600.00
6	1700.00
7	2000.00

✦ Figura A4.6
Un esempio di utilizzo di AVG

Per raggrupparli per livello e in più conoscere stipendio medio e numero di dipendenti in quel livello, scriveremo:

```
SELECT Livello, AVG(Stipendio), COUNT(Livello)
FROM Dipendente
GROUP BY Livello;
```

A video verrà visualizzato:

Livello	AVG(Stipendio)	COUNT(Livello)
5	1600.00	10
6	1700.00	8
7	2000.00	5

✦ Figura A4.7
Un esempio di utilizzo di AVG e COUNT

Per raggrupparli in livelli solo per quelli maggiori del sesto, scriveremo:

```
SELECT Livello, AVG(Stipendio), COUNT(Livello)
FROM Dipendente
GROUP BY Livello
HAVING Livello > 6;
```

A video verrà visualizzato:

Livello	AVG (STIPENDIO)	COUNT(LIVELLO)
7	2000.00	5

✦ Figura A4.8
Un utilizzo della clausola HAVING

4.11 Interrogazioni nidificate e sottointerrogazioni

Sottointerrogazioni (subquery)

Per rispondere a query complesse è possibile strutturare più SELECT. Ciò consente di costruire un'interrogazione al cui interno sono presenti altre interrogazioni, dette **sottointerrogazioni** o **subquery.**

Esempio

Consideriamo le seguenti relazioni relative all'utilizzo di laboratori da parte di una classe di studenti:

Laboratorio (CodLab, NumPosti, NomeLab)
Classe (CodClasse, NumPosti)
Utilizza (CodLab, CodClasse)

e consideriamo la seguente interrogazione:

Q1. Vogliamo conoscere il nome dei laboratori utilizzati dalla classe "A45".
Per rispondere alla Q1 scriveremo:

```
SELECT NomeLab
FROM Laboratorio, (SELECT CodLab FROM Utilizza
                   WHERE CodClasse = 'A45')AS Lab
WHERE Lab.CodLab = Laboratorio.CodLab;
```

sottointerrogazione

Come possiamo vedere, abbiamo annidato due sottointerrogazioni (due SELECT) all'interno della query principale.

4.11.1 Conservazione dei risultati parziali

In un processo di interrogazioni annidate, spesso è utile conservare le *tabelle risultato* di alcune sottointerrogazioni.

SELECT INTO

Per fare questo, occorre aggiungere all'istruzione SELECT la clausola **INTO**, seguita dal nome da assegnare alla nuova tabella.

Riprendiamo l'esempio del paragrafo precedente. Un utilizzo delle *tabelle risultato* parziale è mostrato di seguito:

```
SELECT CodLab INTO Temp1 FROM Utilizza
WHERE CodClasse = 'A45';
```

dove abbiamo creato una tabella temporanea chiamata *Temp1*, relativa alla sottointerrogazione "seleziona i codici dei laboratori della classe "A45".

Poi scriveremo:

```
SELECT CodLab, NumPosti
FROM Temp1, Laboratorio
WHERE Temp1.CodLab = Laboratorio.CodLab;
```

Esempio

Sia data la seguente relazione, che si riferisce ai dipendenti di un'azienda:

Dipendente (<u>CodDip</u>, Cognome, Nome, DataStipendio, Stipendio)

Se si vuole creare una nuova tabella *Settembre*, relativa agli stipendi del mese di settembre dei dipendenti di quella azienda, scriveremo:

```
SELECT * INTO Settembre
FROM Dipendente
WHERE DataStipendio < '01/10/2001' AND DataStipendio > '31/08/2001';
```

Le colonne della nuova tabella avranno gli stessi nomi, formati e dimensioni degli attributi della tabella di origine.

INSERT INTO

Se invece si vogliono aggiungere le righe della tabella risultante alle righe di una tabella già esistente, si deve usare l'istruzione **INSERT INTO**, la cui sintassi è:

INSERT INTO <NomeTabellaDestinazione>
<Query>;

dove il risultato della <Query> viene inserito in <NomeTabellaDestinazione>.

Esempio

Se vogliamo aggiungere le righe relative ai pagamenti di settembre alle righe di una tabella già esistente *StipendiAnno* (che conserva tutti gli stipendi dell'anno in corso), dobbiamo scrivere:

```
INSERT INTO StipendiAnno
SELECT * FROM Settembre;
```

4.11.2 Sottointerrogazioni che producono un singolo valore

Finora abbiamo sempre supposto che il risultato di un'interrogazione producesse

una tabella. Spesso accade che, in conseguenza dell'impostazione della query, il risultato dell'interrogazione restituisca il valore di un solo attributo.

Quando si è sicuri che una sottointerrogazione produca un singolo valore come risultato, è possibile utilizzare tale sottointerrogazione nelle espressioni delle query.

Esempio

Consideriamo le seguenti relazioni:

Regista (<u>CodRegista</u>, Cognome, Nome, CompensoMinimo)
Film (<u>CodFilm</u>, Regia, Titolo, Budget)

Se vogliamo visualizzare il regista del film *Via col vento*, scriveremo:

```
SELECT Cognome, Nome
FROM Regista
WHERE CodRegia = (SELECT Regia
                  FROM FILM
                  WHERE Titolo = 'Via col vento');
```

Come è facilmente verificabile, in questo caso un analogo risultato poteva ottenersi componendo diversamente l'interrogazione.

4.11.3 Sottointerrogazioni che producono valori appartenenti a un insieme

Predicati per query complesse

Nella costruzione delle sottointerrogazioni, è possibile utilizzare i seguenti predicati che semplificano la scrittura di query complesse:

- **ANY** e **ALL**;
- **IN** e **NOT IN**;
- **EXISTS** e **NOT EXISTS**.

ANY e ALL

ANY e **ALL** vengono utilizzati nelle condizioni delle query del tipo:

<Espressione> <OperatoreRelazionale> **ANY | ALL** <Insieme>

dove <Insieme> indica i valori appartenenti a una colonna con il seguente significato:

- **ANY** indica che la condizione precedente è vera se il confronto è vero per almeno uno dei valori presenti nell'<Insieme>;
- **ALL** indica che la condizione precedente è vera se il confronto è vero per tutti i valori dell'<Insieme>.

IN e NOT IN

IN e **NOT IN** vengono utilizzati nelle condizioni delle query del tipo:

<Espressione> **IN | NOT IN** <Insieme>

dove <Insieme> indica i valori appartenenti a una colonna con il seguente significato:

- **IN** indica che la condizione precedente è vera se il valore dell'espressione appartiene a quelli dell'insieme;
- **NOT IN** indica che la condizione precedente è vera se il valore dell'espressione non appartiene a quelli dell'insieme.

