FONDAMENTI DI BASI DI DATI

Antonio Albano, Giorgio Ghelli, Renzo Orsini

12 febbraio 2019

INDICE

Prefazione

Dopo molti anni dalla pubblicazione della prima edizione del volume *Fondamenti di Basi di Dati* presso l'Editore Zanichelli, aggiornato con una seconda versione uscita nel 2005, è tempo di un'ulteriore ammodernamento, che coincide con la sua diffusione con un canale diverso da quello della carta stampata: il web, attraverso il sito http://fondamentidibasididati.it.

Riteniamo che questo materiale possa essere utile non solo per i classici corsi di *Basi di Dati*, fondamentali per le lauree in *Informatica* o *Ingegneria Informatica*, ma, data l'attenzione cha sta oggi avendo l'informatica in sempre piuù ampi settori della formazione e dell'istruzione ad ogni livello, in molti altri corsi di laurea e momenti formativi, in forma anche parziale, come abbiamo potuto sperimentare di persona durante questi ultimi anni.

Il passaggio alla nuova modalità di distribuzione, permettendo di mantenere aggiornati i contenuti, ci ha richiesto di modificare la struttura del sito di supporto al libro, che non avrà più la parte di errata corrige e di approfondimenti, ma conterrà materiale aggiuntivo utile per lo studio, come le soluzioni agli esercizi, i collegamenti ai software gratuiti, gli appunti del corso, e gli esempi scaricabili.

Organizzazione del testo

Il libro inizia presentando le ragioni che motivano la tecnologia delle basi di dati, ed i concetti principali che caratterizzano le basi di dati ed i sistemi per la loro gestione.

In maggior dettaglio, il Capitolo 2 si sofferma sulle nozioni fondamentali di modello informatico finalizzato al trattamento delle informazioni di interesse dei sistemi informativi delle organizzazioni e sui meccanismi d'astrazione per costruire modelli informatici. Il modello di riferimento è il modello ad oggetti, motivato non solo dalla sua naturalezza per la progettazione di basi di dati, ma anche per essere il modello dei dati dell'attuale tecnologia relazionale ad oggetti per basi di dati. Il formalismo grafico adottato si ispira all'*Unified Modeling Language*, UML, ormai uno standard dell'ingegneria del software.

Il Capitolo 3 presenta una panoramica del problema della progettazione di basi di dati, si sofferma sulle fasi dell'analisi dei requisiti e della progettazione concettua-

le usando il modello ad oggetti e il formalismo grafico proposto nel Capitolo 2, e descrive un metodo di lavoro per queste fasi.

I Capitoli 4 e 5 sono dedicati alla presentazione rigorosa del modello relazionale dei dati e ad un'introduzione alla teoria della normalizzazione. La scelta di dedicare questo spazio all'argomento è giustificata dal ruolo fondamentale che svolge il modello relazionale per la comprensione della tecnologia delle basi di dati e per la formazione degli addetti.

I Capitoli 6, 7 e 8 trattano il linguaggio relazionale SQL da tre punti di vista, che corrispondono ad altrettante categorie di utenti: (1) gli utenti interessati ad usare interattivamente basi di dati relazionali, (2) i responsabili di basi di dati interessati a definire e gestire basi di dati, (3) i programmatori delle applicazioni.

Il Capitolo 9 presenta una panoramica delle principali tecniche per la realizzazione dei sistemi relazionali. Vengono presentate in particolare la gestione delle interrogazioni e dei metodi di accesso e la gestione dell'atomicità e della concorrenza in sistemi centralizzati.

Ringraziamenti

L'organizzazione del materiale di questo testo è il risultato di molti anni di insegnamento deii corsi di *Basi di Dati* per le lauree in *Scienze dell'Informazione* e in *Informatica*.

Molte persone hanno contribuito con i loro suggerimenti e critiche costruttive a migliorare la qualità del materiale. In particolare si ringraziano i numerosi studenti che nel passato hanno usato le precedenti versioni del testo e Gualtiero Leoni per la sua collaborazione.

Si ringrazia infine l'Editore Zanichelli per il supporto che ci ha dato in questi anni, e, adesso che il libro è uscito dal suo catalogo, per il permesso di diffondere la nuova versione attraverso il web.

A. A.
G. G.
R. O.

Capitolo 7

SQL PER DEFINIRE E AMMINISTRARE BASI DI DATI

In questo capitolo vengono presentati i comandi SQL-92 per definire e amministrare basi di dati. Il materiale è organizzato presentando nell'ordine i comandi per:

- definire una base di dati e la struttura logica delle sue tabelle, memorizzate o calcolate;
- definire i vincoli d'integrità sui valori ammissibili degli attributi di un'ennupla, di ennuple diverse della stessa tabella (*vincoli intrarelazionali*) o di tabelle diverse (*vincoli interrelazionali*);
- definire aspetti procedurali nello schema della base di dati, sia sotto forma di *procedure memorizzate* che di *trigger*;
- definire aspetti fisici riguardanti criteri di memorizzazione dei dati e tipi di strutture di accesso per rendere più rapide certe operazioni su tabelle di grandi dimensioni. Mentre gli aspetti precedenti fanno parte del cosiddetto *schema logico*, gli aspetti fisici fanno parte del cosiddetto *schema fisico*;
- adeguare la base di dati a nuove esigenze che si manifestano durante il funzionamento a regime (*evoluzione dello schema*);
- limitare i dati accessibili e le modalità d'uso agli utenti autorizzati.

Poiché non tutti questi aspetti sono trattati dallo standard SQL-92, per alcuni di essi si presenteranno le soluzioni presenti in alcuni sistemi commerciali.

7.1 Definizione della struttura di una base di dati

Un DBMS permette la creazione di più schemi all'interno di un'unica base di dati. Uno schema, in questa terminologia, corrisponde ad un insieme di tabelle, memorizzate e calcolate, procedure, trigger, e agli altri "oggetti" che possono essere presenti in una base di dati, ed è associato ad un utente *proprietario*, che stabilisce la la disponibilità degli oggetti in esso contenuti ad altri utenti.

Negli esempi che seguono si useranno i comandi SQL-92 nella forma testuale, ma si tenga presente che di solito i sistemi commerciali prevedono anche strumenti di tipo grafico interattivi, facili da usare ma non standard. Questi strumenti generano poi i

comandi SQL che verranno eseguiti dal sistema. Come base di dati di riferimento si userà quella vista nel capitolo precedente, che per comodità si riporta in Figura 7.1.

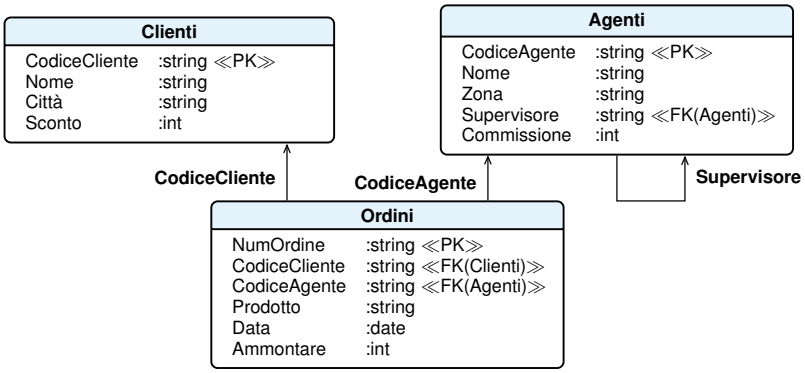

Figura 7.1: Rappresentazione grafica dello schema esempio

Premettiamo, infine, una caratteristica importante dei sistemi relazionali: lo schema di una base di dati si costruisce incrementalmente con opportuni comandi (ad esempio **CREATE TABLE**) dati interattivamente, o attraverso uno strumento grafico, e tutte le definizioni sono memorizzate nella "metabase di dati", o *catalogo* del sistema, la cui struttura verrà discussa più avanti. Non esiste, quindi, in generale, un "testo" memorizzato da qualche parte con tutte le definizioni che costituiscono lo schema; è però possibile usare opportuni programmi SQL che producono dinamicamente un testo di questo tipo interrogando il catalogo e stampando i risultati.

7.1.1 Base di dati

Lo schema di una base di dati viene creata con il comando:

CREATE SCHEMA *Nome* **AUTHORIZATION** *Utente*
Definizioni

dove *Nome* è il nome dello schema della base di dati che viene creata, *Utente* è il nome dell'utente proprietario, e *Definizioni* sono i comandi per la creazione degli elementi della base di dati (tabelle, viste, indici ecc.).

