

# Basi di Dati 1

Simone Lidonnici

29 aprile 2024

# Indice

<b>1</b>	<b>Modello relazionale</b>	<b>3</b>
1.1	Relazioni come tabelle . . . . .	3
1.2	Scrivere tabelle corrette . . . . .	4
1.2.1	Vincoli di integrità . . . . .	5
1.2.2	Chiavi primarie . . . . .	5
1.3	Dipendenze funzionali . . . . .	6
<b>2</b>	<b>Algebra relazionale</b>	<b>7</b>
2.1	Basi dell'algebra relazionale . . . . .	7
2.2	Modifica di una relazione . . . . .	7
2.2.1	Proiezione . . . . .	7
2.2.2	Selezione . . . . .	8
2.2.3	Rinomina . . . . .	9
2.3	Operazioni insiemistiche . . . . .	9
2.3.1	Unione . . . . .	9
2.3.2	Differenza . . . . .	10
2.3.3	Intersezione . . . . .	10
2.4	Operazioni di combinazione . . . . .	11
2.4.1	Prodotto cartesiano . . . . .	11
2.4.2	Concatenazione . . . . .	12
2.4.3	Theta join . . . . .	13
2.5	Quantificazioni universali e esistenziali . . . . .	13
<b>3</b>	<b>Progettare un database relazionale</b>	<b>15</b>
3.1	Notazioni . . . . .	15
3.2	Dipendenza funzionali . . . . .	15
3.3	Chiusura di un insieme di dipendenza funzionali . . . . .	16
3.3.1	Assiomi di Armstrong . . . . .	16
3.4	Chiusura di un insieme di attributi . . . . .	17
3.4.1	Calcolare la chiusura di X . . . . .	18
3.4.2	Dimostrazione dell'algoritmo . . . . .	19
3.5	$F^+ = F^A$ . . . . .	20
3.6	Chiavi di uno schema . . . . .	21
<b>4</b>	<b>Terza forma normale</b>	<b>22</b>
4.1	Forma normale di Boyce-Codd . . . . .	23
4.2	Trasformare uno schema in 3NF . . . . .	23
4.3	Preservare F . . . . .	24
4.3.1	Teorema sulla chiusura . . . . .	24
4.4	Calcolare $X_G^+$ . . . . .	25
4.5	Equivalenza tra insiemi di dipendenze funzionali . . . . .	26
4.6	Trovare le chiavi di uno schema . . . . .	27
4.6.1	Teorema di unicità della chiave . . . . .	27
4.7	Join senza perdite . . . . .	28

4.7.1	Controllare un join senza perdite . . . . .	28
4.8	Copertura minimale . . . . .	30
4.9	Algoritmo di decomposizione . . . . .	30
<b>5</b>	<b>Organizzazione fisica</b>	<b>32</b>
5.1	Hardware di archiviazione e progettazione fisica . . . . .	32
5.1.1	Gerarchia di archiviazione . . . . .	32
5.1.2	Interno di un hard disk . . . . .	33
5.1.3	Passaggio dai concetti logici a quelli fisici . . . . .	34
5.2	Organizzazione dei file . . . . .	34
5.2.1	Organizzazione tramite file heap . . . . .	35
5.2.2	Organizzazione tramite file sequenziali . . . . .	35
5.2.3	Organizzazione tramite file hash . . . . .	35
5.2.4	Organizzazione tramite ISAM . . . . .	35
5.2.5	Organizzazione tramite B-Tree (bilanciato) . . . . .	36
<b>6</b>	<b>Concorrenza</b>	<b>39</b>
6.1	Transazioni . . . . .	39
6.2	Schedule di transazioni . . . . .	40
6.2.1	Garantire la serializzabilità . . . . .	40
6.3	Lock . . . . .	41
6.4	Lock binario . . . . .	41
6.4.1	Equivalenza di schedule con lock binario . . . . .	42
6.4.2	Algoritmo per testare la serializzabilità . . . . .	42
6.5	Lock a due fasi . . . . .	42
6.6	Lock a tre valori . . . . .	43
6.6.1	Equivalenza di schedule con lock a tre valori . . . . .	43
6.6.2	Algoritmo per testare la serializzabilità . . . . .	43
6.7	Lock write-only e read-only . . . . .	44
6.7.1	Equivalenza di schedule con lock write-only e read-only . . . . .	44
6.7.2	Algoritmo per testare la serializzabilità . . . . .	44
6.8	Deadlock . . . . .	45
6.8.1	Verificare e risolvere un deadlock . . . . .	45
6.9	Livelock . . . . .	45
6.10	Definizioni varie . . . . .	46
6.11	Tipi di protocolli . . . . .	46
6.11.1	Protocolli conservativi . . . . .	47
6.11.2	Protocolli aggressivi . . . . .	47
6.12	Timestamp . . . . .	47
6.12.1	Serializzabilità . . . . .	47
6.12.2	Read timestamp e write timestamp . . . . .	47