EXISTS e NOT EXISTS

EXISTS e **NOT EXISTS** vengono utilizzati nelle condizioni delle query del tipo:

EXISTS | NOT EXISTS <Query>

dove la condizione EXISTS è vera se la <Query> restituisce almeno una riga.

Esempio

Consideriamo le seguenti relazioni:

Automobile (<u>CodAuto</u>, Marca, Modello, Targa, Prezzo, CodProp)
ÈDotata (<u>CodAuto</u>, <u>CodAcc</u>)
Accessorio (<u>CodAcc</u>, Descrizione, PrezzoAcquisto, PrezzoVendita, Quantità)

Per rispondere alla query "quali sono gli accessori che hanno un prezzo inferiore **ad almeno uno** di quelli che hanno quantità > 10", scriveremo:

```
SELECT * FROM Accessorio
WHERE PrezzoVendita < ANY
   (SELECT CodAcc FROM Accessorio
   WHERE Quantità > 10);
```

Per rispondere alla query "quali sono gli accessori che hanno un prezzo inferiore **a tutti** quelli che hanno quantità > 10", scriveremo:

```
SELECT * FROM Accessorio
WHERE PrezzoVendita < ALL
   (SELECT CodAcc FROM Accessorio
   WHERE Quantità > 10);
```

Per rispondere alla query "quali sono le automobili sui cui accessori si ha un ricarico superiore a 100 euro", scriveremo:

```
SELECT CodAuto FROM ÈDotata
WHERE CodAcc IN
   (SELECT CodAcc FROM Accessorio
   WHERE (PrezzoVendita – PrezzoAcquisto) > 100);
```

Per rispondere alla query "quali sono le automobili per le quali esiste almeno un accessorio con un prezzo maggiore di 3000 euro", scriveremo:

```
SELECT CodAuto FROM ÈDotata
WHERE EXIST
   (SELECT CodAcc FROM Accessorio, ÈDotata
   WHERE PrezzoVendita > 3000 AND ÈDotata CodAcc = Accessorio.CodAcc);
```

5 Istruzioni DCL di SQL

DCL

Sicurezza dei dati

Una volta creato lo schema relazionale, tramite appositi comandi che fanno parte della parte **DCL** di SQL è possibile impostare le politiche relative alla **sicurezza** dei dati.

Quando si parla di sicurezza dei dati occorre distinguere i seguenti aspetti:

- sicurezza da **guasti** *hardware* e *software*;
- sicurezza da **accessi** *non autorizzati*. Per garantirsi da questa eventualità è necessario:
  - stabilire i **diritti di accesso**;
  - stabilire le **viste**, cioè le modalità con le quali gli utenti possono "vedere" la base di dati;
- sicurezza nelle **transazioni**, per ottenere la quale occorre preservare **integrità e consistenza** dei dati.

Tralasciamo per il momento la sicurezza sui guasti hardware e software, sui quali torneremo nella prossima unità sul DBMS e soffermiamoci sugli altri aspetti.

5.1 Le viste

CREATE TABLE

Le relazioni definite con l'istruzione **CREATE TABLE** sono presenti fisicamente nella base di dati. È possibile utilizzare una qualsiasi organizzazione fisica per la loro memorizzazione.

In SQL è possibile definire un'altra *classe di relazioni*, chiamate **viste**, che non sono fisicamente memorizzate nella base di dati, ma che possono essere definite logicamente. Una vista, infatti, pur non essendo memorizzata, è ottenuta tramite un'operazione di "mapping" ("mappatura") con le tabelle effettivamente memorizzate. A questo punto è possibile definire una vista, modificarla e interrogarla.

Per creare una vista si utilizza la seguente sintassi:

CREATE VIEW <NomeTabellaVista> **AS** <Query>;

dove:

- <NomeTabellaVista> è il NOME assegnato alla fittizia tabella della vista;
- <Query> è una normale query formulata con l'istruzione SELECT.

Esempio

Consideriamo ancora le seguenti relazioni:

Automobile (<u>CodAuto</u>, Marca, Modello, Targa, Prezzo, CodProp)
Accessorio (<u>CodAcc</u>, Descrizione, PrezzoAcquisto, PrezzoVendita, Quantità)
Proprietario (<u>CodFiscale</u>, Cognome, Nome)

Una vista molto utile al magazziniere è quella che gli consente di visualizzare gli accessori che devono essere ordinati, poiché sono terminati in magazzino.
Possiamo allora definire la seguente vista:

```
CREATE VIEW DaOrdinare AS
SELECT * FROM Accessorio
WHERE Quantità = 0;
```

La tabella *DaOrdinare* è definita solo logicamente. Essa è fisicamente rappresentata da una parte della tabella *Accessorio*, vale a dire la parte delle tuple in cui il valore di quantità è pari a 0. Il resto della tabella viene oscurato alla vista.

Accessorio	CodAcc	Descrizione	PrezzoAcquisto	PrezzoVendita	Quantità
	A01	Batteria	100,00	120,0	3
	A02	Tergicristalli	80,00	100,00	0
	A03	Specchietto Dx	50,00	75,00	1
	A04	Specchietto Sin	50,00	75,00	0

DaOrdinare	CodAcc	Descrizione	PrezzoAcquisto	PrezzoVendita	Quantità
	A02	Tergicristalli	80,00	100,0	0
	A04	Specchietto Sin	50,00	75,00	0

✦ Figura A4.9
Un esempio di vista su una tabella

Come si può facilmente intuire, il motivo per cui si creano le viste è quello di fornire a un gruppo di utenti una versione semplificata o parziale di una realtà che può essere anche molto complessa.

Categorie diverse di utenti possono interagire con la base di dati utilizzando il loro punto di vista e trascurando il punto di vista altrui.

Esempio

Nell'esempio precedente il punto di vista era quello del *magazziniere*. Se invece vogliamo il punto di vista del *venditore* e non vogliamo che si conosca il prezzo di acquisto di un accessorio, ma solo quello di vendita, scriveremo:

```
CREATE VIEW Vendita AS
   SELECT CodAcc, Descrizione, PrezzoVendita, Quantità
   FROM Accessorio;
```

Ogni modifica apportata sulla *tabella vista Vendita* e su quella *DaOrdinare* si ripercuote sulla tabella *Accessorio*.

Le viste sono quindi un vero e proprio **strumento di protezione dei dati**.

AS

Spesso si ricorre a viste che agiscono su più tabelle. Questo è perfettamente legale, poiché dopo la clausola **AS** è possibile inserire una qualsiasi query.

