

Normalizzazione di schemi relazionali in Prolog



Anno accademico 2003/2004
Progetto per l'esame di
Linguaggi di Programmazione: Paradigmi di Programmazione
Docente del corso Prof. G. Aguzzi

Giacomo Sacchetti <sacchetti@nepero.net>

22 dicembre 2005

Indice

1	Teoria delle basi di dati	5
1.1	Introduzione	5
1.2	Il modello relazionale	5
1.3	Algebra relazionale	7
1.4	La normalizzazione	8
1.4.1	Dipendenze funzionali	9
1.4.2	Dipendenze derivate	10
1.4.3	Chiusura di un insieme di dipendenze funzionali	11
1.4.4	Chiavi	12
1.4.5	Copertura di un insieme di dipendenze	12
1.4.6	Decomposizione di schemi	12
1.5	Forme normali	14
1.5.1	Forma normale di Boyce-Codd	14
1.5.2	Terza forma normale	14
2	Implementazione in Prolog	17
2.1	Rappresentazione dei concetti base	17
2.2	Algoritmi di base	18
2.2.1	Chiusura di un insieme di attributi	18
2.2.2	Calcolo della copertura minimale	19
2.2.3	Proiezione di un insieme di dipendenze	19
2.3	Terza forma normale	20
2.4	Forma normale di Boyce-Codd	22
3	Uso del programma	23
3.1	Requisiti hardware e software	23
3.2	Installazione del programma	23
3.3	Esecuzione del programma	24

Capitolo 1

Teoria delle basi di dati

1.1 Introduzione

Al giorno d'oggi, sono molte le attività che necessitano dell'uso di una base di dati. I sistemi per la gestione di basi di dati, indicati spesso con DBMS (Data-Base Management System) costituiscono pertanto un forte interesse: la capacità di cercare, modificare, presentare, mettere in relazione dati, nella maniera più veloce e simile agli schemi mentali delle persone che devono usarle, è sempre stata (e sempre sarà) ambizione e necessità di molti programmi per computer.

Per questo motivo, si sono venuti a creare vari modelli teorici di organizzazione dei dati ed a questi sono seguite le implementazioni e l'uso.

Da modelli molto semplici che principalmente lavoravano su archivi sequenziali si è arrivati al modello relazionale ad oggetti passando dal gerarchico, dal reticolare e da altri ancora.

1.2 Il modello relazionale

I database relazionali sono attualmente il tipo di database più diffuso. I motivi di questo successo sono fondamentalmente:

- la semplicità e l'efficienza dei sistemi per rappresentare e manipolare i dati
- il fatto che essi si basano su un modello, quello relazionale, che ha solide basi teoriche

Il modello relazionale fu proposto originariamente da E.F. Codd in un articolo del 1970. Grazie alla sua coerenza ed usabilità, il modello è diventato a partire dagli anni '80 quello più utilizzato per la produzione di DBMS.

Il modello relazionale si basa due concetti, quello di *relazione* e quello di *tabella*. Tali concetti, anche se di natura diversa, sono facilmente riconducibili l'uno all'altro. Infatti, in tale modello, i dati sono immagazzinati in tabelle che prendono il nome di relazioni. Ciascuna relazione ha un numero finito di colonne, dette *attributi* ed un numero variabile di righe, dette *tuple*. Ad ogni attributo è associato un dominio.

Un esempio di relazione che descrive i dati relativi ai dipendenti di un'azienda è quello nella tabella (1.1). È opportuno fare una precisazione sull'utilizzo del termine *relazione*, infatti, può essere utilizzato con tre diversi significati:

Dipendenti

CODICE	COGNOME	UFFICIO	CAPOUFFICIO
001	Rossi	12	Ferrero
212	Verdi	3	Gandolfi
323	Bianchi	1	Barberi
431	Neri	12	Ferrero

Tabella 1.1: Relazione Dipendenti

- *relazione matematica*, sottoinsieme del prodotto cartesiano di insiemi (con ordine)
- *relazione* secondo la definizione del modello relazionale, leggermente diversa dal concetto di relazione matematica
- *relazione* (relationship), costruito del modello concettuale ER (Entity Relationship), usato per descrivere legami fra entità del mondo reale nella progettazione concettuale.

Possiamo allora distinguere, in riferimento alla relazione Dipendenti della tabella 1.1, le seguenti caratteristiche:

- non è definito alcun ordinamento fra le tuple, cioè tabelle con le stesse righe, ma in ordine diverso, rappresentano la stessa relazione.
- le tuple di una relazione sono distinte l'una dall'altra, in quanto in un insieme non possono esserci due elementi uguali.

Al tempo stesso però, all'interno di ogni tupla è definito un ordinamento, in modo da rispettare la corrispondenza tra l' i -esimo attributo e l' i -esimo dominio.

Tuttavia, è possibile associare il nome dell'attributo al dominio dello stesso in modo da ottenere una notazione non posizionale: tale associazione si realizza per mezzo della funzione $\text{Dom} : X \rightarrow D$ che fa corrispondere a ciascun attributo $A \in X$ un dominio $\text{Dom}(A) \in D$. Infine diciamo che una tupla su un insieme di attributi X è una funzione t che associa a ciascun attributo $A \in X$ un valore del dominio $\text{Dom}(A)$. In questo modo una relazione su X è un insieme di tuple su X .

Uno *schema di relazione*, è costituito da un simbolo R detto nome della relazione ed un insieme di attributi $X = \{A_1, A_2, \dots, A_n\}$. Solitamente il tutto è indicato con $R(X)$. Nella tabella 1.1 lo schema di relazione è Dipendenti(CODICE, COGNOME, UFFICIO, CAPOUFFICIO).

Uno *schema di base di dati* è un insieme di schemi di relazione, $\mathbf{R} = \{R_1(X_1), R_2(X_2), \dots, R_m(X_m)\}$.

Un'*istanza di relazione* (o semplicemente relazione) su uno schema $R(X)$ è un insieme di tuple su X .

Un'*istanza di base di dati* su uno schema $\mathbf{R} = \{R_1(X_1), R_2(X_2), \dots, R_m(X_m)\}$ è un insieme di relazioni $\mathbf{r} = \{r_1, r_2, \dots, r_m\}$ dove ogni r_i è una relazione sullo schema $R_i(X_i)$.

Lo *schema di relazione universale* U di un'istanza di base di dati ha come attributi l'unione degli attributi di tutte le relazioni della base di dati.

Le strutture del modello relazionale introdotte finora ci permettono di organizzare le informazioni efficacemente. Tuttavia, ci sono situazioni in cui non è vero che qualsiasi insieme di tuple rappresenta un'informazione corretta. Ad esempio, il voto finale di un esame universitario non può essere né 34 né 28 e lode, oppure non ci possono essere, nello stesso ateneo, due studenti con la stessa matricola.

Ad tal scopo è stato introdotto il concetto di *vincolo d'integrità*, una proprietà che deve essere soddisfatta dalle istanze che rappresentano informazioni corrette per l'applicazione. Tale vincolo può essere visto come un *predicato* che associa ad ogni istanza il valore *vero* o *falso*.