# 1

## Modello relazionale

### Prodotto cartesiano tra domini

Il **dominio** è un insieme, anche infinito, di valori utilizzabili.

Presi  $D_1, D_2, \dots, D_k$  domini, non necessariamente diversi, il **prodotto cartesiano**, scritto:

$$D_1 \times D_2 \times \dots \times D_k$$

é l'insieme:

$$\{(v_1, v_2, \dots, v_k) | v_i \in D_i \forall i\}$$

#### Esempio:

$$D_1 = \{a, b\}$$

$$D_2 = \{x, y, z\}$$

$$D_1 \times D_2 = \{(a, x), (a, y), (a, z), (b, x), (b, y), (b, z)\}$$

### Relazione

Una **relazione**  $r$  è un qualsiasi sottoinsieme di un prodotto cartesiano:

$$r \subseteq D_1 \times D_2 \times \dots \times D_k$$

Il **grado** della relazione è pari al valore di  $k$ , cioè al numero dei domini.

In una relazione, ogni elemento  $t \in r$  è una **tupla** che contiene numero di elementi pari al grado della relazione.

L'elemento  $i$ -esimo di una tupla viene indicato come  $t[i]$  oppure  $t.i$  con  $i \in [1, k]$ .

Una relazione ha una **cardinalità** pari al numero di tuple che la compongono.

#### Esempio:

$$D_1 = \{\text{white}, \text{black}\}$$

$$D_2 = \{0, 1, 2\}$$

$$D_1 \times D_2 = \{(\text{white}, 0), (\text{white}, 1), (\text{white}, 2), (\text{black}, 0), (\text{black}, 1), (\text{black}, 2)\}$$

La relazione  $r = \{(\text{white}, 0), (\text{black}, 1), (\text{black}, 2)\}$  è una relazione di grado 2 e cardinalità 3

La tupla  $t = (\text{white}, 0) \implies t[1] = \text{white}$  e  $t[2] = 0$

## 1.1 Relazioni come tabelle

Le relazioni possono essere rappresentate come tabelle, in cui viene dato un nome ad ogni colonna.

### Attributi e schema relazionale

Un **attributo**  $A$  è una coppia  $(A, \text{dom}(A))$  in cui  $A$  è il nome dell'attributo e  $\text{dom}(A)$  il suo dominio. Due attributi possono avere stesso dominio ma non stesso nome.

Uno **schema relazionale** è l'insieme di tutti gli attributi che definiscono la relazione associata allo schema stesso:

$$R(A_1, A_2, \dots, A_k)$$

In uno schema relazionale, una tupla è una funzione che associa ad ogni attributo  $A$  un elemento del  $\text{dom}(A)$ . Con  $t[A]$  indichiamo il valore della tupla corrispondente all'attributo  $A$ .

Un insieme di tuple  $r$  è un'**istanza di una relazione**.