DROP VIEW

Per eliminare una vista si utilizza l'istruzione DROP VIEW, la cui sintassi è:

DROP VIEW <NomeTabellaVista>;

Sulle viste è possibile utilizzare l'istruzione GRANT.

Esempio

Per concedere i diritti di accesso in lettura all'utente Neri sulla vista *Vendita*, scriveremo:

```
GRANT SELECT ON Vendita TO Neri;
```

6 Un esempio completo di progettazione di una base di dati: la concessionaria di automobili

Riprendiamo l'esempio della concessionaria di automobili già introdotto nelle unità precedenti e vediamo alcune istruzioni SQL per:

- creare lo schema relazionale, utilizzando le istruzioni *DDL* di SQL;
- manipolare dati ed effettuare interrogazioni, utilizzando le istruzioni *DML* di SQL;
- impostare diritti di accesso, utilizzando le istruzioni *DCL* di SQL.

```
CREATE DOMAIN TipoMarca AS char(10)
CHECK (VALUE IN ('Marca1', 'Marca2', 'Marca3', 'Marca4', 'Marca5'));

CREATE DOMAIN TipoRiparazione AS char(10)
CHECK (VALUE IN ('Motore', 'Frizione', 'Testata', 'Marmitta', 'Carrozzeria', 'ImpElettrico',
                 'ImpFrenante', 'ImpRaffreddamento'));

CREATE DOMAIN TipoLivello AS char(1)
CHECK (VALUE BETWEEN '0' AND '9');

CREATE DOMAIN TipoCarb AS char(1)
CHECK (VALUE IN ('Benzina', 'Diesel', 'Metano', 'Misto', 'Elettrica');

CREATE TABLE Proprietario
   (CodFiscale     char(16)          NOT NULL,
   Cognome         char(20),
   Nome            char(20),         NOT NULL,
   PRIMARY KEY (CodFiscale)
   );

CREATE TABLE Riparazione
   (CodRip         char(11)          NOT NULL,
   Tipo            TipoRiparazione,
   Gravità         TipoLivello,
   Spesa           Integer(10)       NOT NULL,
   PRIMARY KEY (CodRip),
   CHECK (Spesa > 0)
   );
```

```
CREATE TABLE Optional
    (CodOpt         char(5)          NOT NULL,
    Descrizione     char(20),
    Prezzo          char(20)         NOT NULL,
    PRIMARY KEY (CodOpt),
    CHECK (Prezzo > 0)
    );

CREATE TABLE Motore
    (CodMotore      char(5)          NOT NULL,
    Cilindrata      Integer,
    Carburante      TipoCarb,
    PRIMARY KEY (CodMotore),
    CHECK (Cilindrata < 6000)
    );
```

Insieme alla clausola **FOREIGN KEY** definita in Automobile, traduce il vincolo referenziale: $VR_{CodMotore}(Motore) \subseteq VR_{CodMotore}(Automobile)$

```
CREATE TABLE Carrozzeria
    (CodCarr        char(5)          NOT NULL,
    NumTelaio       char(20),
    PRIMARY KEY (CodCarr)
    );
```

Insieme alla clausola **FOREIGN KEY** definita in Automobile, traduce il vincolo referenziale: $VR_{CodCarr}(Carrozzeria) \subseteq VR_{CodCarr}(Automobile)$

```
CREATE TABLE Ruota
    (CodRuota       char(5)          NOT NULL,
    Diametro        dec(2,2),
    Larghezza       Integer,
    PRIMARY KEY (CodRuota)
    CHECK (Spesa > 0),
    );
```

Insieme alla clausola **FOREIGN KEY** definita in Automobile, traduce il vincolo referenziale: $VR_{CodRuota}(Ruota) \subseteq VR_{CodRuota}(Automobile)$

```
CREATE TABLE Automobile
    (CodAuto      char(5)         NOT NULL,
    Marca         TipoMarca       NOT NULL,
    Modello       Smallint        NOT NULL,
    Targa         char(7)         UNIQUE,
    Prezzo        int(6)          NOT NULL,
    CodFiscale    char(16),
    CodMotore     char(5),
    CodCarr       char(5),
    CodRuota      char(5)
    PRIMARY KEY (CodAuto),
    FOREIGN KEY (CodFiscale),     REFERENCES Proprietario (CodFiscale),
    FOREIGN KEY (CodMotore),      REFERENCES Motore (CodMotore),
    FOREIGN KEY (CodCarr),        REFERENCES Carrozzeria (CodCarr),
    FOREIGN KEY (CodRuota),       REFERENCES Ruota (CodRuota),
    CHECK (Prezzo > 0)
    );
```

Targa è una chiave candidata

Traduce un vincolo di chiave primaria

Traduce un vincolo di dominio

Traducono i vincoli referenziali dell'aggregazione:
$VR_{CodMotore}$ (Automobile) $\subseteq$ $VR_{CodMotore}$ (Motore)
AND
$VR_{CodCarr}$ (Automobile) $\subseteq$ $VR_{CodCarr}$ (Carrozzeria)
AND
$VR_{CodRuota}$ (Automobile) $\subseteq$ $VR_{CodRuota}$ (Ruota)

```
CREATE TABLE AutoUsata
    (CodAuto        char(5)          NOT NULL,
    AnnoImm         date,
    KmPercorsi      integer,
    PRIMARY KEY (CodAuto),
```

```
    CHECK (AnnoImm < '01/01/1990'),
    CHECK (KmPercorsi < 300000)
    );

CREATE TABLE AutoNuova
    (CodAuto         char(5)          NOT NULL,
    AnniGaranzia     Smallint,
    PRIMARY KEY (CodAuto),
    );

CREATE TABLE Necessita
    (CodAuto         char(5)          NOT NULL,
    CodRip           char(5),         NOT NULL,
    PRIMARY KEY (CodAuto, CodRip),
    FOREIGN KEY (CodAuto)             REFERENCES Automobile (CodAuto)
        ON DELETE CASCADE,
    FOREIGN KEY (CodRip)              REFERENCES Riparazione (CodRip)
    );

CREATE TABLE ÈDotata
    (CodAuto         char(5)          NOT NULL,
    CodOpt           char(5),         NOT NULL,
    PRIMARY KEY (CodAuto, CodOpt),
    FOREIGN KEY (CodAuto)             REFERENCES Automobile (CodAuto),
    FOREIGN KEY (CodOpt)              REFERENCES Optional,) (CodOpt)
    );
```

È relativo al vincolo relazionale:
$VR_{CodRip}(Riparazione) \subseteq VR_{CodRip}(Necessita)$