È possibile classificare i vincoli in base agli elementi coinvolti:

- Un vincolo è *intrarelazionale* se la sua soddisfacibilità è definita rispetto ad una singola relazione. Fra questi possiamo ancora distinguere i *vincoli di tupla* ed i *vincoli di dominio*.
- Un vincolo è *interrelazionale* se coinvolge più relazioni

1.3 Algebra relazionale

L'algebra relazionale è un linguaggio procedurale, basato su concetti di tipo algebrico, per la manipolazione delle basi di dati. Elenchiamo, senza soffermarci sui dettagli, i principali costrutti dell'algebra relazionale:

- l'unione di due relazioni, r_1 e r_2 definite sullo stesso insieme di attributi X è indicata con $r_1 \cup r_2$ ed è una relazione su X contenente le tuple che appartengono a r_1 oppure a r_2 oppure a entrambe;
- la differenza di $r_1(X)$ e $r_2(X)$ si indica con $r_1 - r_2$ ed è una relazione su X contenente le tuple che appartengono a r_1 e non a r_2 ;
- l'intersezione di $r_1(X)$ e $r_2(X)$ indicata con $r_1 \cap r_2$ è la relazione definita su X contenente le tuple che appartengono sia ad r_1 che a r_2 ;
- la ridenominazione di una relazione r con attributi A_1, \dots, A_k , si indica con

$$\rho_{B_1 B_2 \dots B_k \leftarrow A_1 A_2 \dots A_k}(r)$$

essa contiene una tupla t' per ogni tupla t in r con $t'[B_i] = t[A_i]$, $i = 1, \dots, k$.

- la selezione $\sigma_F(r)$ produce una relazione sugli stessi attributi di r che contiene le tuple di r su cui F (formula proposizionale) è vera.
- la proiezione di r su Y , indicata con $\pi_Y(r)$ è l'insieme di tuple su Y ottenute dalle tuple di r considerando soli i valori su Y , cioè $\pi_Y(r) = \{t[Y] \mid t \in r\}$.
- il join naturale $r_1 \bowtie r_2$ di $r_1(X_1)$ e $r_2(X_2)$ è una relazione definita su $X_1 X_2$ come segue

$$r_1 \bowtie r_2 = \{t \text{ su } X_1 X_2 \mid \text{esistono } t_1 \in r_1 \text{ e } t_2 \in r_2 \text{ con } t[X_1] = t_1 \text{ e } t[X_2] = t_2\}$$

- il theta join fra due relazioni $r_1(X_1)$ e $r_2(X_2)$ è dato da

$$r_1 \bowtie_F r_2 = \sigma_F(r_1 \bowtie r_2)$$

1.4 La normalizzazione

Nonostante il modello relazionale sia semplice e largamente diffuso, in situazioni complesse sorgono dei problemi, per cui è necessario ricorrere ad espedienti per modellare ad esempio attributi composti o multivalore, associazioni molti a molti, vincoli intrarelazionali.

In tali situazioni, esistono rappresentazioni diverse per descrivere la stessa cosa, per cui sorge la necessità di provare che queste diverse rappresentazioni sono equivalenti o che una rappresentazione ha una qualità migliore rispetto ad un'altra.

Lo scopo della teoria della normalizzazione è la definizione di criteri formali per provare l'equivalenza di diverse rappresentazioni e la qualità di tali, e la definizione di algoritmi per "trasformare" uno schema in un'altro.

DVDBank

UTENTE	RESIDENZA	TEL.	CODICEDVD	TITOLO	DATA
Rossi	Roma	3212	567	The Others	6/12/2004
Verdi	Firenze	2123	344	Matrix	15/9/2003
Bianchi	Milano	1360	925	L'esorcista	21/3/2001
Neri	Milano	1360	012	Harry Potter	1/6/2004

Tabella 1.2: Relazione DVDBank

La qualità di uno schema relazionale è solitamente misurata con l'assenza di *anomalie*; considerando ad esempio lo schema 1.2, abbiamo:

- la *ripetizione dell'informazione*, infatti ogni volta che un utente chiede un nuovo prestito vengono ripetuti i dati relativi alla residenza ed al numero telefonico. Questo non solo causa uno spreco di spazio, ma complica l'aggiornamento nel caso di cambio di residenza o numero telefonico.
- l'*impossibilità di rappresentare certi fatti*, infatti i dati relativi agli utenti possono essere memorizzati solo se hanno un prestito in corso.

Una soluzione per il precedente problema potrebbe essere quella di decomporre la relazione DVDBank in due relazioni:

Utenti(UTENTE,RESIDENZA,TEL.)

Prestiti(CODICEDVD,TITOLO,DATA, TEL.)

L'associazione fra le due relazioni è stata fatta per mezzo del numero di telefono. I dati delle relazioni sono ottenuti per mezzo di

Utenti= $\pi_{\text{Utente,Residenza,Tel.}}$ (DVDBank)

Prestiti= $\pi_{\text{CodiceDVD,Titolo,Data,Tel.}}$

Anche se questa decomposizione permette di eliminare la ripetizione dei dati, presenta l'anomalia di perdita di informazione (*lossy decomposition*). Infatti, se vogliamo conoscere tutti gli utenti che hanno preso un DVD in prestito nel mese di giugno utilizziamo

$$\pi_{\text{Utente,Residenza}}(\text{Utenti} \bowtie_{\text{Data} \in [1/6,31/6]} \text{Prestiti})$$

e quello che otteniamo è:

UTENTE	RESIDENZA
Bianchi	Milano
Neri	Milano

Tabella 1.3: Risultato query

Ma tale risultato è errato poichè il sig. Neri non ha preso in prestito un libro a giugno! Questo poichè la chiave di giunzione non identifica univocamente gli utenti. Un modo per ovviare questo problema è utilizzare UTENTE come chiave di giunzione. Tuttavia anche questa scelta può risultare inefficiente: per reperire la residenza di un utente nello schema originale basta una selezione, mentre in quest'ultimo caso dobbiamo fare anche la join.

Quello che si evince, quindi, è che la valutazione di uno schema relazionale non è un problema banale, soprattutto se si vuol tener conto anche del costo delle operazioni.

Lo scopo principale della teoria della normalizzazione è quello di fornire strumenti formali per la progettazione di basi di dati che non presentino anomalie, senza prendere in considerazione il costo delle operazioni.

Utilizzeremo in seguito la seguente notazione:

- A, B, A_1, B_2 per indicare attributi
- X, Y, T per indicare insiemi di attributi
- XY per $X \cup Y$, AB per $\{A, B\}$, XA per $X \cup \{A\}$
- $R(T)$ è uno schema di relazione, r una generica istanza, t una tupla di r . Se $X \subseteq T$ allora $t[X]$ è l'ennupla ottenuta da t considerando solamente gli attributi in X .

1.4.1 Dipendenze funzionali

Definizione 1.1 Una dipendenza funzionale fra un insieme di attributi X ed un insieme di attributi Y di uno schema di relazione $R(T)$, con $XY \subseteq T$, è un vincolo di integrità sulle istanze della relazione, espresso nella forma $X \rightarrow Y$ con il seguente significato: un'istanza r di $R(T)$ soddisfa la dipendenza $X \rightarrow Y$ se per ogni coppia di ennuple t_1, t_2 di r , se $t_1[X] = t_2[X]$ allora $t_1[Y] = t_2[Y]$, cioè:

$$X \rightarrow Y \Leftrightarrow \forall t_1, t_2 \in r, t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$$

È importante notare che:

- le dipendenze funzionali sono definite all'interno di uno schema di relazione

- non sono proprietà intensionali, riferite al significato, non inferibili da alcune istanze della relazione

Indicheremo d'ora in poi uno schema di relazione generico con $R \langle T, F \rangle$ con F insieme di dipendenze funzionali definite su T .