Come detto precedentemente, questi elementi possono essere creati o modificati in un qualunque momento successivo, durante una *sessione* d'uso dello schema della base di dati, con gli opportuni comandi.

Uno schema di base di dati può essere rimosso con il comando:

DROP SCHEMA *Nome* [**RESTRICT** | **CASCADE**]

Con la forma **RESTRICT** l'operazione fallisce se vi sono ancora dati, mentre con la forma **CASCADE** si rimuovono automaticamente tutti i dati in essa presenti.[1]

7.1.2 Tabelle

La forma base del comando di creazione di una tabella è la seguente:

CREATE TABLE Nome
"(" *Attributo Tipo* [*Vincolo* {, *Vincolo* }]
{, *Attributo Tipo* [*Vincolo* {, *Vincolo* }] }
[, *VincoloDiTabella* {, *VincoloDiTabella* }] ")"

I vincoli saranno discussi in dettaglio nella sezione 7.2. I tipi più comuni per i valori degli attributi sono:

- **CHAR**(n) per stringhe di caratteri di lunghezza fissa *n*;
- **VARCHAR**(n) per stringhe di caratteri di lunghezza variabile di al massimo *n* caratteri;
- **INTEGER** per interi con la dimensione uguale alla parola di memoria standard dell'elaboratore;
- **REAL** per numeri reali con dimensione uguale alla parola di memoria standard dell'elaboratore;
- **NUMBER**(p,s) per numeri con p cifre, di cui s decimali;
- **FLOAT**(p) per numeri binari in virgola mobile, con almeno p cifre significative;
- **DATE** per valori che rappresentano istanti di tempo (in alcuni sistemi, come Oracle), oppure solo date (e quindi insieme ad un tipo **TIME** per indicare ora, minuti e secondi).

Vediamo un esempio di definizione di una tabella, per ora senza vincoli:

```
CREATE  TABLE Clienti (
        CodiceCliente   CHAR(3),
        Nome            CHAR(30),
        Città           CHAR(30),
        Sconto          INTEGER);
```

Una tabella si può anche creare inserendovi immediatamente un insieme di ennuple ottenute come risultato di una espressione SQL:

```
CREATE  TABLE Nome
    AS  EspressioneSelect;
```

1. Una regola generale del linguaggio è che per ogni comando **CREATE** *Elemento* vi è un corrispondente comando di cancellazione **DROP** *Elemento*.

Ad esempio, i seguenti comandi creano una tabella che rappresenta un archivio storico contenente informazioni sugli ordini precedenti al 2003, togliendoli dalla tabella corrente:

```
CREATE   TABLE  ArchivioStoricoOrdini
    AS   SELECT     *
         FROM       Ordini
         WHERE      Data < '01012003';

DELETE   FROM       Ordini
         WHERE      Data < '01012003';
```

Una tabella può essere eliminata con il comando:

```
DROP TABLE Nome;
```

7.1.3 Tabelle virtuali

Oltre alla costruzione di tabelle normali, contenenti dati, (*tabelle di base*), si possono definire delle *tabelle virtuali* con il comando:

```
CREATE   VIEW NomeVista [ "(" Attributo {, Attributo} ")" ]
    AS   EspressioneSelect
```

Una tabella virtuale, o *vista* (*view*), è il risultato di un'espressione SQL a partire da altre tabelle, sia base che virtuali.

Il contenuto di una tabella virtuale *non è fisicamente* memorizzato nella base di dati, ma solo l'espressione SQL che la definisce viene memorizzata nel catalogo del sistema. In generale, il contenuto viene calcolato ogni volta che la tabella virtuale viene usata come se fosse una normale tabella in un'interrogazione, tranne in quei casi in cui l'ottimizzatore del sistema sia in grado di stabilire che l'interrogazione possa essere "riscritta" combinandola con l'espressione SQL che definisce la tabella virtuale.

Ad esempio, la seguente vista:

```
CREATE   VIEW       OrdiniPerAgente(CodiceAgente, TotaleOrdini)
    AS   SELECT     CodiceAgente, SUM(Ammontare)
         FROM       Ordini
         GROUP BY   CodiceAgente;
```

descrive una tabella virtuale formata, per ogni agente che ha fatto qualche ordine, dal codice dell'agente e dal totale dei suoi ordini. Un'interrogazione come:

```
SELECT    A.Nome, TotaleOrdini
FROM      Agenti A, OrdiniPerAgente O
WHERE     A.CodiceAgente = O.CodiceAgente
          AND TotaleOrdini > 1000;
```

può essere riscritta come:

```
SELECT      A.Nome, SUM(Ammontare) AS TotaleOrdini
FROM        Agenti A, Ordini O
WHERE       A.CodiceAgente = O.CodiceAgente
GROUP BY    A.CodiceAgente, A.Nome
HAVING      SUM(Ammontare) > 1000;
```

Una tabella virtuale può comparire in una espressione **SELECT** esattamente come una tabella base, mentre le operazioni di modifica su una tabella virtuale sono soggette a restrizioni, perché non sempre sono riconducibili a modifiche sulle tabelle base usate per definirla. In particolare, le modifiche sono ammesse solo quando la tabella virtuale è definita con un'espressione che soddisfa le seguenti condizioni:

- la clausola **SELECT** non ha l'opzione **DISTINCT** e i valori degli attributi della tabella virtuale non sono calcolati;
- la clausola **FROM** riguarda una sola tabella base o virtuale, a sua volta modificabile, ovvero sono escluse tabelle virtuali ottenute per giunzione;
- la clausola **WHERE** non contiene *SottoSelect*;
- non sono presenti gli operatori **GROUP BY** e **HAVING**;
- le colonne definite nelle tabelle di base con il vincolo **NOT NULL** devono far parte della tabella virtuale.

Ad esempio, la vista precedente non può essere usata per modifiche, dato che contiene un attributo calcolato (TotaleOrdini).

Con l'introduzione delle tabelle virtuali occorre rivedere in generale il comando di cancellazione delle tabelle, sia base che virtuali. Infatti, se una tabella è utilizzata nella definizione di un'altra virtuale, al momento della sua cancellazione si può specificare quale azione intraprendere:

DROP (*Tabella* | *Vista*) [**RESTRICT** | **CASCADE**]

In entrambi i casi, se viene specificato **RESTRICT**, la tabella o la vista non vengono rimosse se sono utilizzate in altre viste, mentre **CASCADE** provoca la rimozione automatica di tutte le viste che utilizzano la tabella o la vista cancellata.

Uso delle tabelle virtuali

Le tabelle virtuali sono utili per diverse ragioni:

1. per semplificare certi tipi di interrogazioni complesse, o addirittura non esprimibili in SQL su tabelle base. Ad esempio, come già discusso nel capitolo precedente, se vogliamo trovare il numero medio degli agenti per zona, sarebbe un errore scrivere qualcosa del genere:

   ```
   SELECT      AVG(COUNT(*))
   FROM        Agenti
   GROUP BY    Zona;
   ```

   Possiamo però formulare l'interrogazione con l'aiuto di una tabella virtuale:

```
CREATE   VIEW       AgentiPerZona (Zona, NumeroAgenti)
    AS   SELECT     Zona, COUNT(*)
         FROM       Agenti
         GROUP BY   Zona;

SELECT   AVG(NumeroAgenti)
FROM     AgentiPerZona;

DROP     AgentiPerZona;
```

2. per nascondere alle applicazioni alcune modifiche dell'organizzazione logica dei dati (*indipendenza logica*). Ad esempio, supponiamo che per ogni zona vi sia un supervisore, che sia egli stesso un agente, ma privo di supervisore. La memorizzazione dei dati potrebbero essere cambiata nel modo seguente:

   a) aggiungere una tabella contenente le zone e i relativi supervisori:

```
CREATE   TABLE      SupervisoriPerZone
    AS   SELECT     DISTINCT Zona, Supervisore
         FROM       Agenti
         WHERE      Supervisore IS NOT NULL;
```

   b) togliere l'attributo Supervisore dalla tabella degli agenti:

```
CREATE   TABLE      NuoviAgenti
    AS   SELECT     CodiceAgente, Nome, Zona, Commissione
         FROM       Agenti;
```

   c) cancellare la "vecchia" tabella degli agenti:

```
DROP     Agenti;
```

   d) costruire una vista che ricostruisce la "vecchia" situazione di agenti con l'attributo Supervisore:

```
CREATE   VIEW       Agenti(CodiceAgente, Nome, Zona, Supervisore,
                           Commissione)
    AS   SELECT     CodiceAgente, Nome, Z.Zona, Supervisore , Commissione
         FROM       NuoviAgenti A, SupervisoriPerZone z
         WHERE      A.Zona = Z.Zona AND CodiceAgente <> Supervisore
           UNION ALL
         SELECT     CodiceAgente, Nome, Z.Zona, NULL AS Supervisore,
                    Commissione
         FROM       NuoviAgenti A, SupervisoriPerZone Z
         WHERE      A.Zona = Z.Zona AND CodiceAgente = Supervisore;
```

3. possono essere usate per dare visioni diverse degli *stessi* dati (viste utente). Ad esempio, se vogliamo aggregare gli ordini per agente, per applicazioni di tipo statistico, possiamo fornire la tabella virtuale creata all'inizio di questa sezione.