Traduzione del vincolo V3 (Riparazione) del modello relazionale.

```
CREATE ASSERTION ControllaSpesa CHECK
(NOT EXISTS
    (SELECT * FROM Riparazione
    WHERE (Spesa < 1000 AND Gravità ≥ 7)
    )
)

CREATE ASSERTION NuovaUsataInAuto CHECK
(NOT EXISTS
    (
        (SELECT * FROM Automobile, AutoUsata
            WHERE (Automobile.CodAuto = AutoUsata.CodAuto)
        )
            INTERSECT
        (SELECT * FROM Automobile, AutoNuova
            WHERE (Automobile.CodAuto = AutoNuova.CodAuto)
        )
    )
)
```

Traduce il vincolo referenziale della generalizzazione:
$VR_{CodAuto}(AutoUsata) \subseteq VR_{CodAuto}(Automobile)$
XOR
$VR_{CodAuto}(AutoNuova) \subseteq VR_{CodAuto}(Automobile)$

Va verificato che l'insieme intersezione delle tuple di auto nuove e di auto usate sia nullo.

```
CREATE ASSERTION AutoInNuovaUsata CHECK
(NOT EXISTS
   (SELECT * FROM Automobile
   MINUS
      ((SELECT * FROM AutoUsata)
       UNION
      (SELECT * FROM AutoNuova)
      )
   )
)
```

Traduce il vincolo referenziale della generalizzazione:
$VR_{CodAuto}(Automobile) \subseteq VR_{CodAuto}(AutoUsata)$
OR
$VR_{CodAuto}(Automobile) \subseteq VR_{CodAuto}(AutoNuova)$

Va verificato che ogni CodAuto di *Automobile* sia nell'insieme unione delle tuple di AutoNuova o AutoUsata

Come possiamo notare, molti dei vincoli referenziali del modello relazionale sono stati tradotti in SQL.

Occorrerebbe ora creare anche gli *indici*, anche se questa è una tipica attività della progettazione fisica che vedremo nella prossima unità.

6.1 Istruzioni DML di SQL: analisi delle funzioni sui dati

Analisi delle funzioni sui dati

Vediamo alcune operazioni sulla base di dati appena creata:

O1 - Inserire una riparazione per una particolare auto usata;
O2 - Aumentare di un anno la garanzia su tutte le auto nuove;

Q1 - Selezionare il nome del proprietario del veicolo con targa "AB 450 TS";
Q2 - Elencare le targhe delle automobili che sono state acquistate a "settembre";

Q3 - Elencare le automobili usate con prezzo inferiore a un determinato valore;
Q4 - Elencare le riparazioni da effettuare sulle auto acquistate da un determinato cliente.

Vediamo adesso le operazioni in dettaglio.

O1 *Inserire una riparazione per una particolare auto usata.* Per soddisfare la richiesta scriveremo:

```
INSERT INTO Riparazione
(CodRip, Tipo, Gravità, Spesa)
VALUES
('R004', 'Frizione', 6, 500.00);

INSERT INTO Necessita
(CodAuto, CodRip)
VALUES
('A0045', 'R004');
```

L'operazione viene eseguita correttamente in quanto sono rispettati tutti i vincoli di integrità. In particolare:

- è soddisfatto il vincolo di **chiave primaria** (PRIMARY KEY), imposto dalla struttura della tabella, che *obbliga* a inserire un valore diverso per ogni chiave primaria;
- è soddisfatto il vincolo di **dominio** V4 (*Riparazione*), "spesa > 0", che **impedisce** l'inserimento di un preventivo di spesa negativo o nullo;
- è soddisfatto il vincolo **referenziale** dalla tabella *Necessita*:

```
FOREIGN KEY (CodRip) REFERENCES Riparazione (CodRip),
```

che *impedisce* la presenza di un *CodRip* in *Necessita* non presente in *Riparazione*;

- è soddisfatta l'**asserzione** *ControllaSpesa* che *impedisce* l'inserimento di una riparazione che costa meno di 1000 euro (per una gravità di livello maggiore o uguale al settimo).

O2 *Aumentare di un anno la garanzia su tutte le auto nuove*. Per soddisfare la richiesta scriveremo:

```
UPDATE AutoNuove
  SET AnniGaranzia = AnniGaranzia + 1;
```

Q1 *Selezionare il nome del proprietario del veicolo con targa "AB 450 TS"*. Per soddisfare la richiesta scriveremo:

```
SELECT Nome
  FROM Proprietario, Automobile
  WHERE Proprietario.CodFiscale = Automobile.CodFiscale
    AND Automobile.Targa = 'AB 450 TS';
```

Si esegue un'operazione relazionale di join tra *Proprietario* e *Automobile*, selezionando contemporaneamente il codice del veicolo desiderato.

Q2 *Elencare le targhe delle automobili che sono state acquistate a "settembre"*. Per soddisfare la richiesta scriveremo:

```
SELECT Targa
  FROM Automobile
  WHERE DataAcq > '31/08/2002' AND DataAcq < '01/10/2002';
```

Q3 *Elencare le automobili usate con prezzo inferiore a un determinato valore*. Per soddisfare la richiesta scriveremo:

```
SELECT *
  FROM Automobile, AutoUsata
  WHERE Automobile.CodAuto = AutoUsata.CodAuto
  AND prezzo < 10000;
```

Si esegue una join tra *Automobile* e *AutoUsata*.

Q4 *Elencare le riparazioni da effettuare sulle auto acquistate da un determinato cliente*.

Per soddisfare la richiesta scriveremo:

```
SELECT CodAuto INTO TEMP1
  FROM Proprietario, Automobile
  WHERE Proprietario.CodFiscale = Automobile.CodFiscale
  AND CodFiscale ="SLRFBA64M26E506F";
```

ottenendo tutte le auto usate acquistate dal cliente, comprese le auto nuove; se invece scriviamo:

```
SELECT CodAuto INTO TEMP2
  FROM Temp1, AutoUsata
  WHERE Temp1.CodAuto = AutoUsata.CodAuto;
```

otterremo tutte le auto usate acquistate dal cliente. Scrivendo invece:

```
SELECT CodRip INTO TEMP3
  FROM Temp2, Necessita
  WHERE Temp2.CodAuto = Necessita.CodAuto;
```

otterremo tutti i codici delle auto usate acquistate dal cliente che necessitano di riparazione. Infine, scrivendo:

```
SELECT *
  FROM Temp3, Riparazione
  WHERE Temp3.CodRip = Riparazione.CodRip;
```

otterremo tutte le informazioni relative alle riparazioni delle auto usate acquistate dal cliente.

Come possiamo notare, l'interrogazione si compone di una serie di join tra relazioni temporanee, introdotte per non annidare eccessivamente le interrogazioni.