**Esempio:**

$R = \{(\text{Nome}, \text{String}), (\text{Cognome}, \text{String}), (\text{Media}, \text{Real})\}$

Nome	Cognome	Media
Marco	Casu	28
Simone	Lidonnici	27

### Schema di un database

Un insieme di relazioni distinte è uno **schema di un database**:

$$(R_1, R_2, \dots, R_n)$$

Un database relazionale è uno schema di un database in cui sono definite le istanze  $(r_1, r_2, \dots, r_n)$  in cui  $r_i$  è un'istanza di  $R_i \forall i$ .

## 1.2 Scrivere tabelle corrette

I collegamenti tra i vari dati in schemi diversi vengono fatti attraverso i valori.

**Esempio:**

Matricola	Nome	Cognome
2041612	Marco	Casu
2061343	Simone	Lidonnici

Matricola	Voto	Esame
2041612	30	Sistemi
2061343	28	Algoritmi

In alcuni casi i potrebbero mancare delle informazioni, in questi casi si inserisce **NULL** come valore nella tupla, nel campo dell'attributo mancante. **NULL** è un valore che non appartiene a nessun dominio ma può rimpiazzare qualunque valore.

**Esempio:**

Matricola	Nome	Cognome
NULL	Nadia	Ge
2061343	NULL	Lidonnici

### 1.2.1 Vincoli di integrità

I dati devono avere dei vincoli da rispettare, questi vincoli possono essere:

- **Vincolo di dominio:** valori che devono essere per forza contenuti in un insieme specifico
- **Vincolo intra-relazionale:** sono vincoli tra valori di una stessa tupla
- **Vincolo inter-relazionale:** sono vincoli tra valori di diverse istanze
- **Vincolo di chiave primaria:** il valore deve essere diverso da NULL e unico negli attributi appartenenti alla chiave
- **Vincolo di esistenza:** il valore deve per forza essere diversa da NULL

**Esempio:**

Studenti:

Matricola	Nome	Cognome
2041612	Marco	Casu
2061343	Simone	Lidonnici

Voti:

Matricola	Esame	Voto	Lode
2061343	Probabilità	30	Si
2041612	Sistemi	28	NULL

In questo caso i vincoli sono:

- Vincolo di dominio:  $18 \leq \text{Voto} \leq 30$
- Vincolo intra-relazionale:  $\neg(\text{Voto} = 30) \implies \neg(\text{Lode} = \text{Si})$
- Vincolo inter-relazione: La matricola nello schema Voti deve esistere nello schema Studenti
- Vincolo di chiave primaria: La matricola nello schema Studenti deve per forza esistere ed essere unica
- Vincolo di esistenza: Il nome nello schema Studenti deve per forza esistere

### 1.2.2 Chiavi primarie

#### Chiave di una relazione

Una **chiave** di una relazione è un attributo o un gruppo di attributi che devono esistere ed appartenere ad una sola tupla.

Un insieme  $X$  di attributi dello schema  $R$  è una chiave di  $R$  se:

1. Per ogni istanza di  $R$  non esistono due tuple con gli stessi valori in tutti gli attributi di  $X$ , cioè  $\forall t_1, t_2 \in r \quad t_1[X] = t_2[X] \implies t_1 = t_2$
2. Per ogni sottoinsieme  $X' \subset X$  possono esserci delle tuple che hanno tutti gli attributi di  $X'$  uguali ma sono diverse, cioè  $\forall X' \subset X \quad \exists t_1, t_2 | t_1[X'] = t_2[X'] \text{ ma } t_1 \neq t_2$

**Esempio:**

Lezione	Edificio	Aula	Orario
Basi di Dati	Chimica	2	16:00
Algebra	Informatica	2	17:00
Probabilità	Matematica	4	15:30

$X = \{\text{Edificio, Aula, Orario}\}$  è una chiave primaria perché due lezioni non possono essere nello stesso edificio, aula e ora.

Togliendo però qualsiasi dei tre si possono avere delle lezioni diverse che abbiano gli altri attributi uguali (lezioni nello stesso edificio e aula ma ore diverse, nella stessa aula e ora ma edificio diverso o nello stesso edificio e ora ma aule diverse).