7.2 Vincoli d'integrità

I vincoli che possono essere espressi in uno schema relazionale riguardano i valori ammissibili degli attributi di un'ennupla, o di ennuple diverse della stessa tabella (*vincoli intrarelazionali*) o di tabelle diverse (*vincoli interrelazionali*).

Per impedirne la violazione, i vincoli d'integrità vengono controllati dal sistema quando si effettua una modifica della base di dati (operazioni **INSERT**, **DELETE** e **UPDATE**) che li coinvolge. Nel caso di violazione di un vincolo, l'azione eseguita dal sistema dipende da come è stato dichiarato il vincolo. In assenza di direttive esplicite, il sistema interrompe l'operazione annullando la transazione corrente, e segnala il fatto, come accade ad esempio quando si cerca di inserire una ennupla che viola il vincolo di chiave primaria. Nel caso invece che sia stata specificata un'azione da compiere per portare la base di dati in uno stato corretto (come è possibile fare ad esempio per il vincolo di chiave esterna), il sistema si comporta di conseguenza.

1. Vincoli su attributi di un'ennupla.
   a) Si può specificare che un attributo non possa avere il valore **NULL** con il vincolo **NOT NULL**. Questo vincolo è implicito se l'attributo fa parte della chiave primaria.
   b) Con la sintassi **CHECK** *Condizione* è possibile specificare con una condizione i valori ammissibili dell'attributo. Ad esempio, se vogliamo richiedere che l'ammontare minimo di un ordine sia 100, si pone

      Ammontare **INTEGER NOT NULL CHECK** (Ammontare >= 100)

      oppure se vogliamo imporre che un attributo Sesso abbia solo valori M o F, si pone

      Sesso **CHAR**(1) **NOT NULL CHECK** (Sesso **IN** ('M', 'F'))

   c) Si può fare in modo che il sistema assegni un valore di default all'attributo quando viene inserita una ennupla con **DEFAULT** (*Costante* | **NULL**).
   d) È possibile definire vincoli fra i valori di attributi diversi di una stessa ennupla; nella condizione non possono essere coinvolte altre tabelle:

      CHECK Condizione

2. Vincoli intrarelazionali.
   a) Il vincolo **UNIQUE** richiede che non vi siano duplicati nei valori dell'attributo (cioè l'attributo è una chiave). Si noti che la chiave primaria *deve* invece essere dichiarata con il seguente vincolo di **PRIMARY KEY**.

b) È possibile definire una chiave primaria, anche formata da più attributi, assegnandole un nome:

PRIMARY KEY [*NomeChiave*] "(" *Attributo* {, *Attributo*} ")"

Gli attributi delle chiavi devono essere dichiarati **NOT NULL**, e non vi può essere più di un vincolo di tipo **PRIMARY KEY** in una tabella. Una chiave primaria è obbligatoria quando si devono introdurre vincoli d'integrità referenziali.

c) È possibile definire chiavi formate da più attributi:

UNIQUE "(" *Attributo* {, *Attributo*} ")"

3. Vincoli interrelazionali. È possibile definire il vincolo d'integrità referenziale su chiavi esterne, con la seguente sintassi:

FOREIGN KEY [*NomeChiaveEsterna*] "(" *Attributo* {, *Attributo*} ")"
REFERENCES *TabellaReferenziata*
[**ON DELETE** (**NO ACTION** | **CASCADE** | **SET NULL**)]

dove *TabellaReferenziata* è una tabella per la quale è stata definita una chiave primaria (con l'opzione **PRIMARY KEY**) il cui tipo è uguale a quello degli attributi specificati.
Gli attributi di una chiave esterna, al contrario di quelli di una chiave, possono avere il valore **NULL**.
Il vincolo d'integrità referenziale su chiavi esterne ha i seguenti effetti:

- se si cerca di inserire una ennupla nella tabella con il valore della chiave esterna che non corrisponde ad un valore della chiave primaria in *TabellaReferenziata,* il vincolo viene violato e l'operazione fallisce;
- se si cerca di cancellare una ennupla di *TabellaReferenziata* la cui chiave primaria è il valore di qualche chiave esterna nella tabella (violando il vincolo), si opera in base alla specifica del vincolo:
  a) **ON DELETE NO ACTION** richiede il rifiuto dell'operazione e il fallimento della transazione. L'assenza della specifica **ON DELETE** è equivalente a questa opzione;
  b) **ON DELETE CASCADE** richiede la cancellazione delle ennuple che hanno il valore della chiave esterna uguale a quello della chiave primaria delle ennuple cancellate;
  c) **ON DELETE SET NULL** richiede di assegnare il valore nullo agli attributi della chiave esterna.
- se si cerca di modificare una ennupla assegnando agli attributi della chiave esterna un valore che non corrisponde ad un valore della chiave primaria in *TabellaReferenziata,* oppure se si modifica il valore di una chiave primaria in *TabellaReferenziata,* di solito nei sistemi commerciali l'operazione viene rifiutata, mentre nei sistemi che seguono il livello intermedio dell'SQL-92, con l'opzione

ON UPDATE è prevista la possibilità di scelta dell'azione da intraprendere come nel caso delle cancellazioni.

Esempio 7.1 Diamo lo schema SQL completo per la base di dati di Figura 6.1:

```
CREATE   TABLE Clienti (
         CodiceCliente   CHAR(3)      NOT NULL,
         Nome            CHAR(30)     NOT NULL,
         Città           CHAR(30)     NOT NULL,
         Sconto          INTEGER      NOT NULL DEFAULT 0
                         CHECK(Sconto>=0 AND Sconto<100),
         PRIMARY KEY     pk_Clienti (CodiceCliente) );

CREATE   TABLE Agenti (
         CodiceAgente    CHAR(3)      NOT NULL,
         Nome            CHAR(30)     NOT NULL,
         Zona            CHAR(8)      NOT NULL,
         Supervisore     CHAR(3),
         Commissione     INTEGER,
         PRIMARY KEY     pk_Agenti (CodiceAgente),
         FOREIGN KEY     fk_SupervisoreAgente (Supervisore)
                         REFERENCES Agenti
                         ON DELETE SET NULL,
         CHECK           (Supervisore <> CodiceAgente
                          OR Supervisore IS NULL) );

CREATE   TABLE Ordini (
         NumOrdine       CHAR(3)      NOT NULL,
         CodiceCliente   CHAR(3)      NOT NULL,
         CodiceAgente    CHAR(3)      NOT NULL,
         Data            CHAR(8)      NOT NULL,
         Prodotto        CHAR(3)      NOT NULL,
         Ammontare       INTEGER      NOT NULL CHECK(Ammontare > 100),
         PRIMARY KEY     pk_Ordini (NumOrdine),
         FOREIGN KEY     fk_ClienteOrdine (CodiceCliente)
                         REFERENCES Clienti
                         ON DELETE NO ACTION,
         FOREIGN KEY     fk_AgenteOrdine (CodiceAgente)
                         REFERENCES Agenti
                         ON DELETE NO ACTION);
```

```
CREATE   VIEW       OrdiniPerAgente(CodiceAgente, TotaleOrdini)
     AS  SELECT     CodiceAgente, SUM(Ammontare)
         FROM       Ordini
         GROUP BY   CodiceAgente;

CREATE   VIEW       AgentiConOrdini
     AS  SELECT     A.CodiceAgente AS CodiceAgente, Nome, Zona,
                    Supervisore, Commissione, TotaleOrdini
         FROM       OrdiniPerAgente O, Agenti A
         WHERE      A.CodiceAgente = O.CodiceAgente;
```

Si noti la definizione dei vincoli e delle tabelle virtuali. Nella seconda, AgentiConOrdini, si utilizza la prima (OrdiniPerAgente), e non si dichiarano gli attributi della tabella, perché vengono presi quelli specificati nel **SELECT**.