6.2 Istruzioni DCL di SQL: definizione di viste

Definizione di viste

Creiamo due viste sulla base di dati: una per la categoria di utenti *Venditore* e l'altra per la categoria *Meccanico*.

```
SELECT *
   FROM Automobile, AutoUsata
   WHERE Automobile.CodAuto = AutoUsata.CodAuto
   AND prezzo < 10000;
```

```
CREATE VIEW InVendita AS
   SELECT *
   FROM Automobile, AutoNuova
   WHERE Automobile.CodAuto = AutoNuova.CodAuto;
```

Abbiamo creato un Join con *AutoNuova*, in modo da includere anche l'anno di garanzia tra gli attributi.

```
CREATE VIEW InRiparazione AS
   SELECT *
   FROM Riparazione, Necessita
   WHERE Riparazione.CodRip = Necessita.CodRip;
```

Conoscenze

■ VERO o FALSO?

	V	F
1. SET NULL è una politica di violazione di vincoli.	☐	☐
2. CASCADE è una politica di violazione di vincoli.	☐	☐
3. AVG è una funzione SQL per il calcolo della media tra gli elementi di una colonna.	☐	☐
4. COUNT è una funzione SQL per il calcolo del numero di elementi di una colonna.	☐	☐
5. Una SELECT può restituire un solo valore.	☐	☐
6. SQL è un linguaggio di programmazione.	☐	☐
7. SQL embedded necessita di un linguaggio ospite.	☐	☐

■ QUESITI

1. Che tipo di linguaggio è SQL?
2. Cosa significa che SQL assolve alle funzioni di DDL, DML e DCL?
3. Che differenza c'è tra SQL stand-alone e SQL embedded?
4. Perché SQL embedded necessita di un pre-processore SQL per il linguaggio ospite?
5. Come è possibile in SQL specificare un vincolo di chiave primaria?
6. Come è possibile in SQL indicare colonne che fanno parte di una chiave candidata?
7. Come è possibile in SQL specificare un vincolo referenziale?
8. In che cosa consiste l'istruzione CREATE DOMAIN?
9. Come si fa a modificare la struttura di una tabella appena creata per:
   - aggiungere una nuova colonna?
   - cancellare una colonna?
   - modificare il tipo di una colonna?
   - modificare il nome di una colonna?
10. A cosa servono le parole chiavi RESTRICT e CASCADE quando si vuole cancellare una tabella?
11. Come si implementa in SQL un vincolo di integrità di dominio?
12. Come si implementa in SQL un vincolo di integrità referenziale su chiave primaria?
13. Come si implementa in SQL un vincolo di integrità referenziale su chiave esterna?
14. Come si implementa in SQL un vincolo di integrità interrelazionale su valori di attributi non chiave?
15. In che cosa consistono le politiche di violazione di un vincolo?
16. Qual è la politica di default di violazione di un vincolo?
17. Qual è l'istruzione SQL per cancellare una riga di una tabella?
18. Quale verifica effettua SQL prima di cancellare una riga di una tabella?
19. Qual è l'effetto della clausola DISTINCT all'interno di una istruzione SELECT?
20. A che cosa servono le clausole alias?
21. Come si fa a conteggiare il numero di elementi di una specifica colonna?
22. Come si ordina alfabeticamente una tabella Clienti?
23. Come si fa a conservare il risultato parziale di una interrogazione per poterlo utilizzare successivamente?
24. A che cosa servono le clausole ANY e ALL?
25. A che cosa servono le clausole IN e NOT IN?
26. Che cosa sono le viste di una base di dati?
27. Quale delle seguenti istruzioni SQL rappresenta una proiezione?
    SELECT * **FROM** Tabella1;
    SELECT a1, a2 **FROM** Tabella1;
    SELECT * **FROM** Tabella1, Tabella2 **WHERE** k1 = k2;
    SELECT * **FROM** Tabella1 **WHERE** a1 = 'Rossi';
28. Quale delle seguenti istruzioni SQL rappresenta una restrizione?
    SELECT * **FROM** Tabella1;
    SELECT a1, a2 **FROM** Tabella1;
    SELECT * **FROM** Tabella1, Tabella2 **WHERE** k1 = k2;
    SELECT * **FROM** Tabella1 **WHERE** a1 = 'Rossi';

Competenze

■ ESERCIZI

1. Scrivi un'istruzione SQL per contare il numero di clienti presenti nella tabella Cliente.
2. Fai un esempio di interrogazione in cui si utilizza la clausola ANY oppure la clausola ALL.
3. Fai un esempio di interrogazione in cui si utilizza la clausola IN oppure la clausola NOT IN.
4. Crea una vista sulla tabella Cliente.
5. Assegna i diritti di modifica sulla tabella Cliente all'utente "Rossi".

6. Revoca tutti i diritti sulla tabella Cliente all'utente "Rossi".
7. Considerando l'esempio della concessionaria di automobili visto in questa unità:
   - crea un'asserzione che verifichi se il numero delle riparazioni ha superato soglia 100;
   - crea un'asserzione che impedisca l'inserimento di auto usate con più di km 300000 o più di 15 anni.
8. Date le seguenti relazioni:
   Studente (CodStud, Nome, Cognome)
   Interrogato (CodStud, CodMateria, Data, Voto)
   Materia (CodMateria, NomeMateria, NomeDocente, CognomeDocente)
   scrivi un'istruzione SQL per calcolare:
   - la media dei voti delle interrogazioni di uno studente;
   - il voto più basso di uno studente;
   - il voto più alto di uno studente;
   - il numero di interrogazioni dal primo gennaio al primo marzo.
9. Dato il seguente schema relazionale:
   BiciclettaAntica (CodBiciA, CodTipoA, MarcaA, ModelloA, PrezzoA)
   Bicicletta (CodBici, CodTipo, Marca, Modello, Prezzo)
   TipoBici (CodTipo, Descrizione)
   esegui in SQL le seguenti operazioni relazionali:
   union (BiciclettaAntica, Bicicletta)
   difference (BiciclettaAntica, Bicicletta)
   project Bicicletta on Marca, Modello, Prezzo
   restrict Bicicletta where Marca = "Velox"
   intersect (BiciclettaAntica, Bicicletta)
   Bicicletta.CodTipo **join TipoBici**.CodTipo
10. Dato lo schema relazionale precedente, effettua le seguenti operazioni in SQL:
    - O1 Crea le tabelle Bicicletta, BiciclettaAntica, TipoBici;
    - O2 Inserisci una nuova bicicletta;
    - Q1 Elenca tutte le biciclette di marca "Velox" sia antiche che moderne;
    - Q2 Elenca le biciclette di prezzo inferiore a 100 euro compresa la descrizione del tipo di bicicletta;
    - Q3 Raggruppa le biciclette in due fasce di prezzo: economiche (prezzo ≤ 100 euro) e professionali (prezzo > 100 euro).
11. Consideriamo l'esempio relativo alla mostra canina, il cui diagramma ER è stato presentato negli esempi svolti dell'unità A2 e il cui schema relazionale è stato presentato negli esempi svolti dell'unità A3.
    Utilizza le istruzioni DDL di SQL per creare le relative tabelle specificando:
    - gli attributi chiave;
    - i vincoli referenziali;
    - un vincolo di dominio;
    - un vincolo di ennupla.
12. Date le tabelle create nell'esercizio precedente, utilizza le istruzioni DML di SQL per:
    - inserire qualche tupla nelle rispettive tabelle;
    - modificare una delle tuple inserite;
    - cancellare una delle tuple inserite.