## 1.3 Dipendenze funzionali

### Dipendenza funzionale

Una **dipendenza funzionale** è un insieme di attributi:

$$X, Y \subseteq R \mid X \rightarrow Y$$

Si dice “X determina Y” in cui  $X$  è il determinante e  $Y$  è il dipendente.

#### Esempio:

Flight(Code,Day,Pilot,Time)

Questo schema ha dei vincoli dati dal buonsenso:

- un volo con un codice parte sempre alla stessa
- un volo parte con un solo pilota, in un determinato orario e tempo

Questi vincoli determinano delle dipendenze funzionali:

- $\text{Code} \rightarrow \text{Time}$
- $\{\text{Day, Pilot, Time}\} \rightarrow \text{Code}$
- $\{\text{Code, Day}\} \rightarrow \text{Pilot}$

### Soddisfazione di una dipendenza funzionale

Un'istanza  $r$  di uno schema  $R$  soddisfa la dipendenza funzionale  $X \rightarrow Y$  se:

1. La dipendenza è applicabile su  $R$ , cioè  $X \subset R \wedge Y \subset R$
2. Due tuple in  $R$  che hanno gli stessi attributi  $X$  avranno anche gli stessi attributi  $Y$ , cioè  $\forall t_1, t_2 \in r \ t_1[X] = t_2[X] \implies t_1[Y] = t_2[Y]$

# 2

## Algebra relazionale

L'algebra relazionale è una notazione per specifiche domande (dette query) sul contenuto delle relazioni, le operazioni in algebra relazionale hanno una controparte in SQL. Per processare una query il DBMS trasforma SQL in una notazione simile all'algebra relazionale.

Il linguaggio per interrogare un database è un insieme di operatori binari o unitari applicati a una o più istanze per generare una nuova istanza.

### 2.1 Basi dell'algebra relazionale

L'algebra relazionale essendo un'algebra è composta da un dominio e delle operazioni che danno come risultato un elemento interno al dominio.

In questo caso i domini sono le relazioni, rappresentate come un insieme di tuple e i risultati delle operazioni sono un altro insieme di tuple.

Le query sono domande sull'istanza delle relazioni.

Ci sono 3 tipi di operazioni:

- modificare una relazione: proiezione, selezione e rinomina
- operazioni su insiemi: unione, intersezione e differenza
- combinare tuple di due relazioni: prodotto cartesiano, join e theta-join

### 2.2 Modifica di una relazione

#### 2.2.1 Proiezione

##### Proiezione ( $\pi$ )

La **proiezione** esegue un taglio verticale su una relazione e sceglie un sottoinsieme degli attributi, creando una tabella con solo questi attributi.

$$\pi_{A_1, \dots, A_k}(R)$$

Prende solo le colonne di un'istanza con attributi  $A_1, \dots, A_k$ .

**Esempio:**

Customer:



Nome	C#	Città
Rossi	1	Roma
Rossi	2	Milano
Bianchi	3	Roma
Verdi	4	Roma
Rossi	5	Roma

Eseguendo  $\pi_{\text{Nome, Città}}(\text{Customer})$ :

Nome	Città
Rossi	Roma
Rossi	Milano
Bianchi	Roma
Verdi	Roma

### 2.2.2 Selezione

#### Selezione ( $\sigma$ )

La **selezione** fa un taglio orizzontale all'istanza di una relazione e sceglie tutte le tuple che rispettano una determinata condizione.

$$\sigma_C(R)$$

La condizione  $C$  è un'espressione booleana nella forma:

$$A \theta B \text{ o } A \theta a$$

In cui:

- $\theta$  è un operatore di comparazione, cioè  $\{<, >, =, \leq, \geq, \text{ecc...}\}$
- $A$  e  $B$  hanno lo stesso dominio
- $a$  è un elemento del dominio di  $A$  usato come costante

**Esempio:**

Customer:

Nome	C#	Città
Rossi	1	Roma
Rossi	2	Milano
Bianchi	3	Roma
Verdi	4	Roma
Rossi	5	Roma

Eseguendo  $\sigma_{\text{Town}=\text{Roma}}(\text{Customer})$ :

Nome	C#	Città
Rossi	1	Roma
Bianchi	3	Roma
Verdi	4	Roma
Rossi	5	Roma

### 2.2.3 Rinomina

#### Rinomina ( $\rho$ )

La **rinomina** cambia il nome di un attributo in un altro.

$$\rho_{A_1 \leftarrow A_2}(R)$$

L'attributo  $A_1$  viene rinominato in  $A_2$ .