1.4.2 Dipendenze derivate

È facile dedurre da quanto presentato finora che le dipendenze funzionali definite su uno schema di relazione $R \langle T, F \rangle$ non sono solo quelle espresse in F , ma ne esistono altre derivabili da esse.

Definizione 1.2 Dato $R \langle T, F \rangle$, diciamo che $F \models X \rightarrow Y$ (F implica logicamente $X \rightarrow Y$), se ogni istanza valida r dello schema soddisfa anche $X \rightarrow Y$.

Questa definizione non fornisce un modo algoritmico per trovare tutte le dipendenze funzionali implicate da un insieme F : occorre un'assiomatizzazione delle dipendenze funzionali che fornisca un insieme corretto e completo di regole di inferenza, dette *assiomi*, che possono essere usate per derivare nuove dipendenze da un dato insieme di dipendenze.

Definizione 1.3 Sia R_I un insieme di regole di inferenza per F . Indichiamo con $F \vdash X \rightarrow Y$ il fatto che $X \rightarrow Y$ sia derivabile da F usando R_I

- l'insieme R_I è corretto se $F \vdash X \rightarrow Y \Rightarrow F \models X \rightarrow Y$
- l'insieme R_I è completo se $F \models X \rightarrow Y \Rightarrow F \vdash X \rightarrow Y$

L'insieme di regole di inferenza corretto e completo più utilizzato è:

- *Riflessività*: se $Y \subseteq X$ allora $X \rightarrow Y$
- *Arricchimento*: se $X \rightarrow Y$ e $W \subseteq T$ allora $XW \rightarrow YW$
- *Transitività*: se $X \rightarrow Y$ e $Y \rightarrow Z$ allora $X \rightarrow Z$

Le precedenti prendono genericamente il nome di assiomi di Armstrong.

Definizione 1.4 Una derivazione di f da F è una sequenza finita f_1, \dots, f_m di dipendenze, dove $f_m = f$ ed ogni f_i è un elemento di F oppure è ottenuta dalle dipendenze precedenti f_1, \dots, f_{i-1} della derivazione utilizzando una regola di inferenza.

Definizione 1.5 Dato uno schema $R \langle T, F \rangle$ con $X \subseteq T$, la chiusura di X rispetto a F è data da $\{A \in T \mid F \vdash X \rightarrow A\}$ e si indica con X_F^+ o con X^+ se non ci sono ambiguità.

Teorema 1.6 $F \vdash X \rightarrow Y \Leftrightarrow Y \subseteq X^+$

Teorema 1.7 Gli assiomi di Armstrong sono corretti e completi.

1.4.3 Chiusura di un insieme di dipendenze funzionali

Definizione 1.8 Dato un insieme F di dipendenze funzionali, si definisce la chiusura di F come $F^+ = \{X \rightarrow Y \mid F \vdash X \rightarrow Y\}$

Un problema che molto spesso si presenta è il cosiddetto *problema dell'implicazione*, cioè il decidere se una dipendenza funzionale appartiene a F^+ . Applicare gli assiomi di Armstrong esaustivamente a F per avere F^+ è una soluzione che ha complessità esponenziale, quindi è un metodo troppo complesso. Un metodo di complessità inferiore, sempre per risolvere il problema dell'implicazione è quello di controllare se $Y \subseteq X^+$ per decidere se $X \rightarrow Y \in F^+$. Esistono diversi algoritmi per far ciò, presentiamo quello che ha costo minore, detto *chiusura veloce*. Tale algoritmo utilizza le seguenti strutture dati: per ogni dipendenza $f = W \rightarrow V \in F$, sia $num(f)$ il numero di attributi di W che non sono ancora stati inseriti nell'insieme $XPIU$ (che alla fine sarà il vero e proprio X^+); per ogni attributo A , si costruisce una lista $L(A)$ delle dipendenze di F che contengono A nel determinante (cioè nella parte sinistra della dipendenza funzionale). L'idea dell'algoritmo è che quando un attributo A è aggiunto in $XPIU^i$ si visita la lista $L(A)$ e si decrementa il contatore $num(f)$ per ogni dipendenza nella lista. Quando il contatore di $f = W \rightarrow V$ è nullo, allora $W \subseteq XPIU^i$ e gli attributi di V possono essere aggiunti a $XPIU^{i+1}$ e quindi tale dipendenza non è più presa in considerazione in seguito. La complessità di questo algoritmo

Algorithm 1 Chiusura veloce

Require: $R < T, F >, X \subseteq T$

Ensure: $XPIU$

```

1: for all  $f = W \rightarrow V \in F$  do
2:    $num(f) := |W|$ ;
3:   for all  $A \in W$  do
4:     aggiungi  $f$  ad  $L(A)$ ;
5:   end for
6: end for
7:  $XPIU := X$ ;
8:  $NATT := X$ ;
9: while  $NATT \neq \emptyset$  do
10:  scegli  $A$  da  $NATT$ ;
11:   $NATT := NATT - A$ ;
12:  for all  $f = W \rightarrow V \in L(A)$  do
13:     $num(f) := num(f) - 1$ ;
14:    if  $num(f) = 0$  then
15:       $D := V - XPIU$ ;
16:       $NATT := NATT \cup D$ ;
17:       $XPIU := XPIU \cup D$ ;
18:    end if
19:  end for
20: end while

```

è, se indichiamo con a il numero di attributi e con p il numero di dipendenze funzionali, $O(ap)$.

1.4.4 Chiavi

Definizione 1.9 Dato un schema $R \langle T, F \rangle$, un insieme di attributi $W \subseteq T$ è una superchiave di R se $W \rightarrow T \in F^+$

Definizione 1.10 Dato un schema $R \langle T, F \rangle$, un insieme di attributi $W \subseteq T$ è una chiave di R se W è una superchiave e non esiste un sottoinsieme stretto di W che sia una superchiave di R .

Definizione 1.11 Dato un schema $R \langle T, F \rangle$, un attributo $A \in T$ si dice primo se e solo se appartiene ad almeno una chiave, altrimenti si dice non primo.

È possibile che in uno schema ci siano più chiavi, quindi di solito se ne sceglie una (quella con meno attributi), detta *chiave primaria*.

1.4.5 Copertura di un insieme di dipendenze

Al fine di operare su insiemi di dipendenze, è comodo portare tali insiemi ad una forma minima e per far ciò si introduce il concetto di *copertura*.

Definizione 1.12 Due insiemi di dipendenze, F e G sugli attributi T di una relazione R sono equivalenti ($F \equiv G$) se e solo se $F^+ = G^+$. Se $F \equiv G$ allora si dice che F è una copertura di G e viceversa.

Definizione 1.13 Sia F un insieme di dipendenze funzionali.