7.3 Aspetti procedurali

Nei sistemi relazionali commerciali si possono trattare nello schema aspetti procedurali definendo *procedure* o *trigger*.

Le procedure memorizzate nella base di dati sono programmi che vengono eseguiti dal DBMS su esplicita richiesta delle applicazioni o degli utenti. I *trigger*, invece, sono anch'essi delle procedure memorizzate nella base di dati, che però vengono attivate automaticamente dal DBMS quando si fanno determinate operazioni sulle tabelle.

7.3.1 Procedure memorizzate

Le procedure memorizzate nella base di dati sono state previste dall'SQL-92 come procedure con un nome, parametri e un corpo costituito da un unico comando SQL, raggruppabili attraverso un meccanismo di *moduli*. Normalmente, però, i sistemi commerciali prevedono un linguaggio più ricco per definirle, come il linguaggio PL/SQL del sistema Oracle, che verrà presentato nel prossimo capitolo, insieme a esempi di procedure, e il Transact/SQL del sistema Sybase.

Le procedure memorizzate nella base di dati sono utili per diverse ragioni:

1. Consentono di condividere fra le applicazioni del codice di interesse generale. In questo modo si semplificano le applicazioni e quindi la loro manutenzione. Inoltre, essendo le procedure gestite in modo centralizzato, una loro modifica non richiede di essere riportata in tutte le applicazioni che ne fanno uso.
2. Consentono di garantire che certe operazioni sulla base di dati abbiano la stessa semantica per ogni applicazione che ne fa uso.
3. Consentono di controllare in modo centralizzato certi vincoli d'integrità non esprimibili nella definizione delle tabelle.

4. Consentono di ridurre il traffico sulla rete dovuto ad applicazioni remote: il programma cliente, invece di interagire con il DBMS servente spedendo un'interrogazione per volta, spedisce semplicemente una chiamata di una procedura memorizzata, che viene eseguita dal servente, e riceve alla fine solo il risultato finale.
5. Consentono di garantire la sicurezza dei dati perché a certi utenti si può permettere di usare solo certe procedure per ottenere dei dati, ma non di accedere direttamente alle tabelle dalle quali le procedure estraggono i dati restituiti.

L'uso delle procedure memorizzate nella base di dati pone però nuovi problemi nella progettazione di basi di dati, e le metodologie disponibili non aiutano in genere a definire ed organizzare tali procedure. Il loro uso pone, inoltre, nuovi problemi di carattere organizzativo poiché richiede una nuova figura professionale, l'amministratore delle procedure, che può essere diverso dal tradizionale amministratore dei dati.

7.3.2 Trigger

I *trigger* sono presenti in tutti i sistemi commerciali più sofisticati, sebbene non siano stati previsti dallo standard SQL-92.

Un *trigger* specifica un'azione da attivare automaticamente al verificarsi di un'operazione di modifica su una tabella (**INSERT**, **UPDATE**, **DELETE**).

Come esempio, riportiamo la forma base del comando per creare *trigger* in PL/SQL:

CREATE **TRIGGER**, *NomeTrigger*
TipoTrigger
(*TipoOperazione* {**OR** *TipoOperazione*})
[**OF** *Attributo*] **ON** *NomeTabella*
[**FOR EACH ROW**]
[**WHEN** "(" *Condizione* ")"]
Programma

TipoTrigger := (**BEFORE** | **AFTER**)
TipoOperazione := (**SELECT** | **DELETE** | **INSERT** | **UPDATE**)

Nella definizione si specificano le seguenti informazioni:

- il tipo di *trigger* (**BEFORE**, **AFTER**) per stabilire quando il *trigger* debba essere attivato, se prima o dopo l'operazione;
- il tipo di operazione che attiva il *trigger* e su quale tabella (ed eventualmente su quale attributo);
- un'eventuale clausola **FOR EACH ROW** per stabilire quante volte il *trigger* debba essere attivato, se una volta oppure tante volte quante sono le righe della tabella interessate dall'operazione;
- un'eventuale ulteriore condizione che deve essere vera perché il codice del *trigger* venga eseguito;
- il codice da eseguire.

Esempio 7.2 Supponiamo che nella base di dati dell'Esempio 7.1 gli ordini siano fatture da pagare e che si voglia imporre il vincolo che non si accettano nuovi ordini da clienti con uno scoperto maggiore di 10.000:

```
CREATE TRIGGER ControlloFido
BEFORE INSERT ON Ordini
DECLARE
    DaPagare NUMBER;
BEGIN
    SELECT SUM(Ammontare) INTO DaPagare
    FROM   Ordini
    WHERE  CodiceCliente = :new.CodiceCliente;
    IF DaPagare >= 10000 − :new.Ammontare
    THEN
        RAISE_APPLICATION_ERROR(-2061, 'fido superato');
    END IF;
END;
```

Nel corpo di un *trigger*, :new sta per il valore della ennupla da inserire o il valore modificato, mentre :old per il valore precedente. Quindi, in questo caso :new.CodiceCliente è il valore del campo CodiceCliente dell'ennupla da inserire.

Come ulteriore esempio, mostriamo l'uso di un *trigger* per mantenere una tabella memorizzata, ma aggiornata automaticamente in funzione di un'altra tabella, senza l'uso di viste.

Esempio 7.3 Il seguente programma crea una nuova tabella di coppie (agente, totale ordini effettuati), e un *trigger* che automaticamente, da questo momento in poi, manterrà aggiornata la nuova tabella.

```
CREATE TABLE Totali(CodiceAgente CHAR(3), TotaleOrdini INTEGER);

CREATE TRIGGER esempioTrig
AFTER INSERT ON Ordini
FOR EACH ROW
DECLARE
    esiste NUMBER;
BEGIN
    /* Si controlla se l'agente dell'ordine e' gia' presente
        nella tabella dei totali */
    SELECT COUNT(∗) INTO esiste
    FROM   Totali
    WHERE  CodiceAgente = :new.CodiceAgente;
```

```
    IF esiste = 0 /* L'agente non e' presente, deve essere inserita una nuova riga */
    THEN
        INSERT INTO Totali
        VALUES(:new.CodiceAgente, :new.Ammontare);
    ELSE
        UPDATE Totali
        SET TotaleOrdini = TotaleOrdini + :new.Ammontare
        WHERE CodiceAgente = :new.CodiceAgente;
END;
```

Si noti che :new.CodiceAgente rappresenta il valore del campo CodiceAgente della riga di Ordini inserita.

Se vogliamo anche cancellare la riga della tabella Totali quando l'agente corrispondente viene cancellato, è sufficiente il seguente *trigger*:

```
CREATE TRIGGER esempio2Trig
AFTER DELETE ON Agenti
FOR EACH ROW BEGIN
    DELETE Totali WHERE CodiceAgente = :old.CodiceAgente;
END;
```

Uso dei trigger

A seconda del loro uso, possiamo dividere i *trigger* in *passivi* ed *attivi*. Un *trigger* è passivo se serve a provocare un fallimento della transazione corrente sotto certe condizioni, mentre un *trigger* è attivo quando, in corrispondenza di certi eventi, modifica lo stato della base di dati (così, negli esempi precedenti, il primo è passivo mentre gli ultimi due sono attivi).

I *trigger* sono utili per diverse ragioni:

1. *Trigger* passivi:
   a) per definire vincoli d'integrità non esprimibili nel modello dei dati usati (ad esempio vincoli dinamici);
   b) per fare controlli sulle operazioni ammissibili degli utenti basati sui valori dei parametri di comandi SQL. Ad esempio, si possono inserire certi dati solo se il codice del dipartimento è quello dell'utente che esegue l'operazione.
2. *Trigger* attivi:
   a) per definire le cosiddette *regole degli affari* (*business rules*), ovvero le azioni da eseguire per garantire la corretta evoluzione del sistema informativo. Ad esempio, le azioni per la manutenzione automatica del magazzino, spedizione automatica di ordini e solleciti, passaggio di documenti fra utenti diversi ecc.;

b) per memorizzare eventi sulla base di dati per ragioni di controllo (*auditing and logging*). Ad esempio, in un'opportuna tabella si registrano dati sulle operazioni eseguite su tabelle riservate. In verità si memorizzano solo gli effetti di transazioni terminate normalmente perché di solito se una transazione termina prematuramente anche gli effetti dei *trigger* vengono annullati;
c) per propagare su altre tabelle gli effetti di certe operazioni su tabelle (base o calcolate);
d) per mantenere allineati eventuali dati duplicati quando si modifica uno di essi;
e) per mantenere certi vincoli d'integrità modificando la base di dati in modo opportuno; ad esempio, quando una ennupla viene cancellata, il vincolo referenziale può essere mantenuto in maniera attiva cancellando tutte le ennuple che fanno riferimento alla ennupla cancellata.