ESEMPI SVOLTI

1. Consideriamo l'esempio relativo alla mostra canina, il cui diagramma ER è stato presentato negli esempi svolti dell'unità A2 e il cui schema relazionale è stato presentato negli esempi svolti dell'unità A3.
   Traduci in SQL le interrogazioni proposte in A2 e precisamente:
   - Q1 Stila una classifica dei cani all'interno di ogni razza:

```
SELECT CodCane, NomeCane, NomeRazza,
    Punteggio
FROM Cane, Razza
WHERE Cane.CodRazza = Razza.CodRazza
GROUP BY NomeRazza
ORDER BY Punteggio;
```

   - Q2 Stila una classifica dal maggiore al minor punteggio, indipendentemente dalla razza del cane.

```
SELECT CodCane, NomeCane, NomeRazza,
    Punteggio
FROM Cane, Razza
WHERE Cane.CodRazza = Razza.CodRazza
ORDER BY Punteggio;
```

2. Consideriamo l'esempio relativo alla biblioteca, il cui diagramma ER è stato presentato negli esempi svolti dell'unità A2 e il cui schema relazionale è stato presentato negli esempi svolti dell'unità A3.
   Traduci in SQL le seguenti interrogazioni:
   - Q2 Elenca i libri prestati e non ancora restituiti:

```
SELECT CodSocio, CodLibro, DataPrestito
FROM HaPrestato
WHERE DataRestituzione IS NULL;
```

   - Q2 Visualizza tutti i libri prestati a un determinato socio. Per ogni libro visualizza titolo e autore:

```
SELECT Titolo, Autore
FROM HaPrestato, Libro
WHERE HaPrestato.CodLibro = Libro.CodLibro
    AND HaPrestato.CodSocio = 'S003';
```

Attività di riepilogo: l'esame di stato

SOLUZIONE DI UN TEMA DI INFORMATICA
ESAME DI STATO (Istituto Tecnico Industriale: Progetto "Abacus")

Tema

Un'associazione "Banca del Tempo" vuole realizzare una base di dati per registrare e gestire le attività dell'associazione. La "Banca del Tempo" (BdT) indica uno di quei sistemi organizzati di persone che si associano per scambiare servizi e/o saperi, attuando un aiuto reciproco.
Attraverso la BdT le persone mettono a disposizione il proprio tempo per determinate prestazioni (effettuare una piccola riparazione in casa, preparare una torta, conversare in lingua straniera ecc.) aspettando di ricevere prestazioni da altri. Non circola denaro, tutte le prestazioni sono valutate in tempo, anche le attività di segreteria.
Le prestazioni sono suddivise in categorie (lavori manuali, tecnologie, servizi di trasporto, bambini, attività sportive ecc.). Chi dà un'ora del suo tempo a qualunque socio, riceve un'ora di tempo da chiunque faccia parte della BdT. La base di dati dovrà mantenere le informazioni relative a ogni prestazione (quale prestazione, da chi è stata erogata, quale socio ha ricevuto quella prestazione, per quante ore e in quale data) per consentire anche interrogazioni di tipo statistico.
Il territorio di riferimento della BdT è limitato (un quartiere in una grande città o un piccolo comune) ed è suddiviso in zone; la base di dati dovrà contenere la mappa del territorio e delle singole zone, in forma grafica.

Si consideri la realtà di riferimento sopra descritta e si realizzino:

1) la progettazione concettuale della realtà indicata attraverso la produzione di uno schema (ad esempio ER, Entity-Relationship) con gli attributi di ogni entità, il tipo di ogni relazione e i suoi eventuali attributi;
2) una traduzione dello schema concettuale realizzato in uno schema logico (ad esempio secondo uno schema relazionale);
3) le seguenti interrogazioni espresse in algebra relazionale e/o in linguaggio SQL:
   a. produrre l'elenco dei soci (con cognome, nome e telefono) che hanno un "debito" nella BdT (coloro che hanno usufruito di ore di prestazioni in numero superiore a quelle erogate);
   b. data una richiesta di prestazione, visualizzare la porzione di mappa del territorio nel quale si trova il socio richiedente e l'elenco di tutti i soci che si trovano in quella zona in grado di erogare quella prestazione, visualizzandone il nome, cognome, indirizzo e numero di telefono;
   c. visualizzare tutti i soci che fanno parte della segreteria e che non hanno anche altri tipi di prestazione;
   d. produrre un elenco delle prestazioni ordinato in modo decrescente secondo il numero di ore erogate per ciascuna prestazione.
4) (Facoltativo: Sviluppato nel fine modulo successivo) Sviluppare il problema posto scegliendo una delle due seguenti proposte descrivendone le problematiche e le soluzioni tecniche adottabili:
   a. l'associazione BdT vuole realizzare un sito Web per rendere pubbliche le sue attività consentendo anche di effettuare on line le interrogazioni della base di dati previste nel punto 3;
   b. l'associazione BdT vuole realizzare un sito Web attraverso il quale possa raccogliere l'adesione on line di altri associati, attraverso il riempimento di un modulo da inviare via Internet all'associazione.

Metodologia di risoluzione

Il tema di Informatica dell'esame di stato è quasi sempre composto da due fasi:

- una fase prettamente teorica con la quale si richiede al candidato di esporre i fondamenti teorici che stanno alla base delle problematiche inerenti il tema da trattare;
- una fase implementativa che ricalca sommariamente le classiche fasi condotte durante la realizzazione di un sistema informatico. Trattandosi di una simulazione, però, si ha a disposizione soltanto il testo del tema e soprattutto un periodo di tempo molto limitato che occorrerà ottimizzare.