- Data $X \rightarrow Y \in F$, X contiene un attributo estraneo A se e solo se $(F - \{X \rightarrow Y\}) \cup \{X - \{A\} \rightarrow Y\} \equiv F$, ovvero se e solo se $X - \{A\} \rightarrow Y \in F^+$.
- $X \rightarrow Y$ è una dipendenza ridondante se e solo se $(F - \{X \rightarrow Y\}) \equiv F$, ovvero se e solo se $X \rightarrow Y \in (F - \{X \rightarrow Y\})^+$
- F è una copertura canonica se e solo se:
 1. ogni parte destra di dipendenza ha un unico attributo
 2. le dipendenze non contengono attributi estranei
 3. non esistono dipendenze ridondanti

Teorema 1.14 Per ogni insieme di dipendenze F esiste una copertura canonica.

Presentiamo allora un algoritmo per il calcolo della copertura canonica, che assume in input un insieme di dipendenze tali che per ognuna nella parte destra c'è un unico attributo (banalmente da $A \rightarrow BC$ si ha $A \rightarrow B, A \rightarrow C$).

1.4.6 Decomposizione di schemi

Per eliminare le anomalie da schemi relazionali abbiamo fissato inizialmente la tecnica di decomporre lo schema in schemi più piccoli che godono di particolari proprietà ma sono in qualche senso "equivalenti" allo schema originale.

Algorithm 2 Calcolo della copertura canonica

Require: F insieme di dipendenze funzionali**Ensure:** G copertura canonica di F

```

1:  $G := F$ ;
2: for all  $X \rightarrow Y \in G$  with  $|X| > 1$  do
3:    $Z := X$ ;
4:   for all  $A \in X$  do
5:     if  $Y \subseteq [Z - \{A\}]_F^+$  then
6:        $Z := Z - \{A\}$ ;
7:     end if
8:    $G := (G - \{X \rightarrow Y\}) \cap \{Z \rightarrow Y\}$ ;
9:   end for
10: end for
11: for all  $f \in G$  do
12:   if  $f \in (G - \{f\})^+$  then
13:      $G := G - \{f\}$ ;
14:   end if
15: end for

```

Definizione 1.15 Dato uno schema $R(T)$, $\rho = \{R_1(T_1), \dots, R_n(T_n)\}$ è una decomposizione di R se e solo se $\bigcup_i T_i = T$

Per ciò che riguarda l'equivalenza fra lo schema originale e la sua decomposizione si richiedono in genere due condizioni (indipendenti tra loro):

- decomposizioni che *preservano i dati*
- decomposizioni che *preservano le dipendenze*

Decomposizioni che preservano i dati

È necessario introdurre il concetto di *proiezione* su un insieme di dipendenze per capire il concetto di decomposizione che preserva i dati:

Definizione 1.16 Dato $R \langle T, F \rangle$ e $T_i \subseteq T$, la proiezione di F su T_i è:

$$\pi_{T_i}(F) = \{X \rightarrow Y \in F^+ \mid X, Y \subseteq T_i\}$$

Possiamo allora definire

Definizione 1.17 $\rho = \{R_1(T_1), \dots, R_n(T_n)\}$ è una decomposizione di $R \langle T, F \rangle$ che preserva i dati se e solo se per ogni istanza di relazione r che soddisfa $R \langle T, F \rangle$ risulta $r = (\pi_{T_1} r) \bowtie \dots \bowtie (\pi_{T_n} r)$.

In generale invece data una qualunque decomposizione di $R \langle T, F \rangle$ e r istanza che soddisfa $R \langle T, F \rangle$ si ha $r \subseteq (\pi_{T_1} r) \bowtie \dots \bowtie (\pi_{T_n} r)$. Vale allora il seguente teorema:

Teorema 1.18 Sia $\rho = \{R_1(T_1), R_2(T_2)\}$ una decomposizione di $R \langle T, F \rangle$. ρ è una decomposizione che preserva i dati se e solo se $T_1 \cap T_2 \rightarrow T_1 \in F^+$ oppure $T_1 \cap T_2 \rightarrow T_2 \in F^+$.

Decomposizioni che preservano le dipendenze

Definizione 1.19 $\rho = \{R_1(T_1), \dots, R_n(T_n)\}$ è una decomposizione di $R \langle T, F \rangle$ che preserva le dipendenze se e solo se $\bigcup \pi_{T_i}(F) \equiv F$.

Presentiamo di seguito un algoritmo per determinare se $X \rightarrow Y \in G$ senza calcolare G^+ .

Algorithm 3 Chiusura X_G^+

Require: $\rho = \{R_1(T_1), \dots, R_n(T_n)\}$ decomposizione di $R \langle T, F \rangle$, $X \subseteq T$

Ensure: X_G^+ , con $G = \bigcup \pi_{T_i}(F)$

- 1: $X_G^+ := X$;
 - 2: **while** X_G^+ è modificato **do**
 - 3: **for all** $i := 1$ **to** k **do**
 - 4: $X_G^+ := X_G^+ \cup ((X_G^+ \cap T_i)_F^+ \cap T_i)$;
 - 5: **end for**
 - 6: **end while**
-

1.5 Forme normali

I concetti finora introdotti ci servono per affrontare l'obiettivo della normalizzazione: la trasformazione di schemi generici in schemi normali. Presentiamo due tipi di forme normali: la *forma normale di Boyce-Codd (BCNF)* e la *terza forma normale*.

1.5.1 Forma normale di Boyce-Codd

Definizione 1.20 Uno schema $R \langle T, F \rangle$ è in BCNF se e solo se per ogni dipendenza funzionale non banale $X \rightarrow Y \in F^+$, X è una superchiave.

Teorema 1.21 Uno schema $R \langle T, F \rangle$ è in BCNF se e solo se per ogni dipendenza funzionale non banale $X \rightarrow Y \in F$, X è una superchiave.

Corollario 1.22 Uno schema $R \langle T, F \rangle$, con F copertura canonica, è in BCNF se e solo se per ogni dipendenza funzionale elementare $X \rightarrow A \in F$, X è una superchiave.

Presentiamo qui di seguito l'algoritmo che trasforma uno schema generico in BCNF.

1.5.2 Terza forma normale

La terza forma normale è una decomposizione meno restrittiva della BCNF, che gode della seguente proprietà: ogni schema $R \langle T, F \rangle$ ammette una decomposizione che preserva i dati, preserva le dipendenze ed è in 3NF. Tale decomposizione può inoltre essere ottenuta in tempo polinomiale. Lo svantaggio principale di questo tipo di decomposizione è che, essendo meno restrittiva della BCNF, può presentare che schemi con alcune anomalie.

Algorithm 4 BCNF**Require:** $R \langle T, F \rangle$, con F copertura canonica**Ensure:** $\rho = \{R_1, \dots, R_m\}$, decomposizione in BCNF che preserva i dati

- 1: $\rho := \{R \langle T, F \rangle\}$;
- 2: $n := 1$;
- 3: **while** esiste $R_i \langle T_i, F_i \rangle \in \rho$ non in BCNF per $X \rightarrow A$ **do**
- 4: $n := n + 1$;
- 5: $T' := X^+$;
- 6: $F' := \pi_{T'}(F_i)$;
- 7: $T'' := (T_i - T') \cup X$;
- 8: $F'' := \pi_{T''}(F_i)$;
- 9: $\rho := \rho - R_i \langle T_i, F_i \rangle + \{R_i \langle T', F' \rangle, R_n \langle T'', F'' \rangle\}$;
- 10: **end while**

Definizione 1.23 Uno schema $R \langle T, F \rangle$ è in 3NF se e solo se, per ogni dipendenza funzionale non banale $X \rightarrow A \in F^+$, allora X è una superchiave oppure A è primo.