Vantaggi e problemi nell'uso dei trigger

Poiché i *trigger* sono memorizzati nella base di dati e la loro attivazione è sotto il controllo del DBMS, si hanno i seguenti vantaggi:

1. si semplifica la codifica delle applicazioni che non devono preoccuparsi dei controlli effettuati dai *trigger*;
2. i *trigger* sono definiti e amministrati centralmente e quindi gli utenti che usano la base di dati, in modo interattivo o per mezzo di applicazione, non possono evitare i controlli da essi garantiti.

Un problema che si presenta nell'uso dei *trigger* è che i sistemi commerciali adottano una diversa semantica per la loro attivazione e prevedono una diversa modalità di interazione fra i meccanismi di attivazione dei *trigger* e l'esecuzione della transazione che li attiva [Fraternali and Tanca, 1995]. In particolare, i punti principali che differenziano i vari sistemi sono:

– *Granularità*. Se la modifica riguarda un insieme di ennuple (come è possibile per i comandi **UPDATE**, **INSERT** e **DELETE**), in alcuni sistemi il *trigger* viene eseguito una sola volta per il comando (*trigger di comando*), mentre in altri viene eseguito tante volte quante sono le ennuple modificate dal comando (*trigger di riga*). Ad esempio, in Oracle è possibile scegliere fra i due casi: con la specifica esplicita **FOR EACH ROW** si dichiara che il *trigger* deve essere attivato tante volte quante sono le righe modificate. Un approccio diverso è adottato da Sybase, dove un *trigger* è attivato *una* sola volta per comando. In questo caso si possono usare le tabelle speciali inserted e deleted che contengono le righe inserite o cancellate dal comando (una modifica è considerata una cancellazione seguita da un'inserzione). L'esempio precedente diventerebbe così:

```
CREATE TRIGGER esempio3Trig
AFTER DELETE ON Agenti
BEGIN
    DELETE Totali
    WHERE CodiceAgente IN
            (SELECT deleted.CodiceAgente FROM deleted);
END;
```

- *Risoluzione dei conflitti*. Se è possibile definire più *trigger* attivabili dallo stesso comando, in alcuni sistemi l'ordine di definizione dei *trigger* stabilisce anche l'ordine in cui vengono attivati (Oracle), in altri sistemi si può specificare in quale ordine vanno attivati, in altri ancora l'ordine è stabilito dal sistema.
- *Trigger in cascata*. Se un *trigger* T_1 provoca l'attivazione di un altro T_2 (*trigger* in "cascata", *cascade trigger*, o annidati, *nested trigger*), si possono creare dei cicli se T_2 provoca l'attivazione di T_1 (*trigger* ricorsivi). In Sybase è possibile scegliere se permettere o meno le attivazioni in cascata (e in questo caso se permettere attivazioni ricorsive), mentre in Oracle le attivazioni ricorsive sono proibite.
- *Esecuzione dei trigger*. Di solito l'azione di un *trigger* viene eseguita immediatamente come parte della transazione che lo ha attivato. Un'altra possibilità da prevedere sarebbe di poter specificare che l'azione di un *trigger* andrebbe eseguita solo alla fine della transazione che lo attiva (esecuzione differita). Questa possibilità sarebbe utile nel seguente caso:

  1. supponiamo che esista un *trigger* T che elimina un dipartimento quando questo non ha un direttore;
  2. viene eseguita una transazione che elimina un impiegato direttore di un dipartimento, al quale assegna poi un nuovo direttore.

  Con l'esecuzione immediata di T il dipartimento viene eliminato, mentre con l'esecuzione differita di T la transazione produce l'effetto voluto.
- *Interazione con le transazioni*. Un altro aspetto critico è l'interazione dell'esecuzione dell'azione di un *trigger* con l'esecuzione della transazione che lo attiva. Di solito l'azione diventa parte della transazione che viene eseguita globalmente in modo atomico: se l'azione abortisce, abortisce anche la transazione e viceversa.

  In generale questo modo di procedere è quello desiderato, ma esistono casi in cui non lo è. Un esempio è quando un *trigger* viene usato per memorizzare eventi sulla base per ragioni di controllo, visto in precedenza: se una transazione legge un dato questo evento va registrato in un'opportuna tabella, anche nel caso in cui la transazione fallisca.

  Un'altra possibilità sarebbe di poter specificare che l'azione di un *trigger* vada eseguita come una transazione indipendente da quella che lo attiva.

Oltre a queste difficoltà semantiche, i *trigger* presentano problemi per ciò che riguarda la loro progettazione e la realizzazione di applicazioni in un ambiente in cui se ne faccia uso.

Per ciò che riguarda la progettazione dei *trigger*, il problema risiede nel fatto che molte metodologie, tra cui quella descritta nei primi capitoli di questo testo, non forniscono indicazioni riguardo all'utilizzo dei *trigger*, analogamente a quanto è già stato osservato a proposito delle procedure memorizzate. Tuttavia, una corretta progettazione dei *trigger* è cruciale per evitare poi problemi nella fase di realizzazione delle applicazioni.

Per ciò che riguarda la realizzazione delle applicazioni in un ambiente in cui sui dati siano definiti dei *trigger*, possiamo citare in particolare due problemi:

1. Complessità: chi realizza un'applicazione, per conoscerne gli effetti, deve capire non solo in che modo l'applicazione modifica la base di dati in modo diretto, ma anche qual è l'effetto dei *trigger* attivati da tale applicazione. Questo aumenta in generale la complessità della progettazione delle applicazioni. La situazione è ancora più complessa quando si utilizzano *trigger* attivi, poiché questi possono provocare l'attivazione indiretta di ulteriori *trigger*. In questa situazione, il comportamento del sistema tende a diventare imprevedibile, principalmente perché è difficile capire in che ordine e in che esatto momento i diversi *trigger* vengono effettivamente eseguiti. Trattandosi però di *trigger* che modificano lo stato, il tempo di esecuzione ne influenza in generale la semantica, che tende quindi ad uscire dal controllo del programmatore. A questo proposito, molti sistemi definiscono dei meccanismi automatici per impedire l'attivazione mutuamente ricorsiva di più *trigger*, a dimostrazione di quanto siano comuni situazioni in cui il comportamento di un insieme di *trigger* travalica le intenzioni di chi li ha disegnati.
2. Rigidità: un *trigger*, in genere, costringe al rispetto di un certo vincolo adottando una certa strategia; ad esempio, potrebbe forzare il vincolo che ogni dipartimento ha un direttore cancellando i dipartimenti che ne sono privi. Se una specifica applicazione intende mantenere lo stesso vincolo con una strategia diversa, questa applicazione può finire in conflitto con il *trigger*, come esemplificato in precedenza nel caso di una applicazione che vorrebbe cancellare il vecchio direttore e fissarne uno nuovo. In questo caso l'applicazione è costretta a cercare un modo per convivere con il *trigger*, poiché non è possibile, in generale, evitarne l'attivazione.

Concludiamo questa sezione accennando alle *basi di dati attive*: sono basi di dati in cui l'uso dei trigger è notevolmente ampliato al fine di definire delle regole (dette regole ECA, *evento-condizione-azione*), in cui gli eventi sono molto più ampi di quelli classici (ad esempio eventi temporali, eventi su condizioni qualunque, eventi composti, eventi esterni ecc.), e i meccanismi di valutazione delle regole sono più sofisticati. Queste regole vengono usate in maniera estensiva per arricchire gli aspetti funzionali delle basi di dati gestite, rendendole utilizzabili per lo sviluppo di applicazioni più sofisticate di quelle normalmente disponibili. Per approfondire l'argomento, e per esempi di DBMS attivi, si rinvia ai riferimenti riportati nelle note bibliografiche.