Occupiamoci dei temi che hanno per oggetto l'implementazione di una base di dati. Al fine di pervenire a una soluzione articolata e ben strutturata, consigliamo innanzitutto di leggere e comprendere il tema e in particolare le query alle quali rispondere e quindi di procedere nel seguente modo:

1) **Analisi del problema**
   È indispensabile riportare in forma discorsiva e sintetica gli aspetti salienti inerenti la realtà di interesse individuata dalla lettura del tema strutturandoli nel seguente modo:
   a. Visione d'insieme e previsione del carico di lavoro: descrivere il progetto a grandi linee e valutare la quantità di dati presenti nella base di dati.
   b. Contesto del sistema: collocare il sistema da realizzare all'interno di un sistema software già esistente o, se

realizzato *ex novo*, descrivere le eventuali interazioni con altri sistemi. Per questa sottofase è possibile utilizzare i diagrammi dei casi d'uso descritti nell'ultimo modulo di questo volume.

c. Ipotesi aggiuntive: effettuare le ipotesi aggiuntive e chiarificatrici su alcuni punti del tema di interpretazione non univoca o per i quali sono espressamente richieste ipotesi aggiuntive.
d. Vincoli: far emergere la valutazione e il rispetto degli eventuali vincoli imposti dal problema al fine di giustificare le scelte ipotizzate.
e. Approfondimenti: approfondire alcuni aspetti chiave o non specificati o particolarmente rilevanti.
f. Scelte implementative: motivare le scelte implementative nei casi in cui sia possibile proporre soluzioni alternative.
g. Strumenti hardware e software da utilizzare: descrivere gli strumenti software ed eventualmente hardware che si intende utilizzare per l'ambiente di sviluppo e per l'ambiente di esercizio.

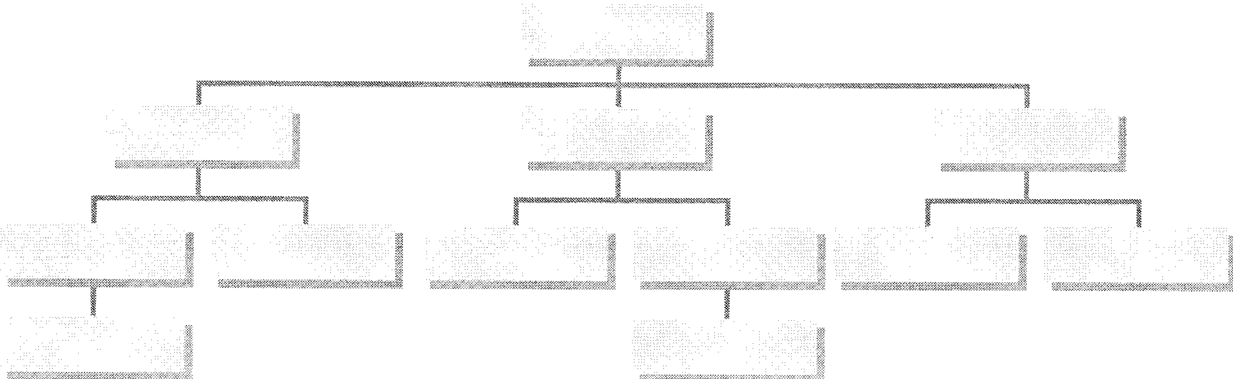
Ricordiamo che le fasi sopra descritte non devono essere rigidamente seguite ma possono essere affrontate in un altro ordine oppure omesse.

2) **Progettazione concettuale**
In questa fase devono essere descritti i dati e le loro relazioni servendosi del diagramma ER. Sarebbe opportuno riprendere le argomentazioni relative alle ipotesi aggiuntive e ai vincoli esaminandole questa volta da un punto di vista delle entità, degli attributi e delle associazioni individuate nel diagramma ER.

3) **Progettazione logica**
In questa fase si trasforma il diagramma ER in uno schema relazionale effettuando inoltre una verifica delle forme normali.

4) **Progettazione fisica**
In questa fase, nel caso in cui il tema lo richieda, vengono descritte funzionalità e caratteristiche proprie del DBMS che consentono di migliorare l'efficienza della gestione degli archivi della base di dati. Una classica attività svolta in questa fase è quella della definizione degli indici.

5) **Analisi delle funzionalità del sistema software**
In questa fase, sviluppata in base alle richieste del tema, vengono descritte le principali funzioni svolte dal sistema software. Funzioni fortemente influenzate, pur se si tratta di analisi, dal tipo di paradigma di programmazione che si vuole utilizzare. Anche in questa fase si può utilizzare il diagramma dei casi d'uso oppure più semplicemente un diagramma di funzioni che evidenzi le principali funzionalità e sottofunzionalità. Abbozzando l'elenco delle operazioni che compongono un'applicazione, è facile notare come al loro interno esiste una gerarchia attraverso la quale è possibile identificare operazioni di carattere generale e operazioni di dettaglio. Lo strumento visivamente più semplice per rappresentare tale gerarchia rimane l'albero gerarchico.



I livelli più alti verranno utilizzati per descrivere le funzioni di carattere generale, mentre quelli situati nelle diramazioni più basse contengono la descrizione delle operazioni di dettaglio.

6) **Analisi dell'interfaccia utente**
In questa fase bisogna dare un'idea dell'interazione del sistema con l'utente utilizzatore. Si descrive il tipo di interfaccia (grafica o testuale); si riportano le principali videate utilizzate ecc.

Svolgimento del tema

Analisi del problema

Da una prima lettura del testo emergono i seguenti punti:

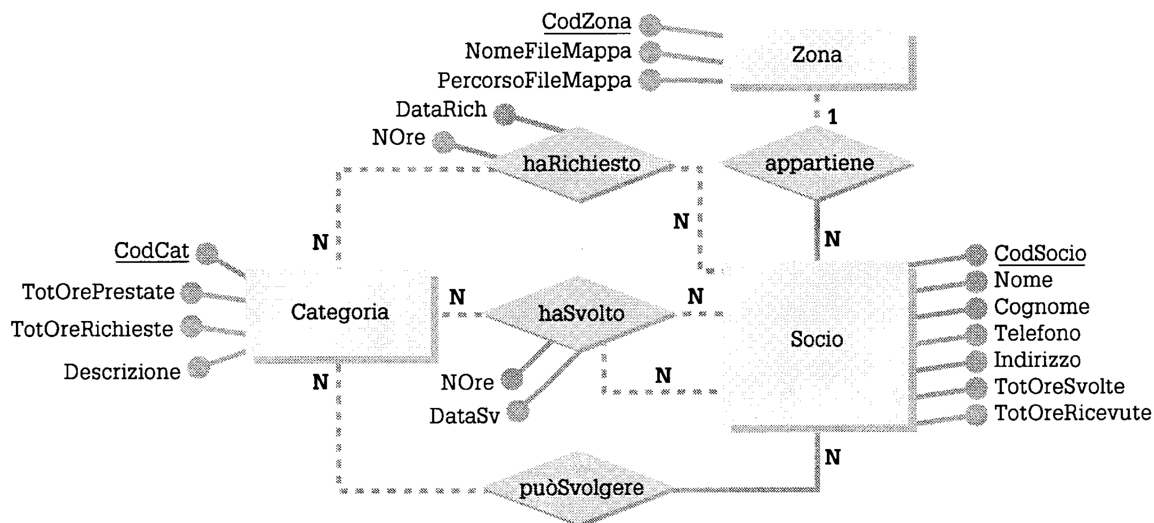
- **visione d'insieme e previsione del carico di lavoro**. La realtà di interesse riguarda un'associazione di volontariato di dimensioni ridotte in quanto si riferisce a persone che operano tutte in una zona territoriale limitata. La soluzione che proporremo pertanto, è impostata su questo carico di lavoro;