Quindi, come si evince da questa definizione, se una relazione è in BCNF allora è anche in terza forma normale.

Teorema 1.24 Uno schema $R \langle T, F \rangle$ è in 3NF se e solo se, per ogni dipendenza funzionale $X \rightarrow A_1 \dots A_n \in F$ e per ogni $i \in \{1 \dots n\}$, $A_i \in X$ oppure X è una superchiave oppure A_i è primo.

Corollario 1.25 Uno schema $R \langle T, F \rangle$, con F copertura canonica, è in 3NF se e solo se, per ogni dipendenza funzionale $X \rightarrow A \in F$, X è un superchiave o A è primo.

Proposizione 1.26 Il problema di decidere se uno schema di relazione è in 3NF è NP-completo.

Presentiamo adesso un algoritmo intuitivo per la normalizzazione di un generico schema in 3NF.

Algorithm 5 Sintesi in 3NF**Require:** $R \langle T, F \rangle$ **Ensure:** $\rho = \{S_1, \dots, S_n\}$, decomposizione di R che preserva i dati e le dipendenze tale che ogni S_i è in 3NF, rispetto alle proiezioni di F su S_i .

- 1: Calcolo copertura canonica G di F ;
- 2: $\rho = \{\}$;
- 3: Sostituzione in G di ogni insieme $X \rightarrow A_1, \dots, X \rightarrow A_h$ di dipendenze con lo stesso determinante con $X \rightarrow A_1, \dots, A_h$;
- 4: Per ogni dipendenza $X \rightarrow Y \in G$ si creazione di uno schema con attributi XY in ρ ;
- 5: Eliminazione da ρ di ogni schema che sia già contenuto in un altro schema di ρ ;
- 6: Nel caso non ci sia schema in ρ con una superchiave per R , aggiunta si una schema con attributi W con W superchiave per R ;

Capitolo 2

Implementazione in Prolog

Questo capitolo descrive le problematiche di implementazione in linguaggio Prolog degli algoritmi di normalizzazione presentati nel capitolo precedente.

2.1 Rappresentazione dei concetti base

La rappresentazione dei concetti di schema di relazione, dipendenza funzionale e chiave sono realizzati rispettivamente tramite i termini `schema`, `fd` e `key`. Descriviamo questi ultimi in dettaglio:

- il termine `schema` rispetta la sintassi

```
schema(<nome della relazione>, <lista degli attributi>)
```

Ad esempio potremmo rappresentare la relazione `impiegati` formata dagli attributi `codice`, `nome`, `stipendio` con

```
schema(impiegati, [codice, nome, stipendio])
```

- il termine `fd` invece presenta la sintassi

```
schema(<nome della relazione>, <parte sinistra>, parte destra)
```

Quindi per rappresentare la dipendenza `codice` \rightarrow `nome`, relativa alla relazione `impiegati`,

```
fd(impiegati, [codice], [nome])
```

- per rappresentare la chiave si utilizza invece

```
key(<nome della relazione>, <lista degli attributi>)
```

e quindi per rappresentare la chiave `codice` della relazione `impiegati` utilizzeremo semplicemente

```
key(impiegati, [codice])
```

2.2 Algoritmi di base

Descriviamo adesso l'implementazione in Prolog degli algoritmi di base della teoria della normalizzazione.

2.2.1 Chiusura di un insieme di attributi

Cominciamo dall'algoritmo per il calcolo della chiusura di un insieme di dipendenze funzionali. Sappiamo che, dato F , decidere se una dipendenza funzionale $X \rightarrow Y$ appartiene a F^+ è equivalentemente a determinare se $Y \subseteq X^+$. Un algoritmo elementare per calcolare X^+ è:

```
slowclosure(Rel,X,Res) :- fd(Rel,LeftSide,RightSide),
    subset(LeftSide,X),
    not(subset(RightSide,X)),
    union(W,X,RightSide,Rel), !,
    slowclosure(Rel, W, Res).
slowclosure(Rel,X,Res) :- Res = X.
```

Questo algoritmo applica esaustivamente all'insieme di attributi X (che è rappresentato tramite una lista) le dipendenze funzionali della relazione a cui tali attributi appartengono. Ad ogni passo l'insieme X^+ temporaneo viene incrementato e l'algoritmo richiamato ricorsivamente, finché non esistono più attributi che contribuiscono al risultato. Questo algoritmo, nonostante "l'eleganza" del codice, ha una complessità elevata. Abbiamo definito nel capitolo precedente l'algoritmo di chiusura veloce, di complessità minore, la cui implementazione in Prolog è la seguente

```
fastclosure(Rel,X,Res) :- buildstructure(Rel, [], [], Num, L),
    XP = X, NATT = X,
    doclosure(XP, NATT, Num, L, XPIU),
    Res = XPIU.

buildstructure(Rel, Num, L, App, App2) :-
    fd(Rel,LeftSide,RightSide),
    card(LeftSide,N),
    not(member([LeftSide, RightSide, N],Num)),
    addtolistoflist(Num,[LeftSide, RightSide, N], NewNum),
    buildstructureL(LeftSide, RightSide, L, NewL), !,
    buildstructure(Rel, NewNum, NewL, App, App2).
buildstructure(Rel, Num, L, App, App2) :- App = Num, App2 = L.

doclosure(X, NATT, Num, L, XPIU) :-
    getelem(A, NATT),
    diff(NATT2, NATT, A),
    updatenum(X, NATT2, Num, A, L, [],
        FinalNum, FinalXPIU, FinalNATT),
    !,
    doclosure(FinalXPIU, FinalNATT, FinalNum, L, XPIU).
doclosure(X, NATT, Num, L, XPIU) :- XPIU = X.
```

È abbastanza evidente che il goal principale è `fastclosure`, che ha 3 parametri: il nome della relazione `Rel`, l'insieme di attributi `X` di `Rel` di cui si vuol fare la chiusura e `Res` variabile da unificare per avere il risultato. Questo predicato richiama `buildstructure` e `doclosure`: il primo “provvede” alla creazione delle strutture `num` e `L` dell’algoritmo (utilizzando anche ulteriori predicati che non riportiamo qui per brevità, ma che sono presenti nell’appendice A) ed il secondo esegue la vera e propria chiusura tramite l’utilizzo di queste strutture.