7.4 Progettazione fisica

La progettazione fisica di una base di dati è molto critica perché comporta numerose decisioni che devono tener conto delle caratteristiche del DBMS e del tipo di uso che le applicazioni fanno della base di dati. Per il carattere non specialistico del testo, ci limiteremo a ricordare che le principali informazioni che vanno fornite dalla progettazione fisica di una base di dati relazionale sono:

- distribuzione delle tabelle sugli archivi del sistema operativo; è necessario stabilire quali archivi contengano le tabelle, fornendo in particolare la possibilità di spezzare una singola tabella su archivi diversi, e di mantenere più tabelle in un singolo archivio;
- dimensioni degli archivi e loro organizzazione; l'organizzazione di un archivio determina, ad esempio, se i record sono inseriti nella tabella in ordine di arrivo, oppure ordinate sul valore di un attributo, oppure in base al risultato dell'applicazione di una funzione *hash* al valore di un attributo. Le organizzazioni dei dati verranno presentate nel Capitolo 9;
- presenza di indici; un indice su di un attributo A di una tabella è una struttura dati che permette di trovare rapidamente una registrazione nella tabella a partire dal suo valore per l'attributo A. L'importanza degli indici nell'esecuzione delle interrogazioni verrà mostrato nel Capitolo 9.

Come per gli aspetti procedurali, le modalità di definizione degli aspetti fisici di una base di dati non sono standardizzate. Vedremo quindi, a titolo di esempio, quelle previste in Oracle per la definizione degli indici.

7.4.1 Definizione di indici

Un indice I su un attributo A di una tabella R, dal punto di vista logico, è una tabella con due attributi $I(A, \text{TID})$, con gli elementi ordinati su A e valori $(A := a_i, \text{TID} := r_j)$, dove a_i è un valore di A presente in un'ennupla di R, ed r_j è un riferimento (TID) all'ennupla di R con il valore a_i di A.. Se A non è una chiave, nell'indice sono presenti tante ennuple con lo stesso valore a_i di A quante sono le ennuple di R con il valore a_i di A.

Un indice può essere anche definito su un insieme di attributi, e in questo caso il primo elemento della coppia è formato da una combinazione dei valori relativi. Di solito una tabella è memorizzata in modo seriale, mentre un indice è memorizzato con una struttura ad albero per trovare con pochi accessi, a partire da un valore v, le ennuple di R in cui il valore di A è in una relazione specificata con v.

L'esistenza degli indici non cambia il modo in cui si formulano le interrogazioni (*indipendenza fisica*), ma viene sfruttata dall'ottimizzatore del sistema per stabilire le strategie di esecuzione delle interrogazioni, in particolare se e quali indici usare.

La creazione e distruzione di indici può essere fatta in ogni momento su tabelle già esistenti: questa operazione può invalidare i risultati di ottimizzazioni precedenti.

La forma base del comando per creare indici è:

CREATE [**UNIQUE**] **INDEX** *NomeIndice*
ON *NomeTabella*
"(" *Attributo* [**ASC** | **DESC**] {, *Attributo* [**ASC** | **DESC**] } ")"
[**TABLESPACE** *NomeArchivio*]
[**STORAGE** *ParametriDiMemoria*]

L'opzione **UNIQUE** è usata per specificare che l'indice riguarda un attributo chiave, **TABLESPACE** richiede la memorizzazione dell'indice nell'archivio specificato, e **STORAGE** modifica i parametri di allocazione fisica default dell'archivio. Le opzioni **ASC** e **DESC** sono usate per scegliere l'ordinamento per valori crescenti o decrescenti di un attributo, per default crescente.

Un indice può essere eliminato con il comando **DROP INDEX** *NomeIndice*.

I sistemi costruiscono automaticamente un indice sugli attributi delle chiavi, per controllare facilmente che i loro valori siano diversi nella tabella.

Come scegliere gli indici

In generale non è semplice stabilire quali indici creare su una base di dati per migliorare le prestazioni complessive delle applicazioni. Infatti, se è vero che gli indici favoriscono le operazioni di ricerca, è anche vero che occupano memoria (di solito si stima che un indice occupa il 20% della memoria occupata dalla relazione), e vanno aggiornati ad ogni operazione **INSERT**, **UPDATE** e **DELETE**, aumentando i loro tempi di esecuzione. Altro aspetto che complica il problema è che gli indici vanno scelti conoscendo le strategie di ottimizzazione del sistema per evitare di costruire indici che non verranno mai usati.

Sono stati studiati algoritmi opportuni per scegliere gli indici ed alcuni sistemi forniscono strumenti per aiutare il progettista nel farlo [Albano, 2001]. Si tengano comunque presenti i seguenti suggerimenti almeno per evitare scelte errate:

1. Non creare indici su tabelle piccole, ovvero che occupano meno di sei pagine.
2. Non creare indici su attributi poco selettivi, ovvero che hanno pochi valori diversi come il sesso o lo stato civile.
3. Evitare indici su attributi modificati frequentemente.
4. Prevedere più di quattro indici per relazione solo se le operazioni di modifica sono rare.
5. Creare indici sulle chiavi esterne per agevolare l'esecuzione delle operazioni di giunzione.
6. Sono utili indici ordinati su attributi usati frequentemente nelle opzioni **ORDER BY**, **DISTINCT** e **GROUP BY**. Infatti, l'esecuzione di interrogazioni con queste opzioni, in assenza di indici, comporta la creazione di tabelle temporanee da ordinare.
7. Prevedere indici su attributi usati frequentemente nelle operazioni di restrizione con condizione di uguaglianza o comunque molto restrittive.

7.5 Evoluzione dello schema

Una caratteristica dei sistemi relazionali, assente nei sistemi che li hanno preceduti, è la possibilità di modificare la struttura delle tabelle anche dopo che vi sono stati inseriti dei dati.

In particolare, una tabella può essere modificata con il comando **ALTER TABLE** per:

1. aggiungere un nuovo attributo:

   ALTER TABLE *Nome* **ADD** *Attributo Tipo*

   I valori della nuova colonna saranno tutti nulli (e non sarà possibile imporre il vincolo **NOT NULL** prima di assegnare nuovi valori a tutta la colonna);
2. eliminare un attributo:

   ALTER TABLE *Nome* **DROP** *Attributo*
3. modificare la definizione di un attributo:

   ALTER TABLE *Nome* **MODIFY** *Attributo Tipo*

   Le modifiche di tipo sono soggette a certe restrizioni di compatibilità. Ad esempio, è possibile passare da **INTEGER** a **FLOAT**, o da **CHAR**(4) a **CHAR**(6), ma non da **CHAR**(4) a **INTEGER**;
4. cambiare il nome di un attributo:

   ALTER TABLE*Nome* **RENAME** *VecchioAttributo NuovoAttributo*
5. aggiungere o eliminare un vincolo d'integrità:

   ALTER TABLE *Nome Vincolo*
   ALTER TABLE *Nome* **DROP** *Vincolo*

Ad esempio, se dopo aver creato la tabella Ordini si vuole aggiungere una colonna che registra i pagamenti effettuati, assumendo che gli ordini già fatti siano stati tutti pagati, si può scrivere:

```
ALTER TABLE Ordini ADD Pagato INTEGER;

UPDATE Ordini
   SET Pagato = Ammontare
```

7.6 Utenti e Autorizzazioni

Per garantire la protezione dei dati da accessi non desiderati, i DBMS forniscono normalmente un sistema di permessi basato sui concetti di *utente*, *autorizzazione* (o *privilegio*) e *profilo*.

Gli utenti sono creati dall'amministratore della base di dati, e da questi possono ricevere le autorizzazioni di base, che gli permettono di iniziare a lavorare (collegarsi,

creare schemi, tabelle, ecc.). Se un utente crea un oggetto, come una tabella, a sua volta può autorizzare altri utenti a lavorare su di esso.

Un'autorizzazione è il permesso di eseguire una o più operazioni, e si indica con il nome dell'operazione o con un nome simbolico (ad esempio **SELECT** o **CONNECT**). Un utente ha associato un insieme di autorizzazioni, che delimitano le operazioni che può effettuare (e i dati su cui può effettuarle).

Per semplificare la gestione dell'assegnazione delle autorizzazioni, sono definiti i profili, cioè insiemi di autorizzazioni che possono essere assegnati ad un utente in maniera globale, e che definiscono le categorie tipiche di utenti di una certa base di dati.