## 2.3 Operazioni insiemistiche

Le operazioni insiemistiche possono essere fatte solo su istanze compatibili.

Due istanze sono compatibili se:

- hanno lo stesso numero di attributi
- gli attributi corrispondenti hanno lo stesso dominio

### 2.3.1 Unione

#### Unione ( $\cup$ )

L'**unione** crea una nuova istanza contenente tutte le tuple appartenenti a una delle due istanze unite.

$$r_1 \cup r_2$$

Bisogna stare attenti a non unire istanze con attributi che hanno lo stesso dominio ma che non c'entrano niente (ed esempio età e matricola).

Nel caso in cui gli attributi abbiano nomi diversi vengono presi i nomi della prima relazione.

#### Esempio:

Teachers:

Nome	Codice	Dipartimento
Rossi	1	Matematica
Bianchi	2	Fisica
Verdi	3	Inglese

Admin:

Nome	Codice	Dipartimento
Esposito	1	Matematica
Perelli	2	Informatica
Verdi	3	Inglese

Eseguendo  $AllStaff = Teachers \cup Admin$ :

Nome	Codice	Dipartimento
Rossi	1	Matematica
Bianchi	2	Fisica
Verdi	3	Inglese
Esposito	1	Matematica
Perelli	2	Informatica

In questo caso abbiamo cancellato una riga che contiene (Verdi, 3, Inglese), per sistemarlo potremmo cambiare il codice degli insegnanti in T1,T2, ecc... e quello degli admin in A1,A2, ecc...

### 2.3.2 Differenza

#### Differenza ( $-$ )

La **differenza** crea una nuova istanza contenente tutte le tuple della prima che non sono presenti nella seconda. Si identifica con  $-$ .

$$r_1 - r_2$$

La differenza al contrario di unione e intersezione non è commutativa.

#### Esempio:

Students:

Nome	Codice	Dipartimento
Rossi	1	Matematica
Bianchi	2	Inglese
Verdi	3	Informatica

Admin:

Nome	Codice	Dipartimento
Esposito	4	Informatica
Bianchi	2	Inglese
Perelli	5	Matematica

Eseguendo Students – Admins:

Nome	Codice	Dipartimento
Rossi	1	Matematica
Verdi	3	Informatica

Eseguendo Admin – Students:

Nome	Codice	Dipartimento
Esposito	4	Informatica
Perelli	5	Matematica

### 2.3.3 Intersezione

#### Intersezione ( $\cap$ )

L'**intersezione** crea una nuova istanza con le tuple in comune alle due istanze.

$$r_1 \cap r_2$$

Può anche essere definita come  $r_1 - (r_1 - r_2)$ .

## 2.4 Operazioni di combinazione

### 2.4.1 Prodotto cartesiano

#### Prodotto cartesiano ( $\times$ )

Il **prodotto cartesiano** crea una relazione con tutte le possibili combinazioni delle tuple della prima e della seconda relazione.

$$R_1 \times R_2$$

#### Esempio:

Customers:

Nome	C#	Città
Rossi	1	Roma
Rossi	2	Milano
Bianchi	3	Roma

Admin:

O#	CC#	A#	Pezzi
1	1	2	100
2	2	2	250
3	3	3	50

Eseguendo  $\text{Orders} = \text{Customer} \times \text{Order}$ :

Nome	C#	Città	O#	CC#	A#	Pezzi
Rossi	1	Roma	1	1	2	100
Rossi	1	Roma	2	2	2	250
Rossi	1	Roma	3	3	3	50
Rossi	2	Milano	1	1	2	100
Rossi	2	Milano	2	2	2	250
Rossi	2	Milano	3	3	3	50
Binchi	3	Roma	1	1	1	100
Binchi	3	Roma	2	2	2	250
Binchi	3	Roma	3	3	3	50

In questo caso ho collegato dei Customer ad alcuni Order con C# diversi sbagliando e avendo alcune tuple in eccesso, quindi eseguo una selezione solo delle tuple che hanno C# uguale a CC# e poi successivamente una proiezione per eliminare l'attributo CC# ora superfluo.