2.2.2 Calcolo della copertura minimale

Il calcolo della copertura minimale, realizzato dal predicato `findminimalcover`,

```
findminimalcover(Rel) :-
    elimstrangerattr(Rel),
    elimredundantdipfun(Rel).
```

consiste nell’eliminazione degli attributi estranei e delle dipendenze funzionali ridondanti. Per quanto riguarda l’eliminazione di attributi estranei, abbiamo il predicato `elimstrangerattr`. Il metodo con cui gli attributi estranei vengono eliminati è semplice: si tolgono attributi da ogni parte sinistra di dipendenze funzionale e si controlla se la chiusura degli attributi rimanenti contiene la parte destra originale.

```
elimstrangerattr(Rel) :-
    fd(Rel, LeftSide, RightSide),
    reduceleftside(Rel, LeftSide,
                  RightSide, NewLeftSide),
    not(LeftSide = NewLeftSide),
    retract(fd(Rel, LeftSide, RightSide)),
    asserta(fd(Rel, NewLeftSide, RightSide)),
    fail.
elimstrangerattr(Rel).
```

L’eliminazione delle dipendenze ridondanti è analogo, si esegue un `retract` di ogni dipendenze funzionale e si controlla la chiusura della parte sinistra.

```
elimredundantdipfun(Rel) :-
    fd(Rel, LeftSide, RightSide),
    retract(fd(Rel, LeftSide, RightSide)),
    fastclosure(Rel, LeftSide, LClosure),
    (
        (not(subset(RightSide, LClosure)),
         asserta(fd(Rel, LeftSide, RightSide)))
    );
    (subset(RightSide, LClosure))
),
fail.
elimredundantdipfun(Rel).
```

2.2.3 Proiezione di un insieme di dipendenze

Il calcolo della proiezione di un insieme di dipendenze è un algoritmo di complessità esponenziale, realizzato tramite

```

calculateproj(Rel, I, T, RelProjected) :-
    insiemepotenza(T, Pot),
    appendindextoname(Rel, I, RelProj),
    realcalculatepot(Rel, T, Pot, RelProj),
    RelProjected = RelProj.

```

I predicati utilizzati sono:

- `insiemepotenza`, che calcola l'insieme potenza dell'insieme di attributi T
- `appendindextoname`, che dà un nome allo schema "proiettato"
- `realcalculatepot`, che esegue le `assert` dell'insieme proiettato

2.3 Terza forma normale

L'implementazione Prolog per "trasformare" uno schema generico alla terza forma normale è la seguente

```

thirdnf(Rel) :-
    remembercovering(Rel),
    groupbyleftside(Rel),
    makesubschema(Rel),
    removetransitivedep(Rel),
    makerealsubschema(Rel).

```

Tale implementazione è organizzata per richiamare i predicati relativi alle 5 principali fasi dell'algorithmo $3NF$ visto nel capitolo precedente. Il predicato `remembercovering`, la cui implementazione Prolog è

```

remembercovering(Rel) :- fd(Rel, LeftSide, RightSide),
    assertz(remember(fd(Rel, LeftSide, RightSide))), fail.
remembercovering(Rel).

```

che ha come funzionalità quella di definire dei fatti `remember` che hanno come argomento tutte le dipendenze funzionali della relazione che si vuol normalizzare. Tali definizioni saranno poi utilizzate nei predicati successivi.

Il predicato `groupbyleftside` raggruppa tutte le dipendenze funzionali con la medesima parte sinistra, creando un'unica dipendenza funzionale con la parte sinistra comune e la parte destra data dall'unione delle precedenti.

```

groupbyleftside(Rel) :- fd(Rel, LeftSide, _),
    not(group(Rel, [LeftSide])),
    asserta(group(Rel, [LeftSide])),
    fastclosure(Rel, LeftSide, Closure),
    asserta(clo(Rel, LeftSide, Closure)), fail.
groupbyleftside(Rel).

```

Il precedente codice in realtà genera un insieme di fatti:

- i fatti che riguardano le parti sinistre (`group`)
- i fatti riguardanti la chiusura della parti sinistre (`clo`)

Tali fatti saranno utilizzati nei predicati successivi. Il predicato `makesubschema` provvede al raggruppamento vero e proprio di parti sinistre di dipendenze funzionali in sottoschemi:

```

makesubschema(Rel) :- clo(Rel, LeftSide1, LeftClosure1),
                      clo(Rel, LeftSide2, LeftClosure2),
                      not(LeftClosure1=LeftClosure2),
                      subset(LeftSide2,LeftClosure1),
                      subset(LeftSide1,LeftClosure2),
                      not(alreadyexistgroup(Rel,LeftSide1,LeftSide2)),
                      mergeleftside(Rel, LeftSide1, LeftSide2),
                      asserta(fdj(Rel,LeftSide1,LeftSide2)),
                      asserta(fdj(Rel,LeftSide2,LeftSide1)),fail.
makesubschema(Rel) :- clo(Rel, LeftSide, LeftClosure),
                      retract(clo(Rel,LeftSide,LeftClosure)), fail.
makesubschema(Rel) :- fdj(Rel, LeftSide, RightSide),
                      fd(Rel,LeftSide,A),
                      subset(A,RightSide),
                      retract(fd(Rel,LeftSide,A)), fail.
makesubschema(Rel).

```

Infatti, con questo predicato si fondono le dipendenze del tipo $X \rightarrow Y$ e $Y \rightarrow X$. I fatti `fdj` definiti in questo predicato sono utilizzati proprio a tale scopo. Con il predicato `removetransitivedep` si provvede all'eliminazione delle dipendenze transitive:

```

removetransitivedep(Rel) :-
    assertz((fd(Rel, LeftSide,RightSide) :-
             fdj(Rel,RightSide,LeftSide))), fail.
removetransitivedep(Rel) :- fd(Rel, LeftSide, RightSide),
    not(fdj(Rel, LeftSide, RightSide)),
    retract(fd(Rel, LeftSide, RightSide)),
    fastclosure(Rel, LeftSide, Z),
    (
    ( not(subset(RightSide, Z)),
      asserta(fd(Rel, LeftSide, RightSide)))
    ;
    ( subset(RightSide, Z), elimin(Rel,LeftSide))
    ),
    fail.
removetransitivedep(Rel) :-
    retract((fd(Rel, LeftSide, RightSide) :-
             fdj(Rel, LeftSide, RightSide))), fail.
removetransitivedep(Rel).

```

Tale algoritmo in pratica rimuove alcune dipendenze a valuta se la chiusura della parte sinistra è contenuta nella parte destra originale. Il predicato `makerealsubschema` di occupa invece di eseguire la vera e propria divisione in sottoschemi, considerando i gruppi (fatti `group`) finora generati e stabilendone un nome.

```

makerealsubschema(Rel) :- makerealsubschema(Rel, 0).
makerealsubschema(Rel, NR) :- group(Rel, G), NewNR is NR+1,

```

```

    makename(Rel, NewNR, NewRel),
    makeschema(Rel, NewRel, G), assertsomekeys(NewRel, G),
    assertz(decomp(Rel, NewRel)), assertz(in3nf(NewRel)),
    assertz(tnfdecomp(Rel, NewRel)), retract(group(Rel,G)),!,
    makerealsubschema(Rel, NewNR).
makerealsubschema(Rel, NR) :- killmodifiedfds(Rel),
    reassertrememberedfds.

```

2.4 Forma normale di Boyce-Codd

L'algoritmo per la normalizzazione in BCNF segue quello presentato nel capitolo precedente:

```

bcnforiginal(Rel, Res) :-
    findminimalcover(Rel),
    dobcnforiginal(0, 1, [Rel], Res).