Un utente viene creato con il seguente comando:

CREATE USER *NomeUtente* **PROFILE** *NomeProfilo*
IDENTIFIED BY *Password*
DEFAULT TABLESPACE *NomeArchivio*
TEMPORARY TABLESPACE *NomeArchivio*
ACCOUNT UNLOCK

che crea un utente assegnandogli un profilo, una password e degli spazi di lavoro. Il profilo conterrà tipicamente almeno le autorizzazioni di base (ad esempio **CONNECT**, per collegarsi alla base di dati, o **RESOURCE** per costruire tabelle e altri oggetti di base).

Oltre alle autorizzazioni associate ad un profilo, il proprietario di una risorsa può assegnare e ritirare singole autorizzazioni sulla risorsa con i seguenti comandi, dei quali si mostrano solo alcune possibilità, riferite a tabelle base e tabelle virtuali:

GRANT *Autorizzazioni*
ON *Tabella*
TO (**PUBLIC** | *Utente* {, *Utente* })
[**WITH GRANT OPTION**]

REVOKE [**GRANT OPTION FOR**] *Autorizzazioni*
ON *Tabella*
FROM *Utente* {, *Utente* }

dove le autorizzazioni possibili sono le seguenti:

- **SELECT** per consentire l'accesso a tutte le colonne della tabella.
- **INSERT** per consentire l'inserzione di ennuple nella tabella.
- **UPDATE** per consentire la modifica dei campi delle ennuple della tabella. La forma ristretta **UPDATE**(*Attributi*) autorizza la modifica solo dei campi specificati.
- **DELETE** per consentire la cancellazione di ennuple dalla tabella.
- **ALL PRIVILEGES** per consentire tutte le operazioni.

Con il comando **GRANT** si autorizza l'uso di una tabella a tutti (**PUBLIC**) o solo ad alcuni utenti. Se viene specificato **WITH GRANT OPTION** gli utenti hanno a loro volta il diritto di concedere le stesse autorizzazioni ad altri utenti ancora (di nuovo con la clausola **WITH GRANT OPTION**).

Con il comando **REVOKE** si tolgono delle autorizzazioni a certi utenti, oppure, semplicemente, il diritto che hanno a trasferirle ad altri utenti (**REVOKE GRANT OPTION FOR**). La rimozione di un'autorizzazione (o del diritto a trasmetterla) ad un certo utente provoca in maniera automatica la rimozione della stessa autorizzazione a tutti gli altri utenti che l'avevano ricevuta da questo.

Il meccanismo delle autorizzazioni, insieme alle tabelle virtuali, offre una grande flessibilità nel controllo dell'accesso e della modifica dei dati. Ad esempio, per accedere ad una tabella virtuale V che usa nella propria definizione una tabella R è sufficiente avere i diritti relativi a V. In questo modo è possibile fornire ad un utente la possibilità di vedere o modificare solo quella parte di una tabella reale che è riflessa nella tabella virtuale, senza concedergli alcun diritto sulla tabella reale.

7.7 Schemi esterni

Uno schema esterno, secondo la proposta ANSI/X3/SPARC citata nel Capitolo 1, è la definizione della struttura della base di dati per una certa classe di utenti e della modalità d'uso dei dati ad essi consentita.

I DBMS relazionali offrono soluzioni diverse per definire schemi esterni.

1. Una base di dati è descritta da un unico schema S e con il meccanismo delle autorizzazioni si dichiara chi può accedere alle tabelle base o virtuali presenti nello schema e con quali modalità. Esiste, quindi, un comando **CREATE SCHEMA**, ma non un comando **CREATE EXTERNAL SCHEMA**.
2. Si possono definire più schemi S_i che usano tabelle base o virtuali presenti nello schema S che descrive la base di dati. Per ogni schema S_i si autorizzano gli utenti con le possibilità previste nel caso precedente. Con questa soluzione ogni schema S_i ha il ruolo di schema esterno come previsto dalla proposta ANSI/X3/SPARC.

In entrambi i casi, le tabelle virtuali sono il meccanismo fondamentale per modificare la visione della struttura dei dati memorizzati e per garantire l'indipendenza logica delle applicazioni.

7.8 Cataloghi

I sistemi relazionali prevedono che le informazioni relative allo schema (tabelle, viste, vincoli, *trigger*, utenti, autorizzazioni, indici ecc.), i cosiddetti *metadati*, vengano memorizzati in opportune tabelle interrogabili da utente, dette *catalogo del sistema*.

Vediamo alcuni attributi di un campione di possibili tabelle del catalogo per raccogliere informazioni su utenti, basi di dati, tabelle, colonne e indici:

```
PASSWORD(NomeUtente, ParolaChiave)
SYSDB(NomeBaseDati, Proprietario, Cammino, Commenti)
SYSTABLE(NomeTabella, Proprietario, BaseODerivata, NumeroColonne,
        NomeArchivioFisico, Commenti)
```

SYSCOLS(NomeColonna, Tabella, Numero, Tipo, Lunghezza, Default, Commenti)
SYSINDEX(NomeIndice, Tabella, Proprietario, NumeroColonna, Commenti)

Altre tabelle, una decina, contengono informazioni sui vincoli, le tabelle virtuali, le autorizzazioni ecc.

Altre informazioni raccolte nei cataloghi di sistema riguardano aspetti quantitativi sui dati, le *statistiche*, e sono utilizzate dall'ottimizzatore delle interrogazioni.

Normalmente le tabelle del catalogo sono consultabili dall'utente, ma non modificabili per impedire interferenze con il funzionamento del sistema. Spesso queste tabelle sono delle "viste" di tabelle più complesse o vengono ricopiate dal sistema in strutture dati in memoria temporanea e ottimizzate per la valutazione delle interrogazioni.

Un utilizzo importante dei metadati è la loro consultazione interattiva da parte degli utenti (o dell'amministratore) usando l'SQL. Per ragioni di completezza i metadati saranno autoreferenziati: ad esempio, la tabella SYSTABLE conterrà anche una riga relativa a se stessa. Quindi la struttura dei metadati stessi può essere consultata (naturalmente conoscendo precedentemente un insieme minimo di informazioni essenziali).

7.9 Strumenti per l'amministrazione di basi di dati

Oltre al catalogo dei dati, molti altri strumenti sono offerti dai produttori di DBMS o da ditte indipendenti per assistere l'amministratore di basi di dati nelle numerose attività di sua competenza. Vediamone alcuni per le attività considerate più critiche:

1. progettazione concettuale di basi di dati e generazione dei comandi per la definizione dei relativi schemi relazionali;
2. definizione della memoria da assegnare alle tabelle e agli indici e sua riorganizzazione in caso di eccessiva frammentazione;
3. controllo dell'esecuzione di comandi SQL per migliorare le prestazioni delle applicazioni critiche;
4. pianificazione ed esecuzione delle procedure per la generazione di copie di sicurezza della base di dati;
5. controllo del funzionamento del DBMS e generazione di opportune statistiche su utilizzazione della memoria permanente e del buffer, operazioni di I/O, blocchi dei dati e condizioni di stallo delle transazioni attive ecc.

7.10 Conclusioni

Sono state presentate le caratteristiche dell'SQL per la definizione e amministrazione di basi di dati.

Il linguaggio messo a disposizione a questo scopo dai vari sistemi è in realtà, in genere, molto più complesso del sottoinsieme qui illustrato, ed è ricco di opzioni che differiscono in modo sostanziale da un sistema ad un altro. Questo non dipende tanto

dalla povertà dello standard, quanto dal fatto che le attività di cui si occupa l'amministratore della base di dati, ovvero la definizione dello schema fisico, la gestione degli schemi, degli utenti, delle autorizzazioni e dell'affidabilità, coinvolgono quegli aspetti fisici sui quali i vari sistemi differiscono in modo molto sensibile.