- **contesto del sistema**. Bisogna realizzare un sistema software *ex novo* non legato a precedenti sistemi informatici e quindi la sua interazione con il "mondo esterno" avviene esclusivamente, in questo caso, con l'utente utilizzatore;
- **ipotesi aggiuntive**. Il punto 3.d del tema si presta a più interpretazioni. Abbiamo ritenuto di interpretare le "prestazioni" richieste globalmente per Categorie. Questo implicherà l'aggiunta di appositi attributi per totalizzare le ore prestate e quelle richieste nell'entità che rappresenterà le categorie nel modello concettuale;
- **vincoli**. Ogni attributo del modello concettuale riguardante le ore svolte o richieste o ricevute deve essere maggiore o uguale a zero;
- **approfondimenti**. Considereremo una prestazione univocamente determinata dalla terna (data, socio, categoria), supponendo che possa essere richiesta al più una prestazione al giorno;
- **scelte implementative**:
  - con gli attributi totalizzatori delle ore prestate e richieste, introduciamo una ridondanza nel modello concettuale dei dati. Questo serve a ottimizzare i tempi di risposta alle query nonché a semplificarne la scrittura. Per non violare alcun vincolo di consistenza dei dati occorre aggiornare tali attributi quando:
    - si inserisce una richiesta di prestazione da svolgere;
    - si inserisce una prestazione svolta;
  - il CodZona può essere costruito mettendo insieme le coordinate di una mappa, così come avviene per "TuttoCittà": ad esempio, F3 individua la zona caratterizzata dalle coordinate F e 3;
  - consideriamo tra le categorie anche l'attività di segreteria;
  - si noti che haSvolto è un'associazione ternaria, in quanto l'entità Socio rappresenta tanto l'erogatore quanto il fruitore della prestazione di una data categoria;
- **strumenti hardware e software da utilizzare**. Utilizzeremo i seguenti strumenti software:
  - SQL per le interrogazioni;
  - modello ER per la progettazione concettuale;
  - modello relazionale per la progettazione logica;
  - pseudocodice lato server + HTML per il punto 4.

Progettazione concettuale

Utilizziamo gli attributi TotOrePrestate e TotOreRichieste per l'entità Categoria per quanto visto nelle ipotesi aggiuntive dell'analisi del problema.
Un esempio di alcuni vincoli:
VI:(Categoria.TotOrePrestate ≥ 0), V2:(Categoria.TotOreRichieste ≥ 0), V3:(haSvolto.NOre ≥ 0), V4:(haRichiesto.NOre ≥ 0).

Diagramma ER



Progettazione logica

Schema relazionale

Socio(<u>CodSocio</u>: Intero(5), Nome: Carattere(25), Cognome: Carattere(30), Indirizzo: Carattere(30), Telefono:Carattere(15), TotOreSvolte: Intero(4), TotOreRicevute: Intero(4), CodZona: Intero(4))
Categoria(<u>CodCat</u>: Intero(4), Descrizione: Carattere(30), TotOrePrestate: Intero(4), TotOreRichieste: Intero(4))
HaRichiesto(<u>CodCat</u>: Intero(4), <u>CodSocio</u>: Intero(5), <u>DataRich</u>: Data, NOre: Intero(2))

HaSvolto(CodCat: Intero(4), CodSocio1: Intero(5), CodSocio2: Intero(5), DataSv: Data, NOre: Intero(2))
PuòSvolgere(CodCat: Intero(4), CodSocio: Intero(5))
Zona(CodZona: Intero(4), NomeFileMappa: Carattere(20), PercorsoFileMappa: Carattere(30))

A queste tabelle, necessarie e sufficienti per risolvere completamente i quesiti posti dal tema, potranno essere aggiunte altre tabelle al fine di facilitare le operazioni di ricerca e di inserimento dei dati (indici, ecc.).
Un esempio di alcuni vincoli:
V1:(Categoria): "TotOrePrestate ≥ 0", V2(Categoria): "TotOreRichieste ≥ 0", V3:(HaSvolto): "NOre ≥ 0", V4(HaRichiesto): "NOre ≥ 0".

Interrogazioni SQL

a)
```
SELECT Cognome, Nome,Telefono
FROM Socio
WHERE TotOreRicevute > TotOreSvolte;
```

b)
```
SELECT Cognome, Nome, Telefono, Indirizzo
FROM PuoSvolgere, Socio
WHERE PuoSvolgere.CodSocio = Socio.CodSocio
AND PuoSvolgere.CodCat = Y
AND Socio.CodZona = (SELECT CodZona
                     FROM HaRichiesto, Socio
                     WHERE HaRichiesto.CodSocio = Socio.CodSocio
                       AND HaRichiesto.CodSocio = X AND HaRichiesto.CodCat = Y AND
                           HaRichiesto.DataRich = Z);
```

dove *X* contiene il codice del socio richiedente, *Y* contiene la categoria e *Z* la data della richiesta;

c)
```
SELECT Cognome, Nome, Telefono
INTO SocioSegretario
FROM Categoria, Socio
WHERE Descrizione = 'segreteria';

SELECT Cognome, Nome, Telefono
INTO SocioNonSegretario
FROM Categoria, Socio
WHERE Descrizione<>'segreteria';

SocioSegretario
INTERSECT
SocioNonSegretario;
```

d)
```
SELECT Descrizione, TotOreSvolte
FROM Categoria
ORDER BY TotOrePrestate DESC;
```

Analisi delle funzionalità
Dal tipo di query richiesto è possibile prevedere le principali funzionalità svolte dal nostro sistema software, rappresentate dal seguente diagramma di funzioni. Rimandiamo all'ultimo modulo per lo studio dei diagrammi dei casi d'uso, un altro strumento con il quale è possibile rappresentare tali funzionalità.