```

Il predicato `findminimalcover`, visto precedentemente, si occupa del calcolo della copertura canonica (o minimale). Il vero e proprio calcolo della forma normale di Boyce-Codd è effettuato dal predicato `dobcnforiginal`. L'algoritmo è stato descritto nel capitolo precedente, per cui riportiamo direttamente l'implementazione:

```

dobcnforiginal(I, N, RelList, Res) :-
    schema(Rel, RelAttr),
    NewI is I+1,
    isnotinbcnfwhy(Rel, X),
    NewN is N+1,
    fastclosure(Rel, X, Tfirst),
    calculateproj(Rel, NewI, Tfirst, RelFirst),
    diffset(Tsecond2, RelAttr, Tfirst),
    fusion(Tsecond2, X, Tsecond),
    calculateproj(Rel, NewN, Tsecond, RelSecond),
    retract(schema(Rel,RelAttr)),
    asserta(schema(RelFirst, Tfirst)),
    assertz(schema(RelSecond, Tsecond)),
    updaterellist(RelList, Rel, RelFirst,
        RelSecond, NewRelList), !,
    dobcnforiginal(NewI, NewN, NewRelList, Res).
dobcnforiginal(I,N, RelList, Res) :- Res = RelList.

```

Capitolo 3

Uso del programma

3.1 Requisiti hardware e software

Il programma `normpro` è stato progettato per essere indipendente dalla piattaforma hardware e dal sistema operativo. Per questo motivo è rilasciato il codice sorgente del programma, in modo da poterlo eseguire direttamente dall'interprete Prolog. A tal proposito è doveroso precisare che l'interprete Prolog deve necessariamente essere `swi-prolog` versione 5.4.7 e tale deve comprendere le librerie XPCCE versione 6.4.3.

3.2 Installazione del programma

Il programma viene fornito in formato di archivio compresso, nei formati ZIP o TGZ. Una volta estratti i file dall'archivio, la struttura file dovrebbe essere la seguente:

```
images/3nf.xpm
  bcnf.xpm
  cmd_guida.xpm
  dialog.xpm
  help.xpm
  normpro_edit.xpm
  add.xpm
  cmd_cancel.xpm
  cmd_nuovo.xpm
  esci.xpm
  main.xpm
  normpro_main.xpm
  cmd_chiudi.xpm
  cmd_ok.xpm
  norm_res.xpm
  cmd_esci.xpm
  del.xpm
help/guidabase.hlp
```

```

    guidanorm.hlp
normpro.pl

```

3.3 Esecuzione del programma

Dopo avere correttamente installato il programma, è necessario avviare l'interprete Prolog nella cartella dove il programma è stato copiato. Ad esempio, sotto ambiente UNIX:

```

guest@linuxbox normpro-1.0 $ xpce
XPCE 6.4.3, February 2005 for i686-linux and X11R6
Copyright (C) 1993-2002 University of Amsterdam.
XPCE comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
The host-language is SWI-Prolog version 5.4.7

```

```

For HELP on prolog, please type help. or apropos(topic).
on xpce, please type manpce.

```

```
?-
```

A tal punto è richiesta la “compilazione” del programma, che può essere effettuata digitando:

```
?- [normpro].
```

```
...
```

```
% normpro compiled 0.29 sec, 76,620 bytes
```

```
Yes
```

```
?-
```

In tale fase verranno visualizzati molti warning relativi alle variabili singleton, ma tali non compromettono i risultati del programma.

A questo punto, per avviare il programma è sufficiente digitare

```
?- normpro.
```

La schermata che viene visualizzata è rappresentata nella figura (3.1). È presentata un'interfaccia grafica (sviluppata utilizzando XPCE) che consente di:

- uscire dal programma
- visualizzare la guida (situazione rappresentata in figura 3.2)
- definire un nuovo schema di relazione su chi richiamare gli algoritmi di normalizzazione

Quest'ultimo punto è indubbiamente la funzione principale di questo programma. Viene pertanto presentata l'interfaccia mostrata in figura (3.3). Come ben mostra tale figura, attraverso questa interfaccia è possibile inserire i dati dello