Esercizi

1. Si definisca uno schema relazionale con i comandi **CREATE TABLE** per trattare le informazioni e i vincoli d'integrità sui dipendenti di un'azienda, con attributi CodiceFiscale, Nome, AnnoAssunzione e Salario, e sui loro familiari a carico, con attributi Nome, AnnoNascita e RelazioneDiParentela.
2. Si supponga che sia stato definito uno schema relazionale con le istruzioni:

   CREATE TABLE R (K **CHAR**(8) **NOT NULL**, A **CHAR**(8), B **CHAR**(8))
   PRIMARY KEY(K)

   GRANT SELECT ON R **TO** caio

   e siano stati immessi dei dati in R. Usando i comandi:

   DROP TABLE Nome;
   CREATE TABLE Nome (Attributo Tipo, ...) **AS** ExprSQL'
   CREATE VIEW Nome (Attributo, ...) **AS** ExprSQL;
   GRANT SELECT ON Nome **TO** Utente;

   mostrare come si possa modificare lo schema in modo che i dati di R vengano memorizzati esclusivamente nelle tabelle:

   R1(K **CHAR**(8) **NOT NULL**, A **CHAR**(8))
   R2(K **CHAR**(8) **NOT NULL**, B **CHAR**(8))

   e l'utente "caio" possa continuare a lavorare sulla base di dati come se esistesse la tabella:

   R(K **INTEGER**(8) **NOT NULL**, A **INTEGER**(8), B **INTEGER**(8)).

3. Definire lo schema per la base di dati relazionale ottenuta dall'Esercizio 5.3.
4. Si mostri come trattare il vincolo di chiave esterna con i *trigger*.
5. Si ricorda che date due sottoclassi C_1 e C_2 di una classe C, diciamo che:

   a) C_1 e C_2 soddisfano il vincolo di copertura di C se $C_1 \cup C_2 = C$;
   b) C_1 e C_2 soddisfano il vincolo di disgiunzione se non hanno nessun elemento in comune.

   In generale si possono quindi avere quattro tipi di situazioni:

   a) copertura disgiunta o partizione (vincolo di copertura e di disgiunzione);

b) copertura non disgiunta (solo vincolo di copertura);
c) sottoinsiemi disgiunti (solo vincolo di disgiunzione);
d) sottoinsiemi non disgiunti (nessun vincolo).

Si supponga di rappresentare C_1, C_2 e C con tre relazioni RC_1, RC_2 ed RC, con RC_1 ed RC_2 che contengono gli attributi propri di C_1 e C_2 e una chiave esterna per RC. Si supponga di poter dichiarare nello schema solo il vincolo di chiave primaria e di chiave esterna. Si mostri se è possibile rappresentare i vincoli dei quattro tipi di sottoclassi con il comando **CREATE TABLE**.

6. Si risolva l'esercizio precedente usando i *trigger*.

Note bibliografiche

Per maggiori dettagli sui comandi SQL per definire e amministrare basi di dati si rimanda ai testi citati nel Capitolo 1, mentre per le estensioni previste dei vari sistemi commerciali è necessario consultare i relativi manuali. Un testo molto utile per la messa a punto delle basi di dati che ogni DBA dovrebbe consultare è [Shasha and Bonnet, 2002].

I *trigger* e le basi di dati attive, con esempi di sistemi e metodologie di progetto, sono discussi in [Widom and Ceri, 1996], [Zaniolo et al., 1997].

BIBLIOGRAFIA

Abiteboul, S. and Bidoit, N. (1984). An algebra for non normalized relations. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*. 135

Abiteboul, S., Hull, R., and Vianu, V. (1995). *Database Foundations*. Addison-Wesley, Reading, Massachusetts. 27, 136, 178

Albano, A. (2001). *Costruire sistemi per basi di dati*. Addison-Wesley, Milano. 220, 252, 279

Albano, A., Antognoni, G., and Ghelli, G. (2000). View operations on objects with roles for a statically typed database language. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):548–567. 40, 48, 49

Albano, A., Cardelli, L., and Orsini, R. (1985). Galileo: A strongly typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260. Also in S.B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990. 40, 48, 49

Armstrong, W. (1974). Dependency structures of database relationships. In *Proceedings of the IFIP Congress*, pages 580–583. 143

Atzeni, P. and Antonellis, V. D. (1993). *Relational Database Theory*. Morgan Kaufmann Publishers, San Mateo, California. 136, 156, 178

Atzeni, P., Ceri, S., Paraboschi, S., and Torlone, R. (2002). *Basi di dati. Modelli e linguaggi di interrogazione*. McGraw-Hill, Milano. 27, 99

Batini, C., Ceri, S., and Navathe, S. (1992). *Conceptual Database Design. An Entity-Relationship Approach*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California. 72, 99

Batini, C., Lenzerini, M., and Navathe, S. (1987). A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):2–18. 90

Beeri, C. and Bernstein, P. (1979). Computational problems related to the design of normal form relational schemas. *ACM Transactions on Database Systems*, 4(1):30–59. 147

Bernstein, P. (1976). Synthesizing third normal form relations from functional

dependencies. *ACM Transactions on Database Systems*, 1(4):277–298. 169

Ceri, S., editor (1983). *Methodology and Tools for Database Design*. North-Holland, Amsterdam. 99

Ceri, S., Gottlob, G., and Tanca, L. (1990). *Logic Programming and Data Bases*. Springer-Verlag, Berlin. 136

Codd, E. (1970). A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387. 136, 141

Connolly, T. and Begg, C. (2000). *Database Solutions. A step-by-step guide to building databases*. Addison-Wesley, Reading, Massachusetts. 99

Diederich, J. and Milton, J. (1988). New methods and fast algorithms for database normalization. *ACM Transactions on Database Systems*, 13(3):339–365. 152

Elmasri, R. and Navathe, S. (2001). *Sistemi di basi di dati. Fondamenti. Prima edizione italiana*. Addison-Wesley, Milano. 27

Fraternali, P. and Tanca, L. (1995). A structured approach for the definition of the semantics of active databases. *ACM Transactions on Database Systems*, 20(4):414–471. 216

Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274. 73

Kifer, M., Bernstein, A., and Lewis, P. M. (2005). *Database Systems*. Addison-Wesley, Reading, Massachusetts, second edition. 27, 202, 250

Lucchesi, C. and Osborn, S. (1978). Candidate keys for relations. *Journal of Computer and System Sciences*, 17(2):270–280. 148

Maciaszek, L. A. (2002). *Sviluppo di sistemi informativi con UML*. Addison-Wesley, Milano. 99

Maier, D. (1983). *The Theory of Relational Databases*. Computer Science Press, Rockville, Maryland. 136, 152, 153, 178

Mannila, H. and Räihä, K. (1992). *The Design of Relational Databases*. Addison-Wesley, Reading, Massachusetts. 178

Marco, T. D. (1979). *Structured Analysis and System Specification*. Prentice Hall, Inc., Englewood Cliffs, New Jersey. 72

Ramakrishnan, R. and Gehrke, J. (2003). *Sistemi di basi di dati*. McGraw-Hill, Milano. 27

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Prentice Hall International, Inc., London. 72, 73

Schmidt, J. (1977). Some high level language constructs for data of type relation. *ACM Transactions on Database Systems*, 2(3):247–261. 239

Shasha, D. and Bonnet, P. (2002). *Database Tuning: principles, experiments, and troubleshooting techniques*. Morgan Kaufmann Publishers, San Mateo, California. 226

Silberschatz, H. F., Korth, H. F., and Sudarshan, S. (2002). *Database System Concepts*. McGraw-Hill, New York, 4th edition. 27, 136

Teorey, T. (1999). *Database Modeling and Design. The E-R Approach*. Morgan Kaufmann Publishers, San Mateo, California, third edition. 99

Tsou, D. and Fischer, P. (1982). Decomposition of a relation scheme into Boyce-Codd

Normal Form. *ACM SIGACT News*, 14(3):23–29. 165

Ullman, J. (1983). *Principles of Database Systems*. Computer Science Press, Rockville, Maryland, second edition. 158

Ullman, J. D. and Widom, J. (2001). *A First Course in Database System*. Prentice Hall, Inc., Englewood Cliffs, New Jersey, second edition. 27, 136, 152

van der Lans, R. (2001). *Introduzione a SQL. Seconda edizione italiana*. Addison-Wesley, Milano. 202, 250

Widom, J. and Ceri, S., editors (1996). *Active Database Systems: Trigger and Rules for Advanced Database Programming*. Morgan Kaufmann Publishers, San Mateo, California. 226

Yourdon, E. (1989). *Modern Structured Analysis*. Yourdon Press, Englewood Cliffs, New Jersey. 72

Zaniolo, C., Ceri, S., Faloutsos, C., Snodgrass, R., Subrahmanian, V., and Zicari, R., editors (1997). *Introduction to Advanced Database Systems*. Morgan Kaufmann Publishers, San Mateo, California. 226

INDICE ANALITICO