3.3. ESECUZIONE



Figura 3.2: Schermata iniziale con guida

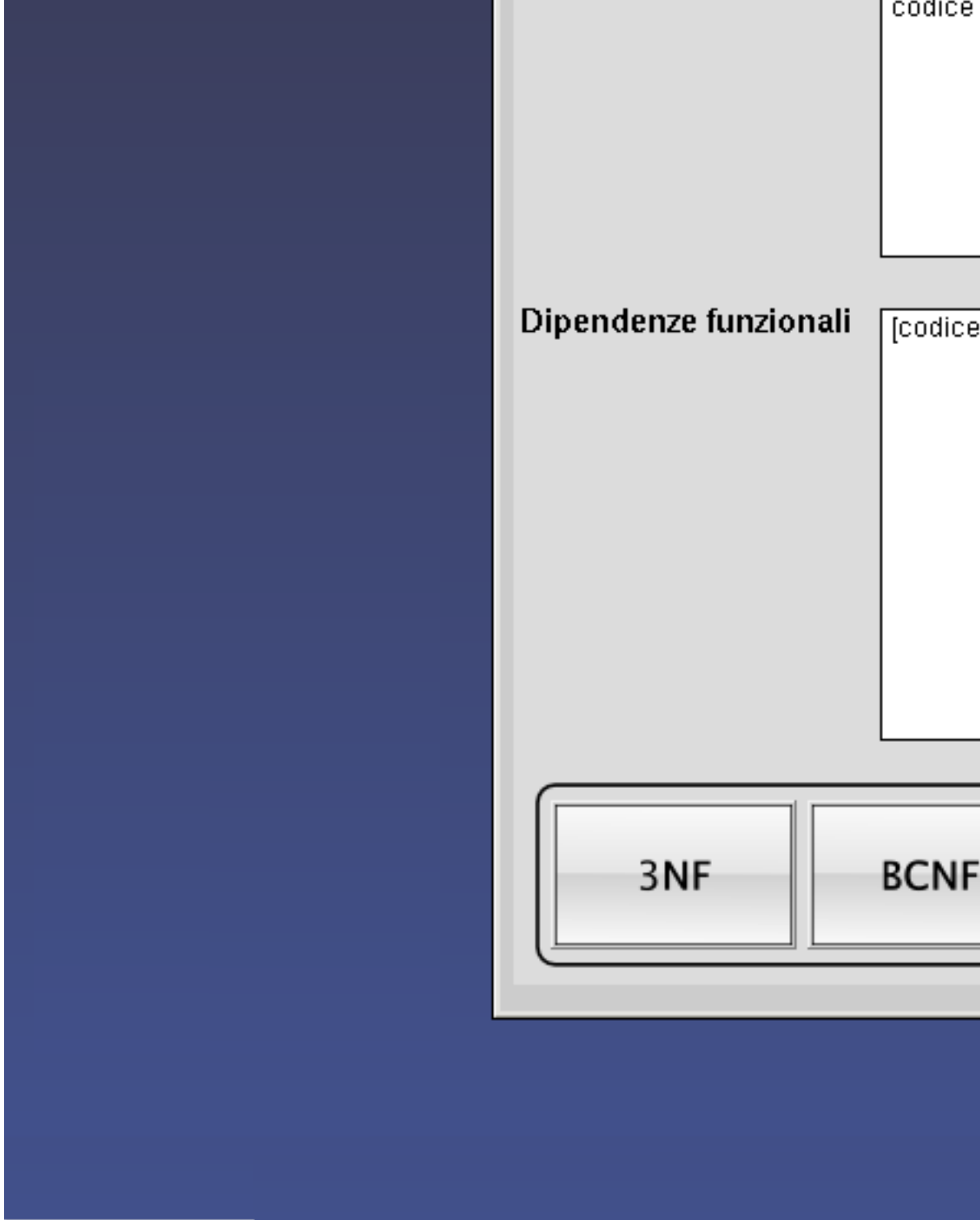


Figura 3.3: Interfaccia per la definizione dello schema relazionale

schema che si vuol normalizzare: questi sono il nome della relazione, gli attributi della relazione stessa e le dipendenze funzionali ad essa associate. Per la definizione del nome della relazione è sufficiente digitare da tastiera il nome dello schema in corrispondenza della casella di testo etichettata da Nome della relazione. La definizione della lista degli attributi è possibile, invece, attraverso i pulsanti Aggiungi e Rimuovi, collocati in corrispondenza dell'etichetta Lista attributi. Attraverso tali pulsanti è possibile modificare la lista degli attributi che viene presentata. Con un meccanismo simile è possibile definire le dipendenze funzionali, poste in corrispondenza dell'etichetta Dipendenze funzionali. In questo caso però, è necessario utilizzare una precisa sintassi per la loro definizione:

- le dipendenze hanno la forma:

$$[lista\ attributi\ parte\ sinistra] \rightarrow [lista\ attributi\ parte\ destra]$$

- le liste di attributi devono essere del tipo

$$attributo1, attributo2, \dots$$

- gli attributi citati nelle liste devono essere presenti nella definizione dello schema

È possibile consultare la guida durante il processo di definizione dello schema premendo sul pulsante Guida posto nella parte inferiore dell'interfaccia (vedi figura 3.4).

Una volta definito lo schema è possibile procedere con la normalizzazione, scegliendo tra normalizzazione 3NF o BCNF attraverso i pulsanti nella parte in basso dell'interfaccia. La pressione di tali pulsanti causerà l'avvio della procedura di normalizzazione, ed infine i risultati, come mostrato in figura (3.5).

Dipendenze funzionali

[codice]->[nome]

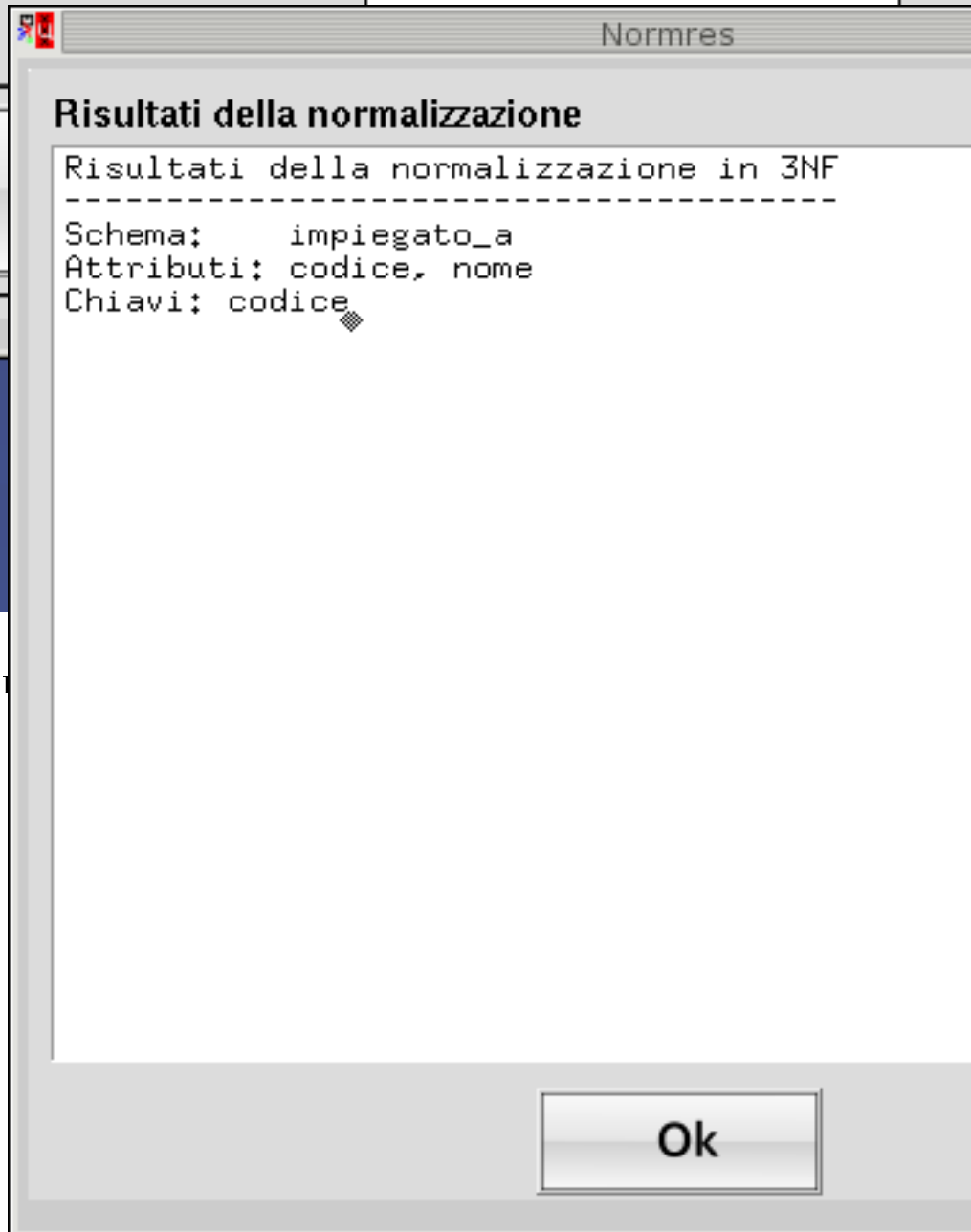


Figura 3.4: I

Figura 3.5: Risultati della normalizzazione

Elenco delle tabelle

1.1	Relazione Dipendenti	6
1.2	Relazione DVDBank	8
1.3	Risultato query	9

Elenco delle figure

3.1	Schermata iniziale	25
3.2	Schermata iniziale con guida	25
3.3	Interfaccia per la definizione dello schema relazionale	26
3.4	Interfaccia per la definizione dello schema relazionale e guida	28
3.5	Risultati della normalizzazione	28

Bibliografia

- [1] E.F. Codd: *A relational model of data for large shared data banks*, Communications of ACM 13, 6 (June 1970)
- [2] Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Riccardo Torlone: *Basi di dati*, McGraw-Hill, 1999
- [3] A. Albano, G. Ghelli, R. Orsini: *Basi di dati relazionali e a oggetti*, Zanichelli, 1997
- [4] Luca Console, Evelina Lamma, Paola Mello: *Programmazione logica e Prolog*, UTET Libreria, 1991
- [5] Paul Douglas, Steve Barker: *A logic programming E-Learning tool for teaching database dependency theory*, 2000