Eseguendo  $\text{Final} = \pi_{-CC\#}(\sigma_{C\#=CC\#}(\text{Customer} \times \text{Order}))$ :

Nome	C#	Città	O#	A#	Pezzi
Rossi	1	Roma	1	2	100
Rossi	2	Milano	2	2	250
Binchi	3	Roma	3	3	50

### 2.4.2 Concatenazione

#### Concatenazione ( $\bowtie$ )

La **concatenazione** seleziona le tuple di un prodotto cartesiano che hanno gli stessi valori negli attributi in comune:

$$r_1 \bowtie r_2 = \pi_{XY}(\sigma_C(r_1 \times r_2))$$

Dove:

- $C = (R_1.A_1 = R_2.A_1) \wedge (R_1.A_2 = R_2.A_2) \wedge \dots \wedge (R_1.A_k = R_2.A_k)$
- $X$  è l'insieme degli attributi di  $r_1$
- $Y$  è l'insieme degli attributi di  $r_2$  non in  $r_1$
- Gli attributi doppi vengono automaticamente tolti

Ci sono alcuni casi speciali:

- Se non ci sono istanze che soddisfano le condizioni il risultato sarà l'insieme vuoto (che è comunque una relazione)
- Se le relazioni non hanno attributi in comune il risultato è semplicemente il prodotto cartesiano

Dobbiamo inoltre stare attenti che attributi con lo stesso nome abbiano lo stesso significato.

**Esempio:**

Customers:

Nome	C#	Città
Rossi	1	Roma
Rossi	2	Milano
Bianchi	3	Roma

Admin:

O#	CC#	A#	Pezzi
1	1	2	100
2	2	2	250
3	3	3	50
1	1	3	200

Eseguendo  $\text{Orders} = \text{Customer} \bowtie \text{Order}$ :

Nome	C#	Città	O#	CC#	A#	Pezzi
Rossi	1	Roma	1	1	2	100
Rossi	1	Roma	1	1	3	200
Rossi	2	Milano	2	2	2	250
Binchi	3	Roma	3	3	3	50

### 2.4.3 Theta join

#### Theta join

Il **theta join** è una variazione del join in cui la condizione di selezione non è per forza il fatto che attributi con nome uguale abbiano valore uguale ma possiamo scegliere noi quali attributi collegare tra loro.

$$r_1 \bowtie_{A\theta B} r_2 = \pi_{XY}(\sigma_{A\theta B}(r_1 \times r_2))$$

In questo caso il join non collegherà due attributi con lo stesso nome ma collegherà l'attributo  $A$  e  $B$  in base alla condizione.

#### Esempio:

Customers:

Nome	C#	Città
Rossi	1	Roma
Rossi	2	Milano
Bianchi	3	Roma

Admin:

O#	CC#	A#	Pezzi
1	1	2	100
2	2	2	250
3	3	3	50

Eseguendo  $\text{Orders} = \text{Customer} \bowtie_{C\#=CC\#} \text{Order}$ :

Nome	C#	Città	O#	CC#	A#	Pezzi
Rossi	1	Roma	1	1	2	100
Rossi	1	Roma	1	1	3	200
Rossi	2	Milano	2	2	2	250
Binchi	3	Roma	3	3	3	50

## 2.5 Quantificazioni universali e esistenziali

Quando si scrive una query possiamo intercambiare la quantificazione universale con quella esistenziale, negando la proposizione:

$$\forall x \rightarrow \varphi(x) = \neg \exists x \rightarrow \neg \varphi(x)$$

Bisogna ricordare che il contrario di “sempre” è “esiste almeno un caso in cui è falso” e non “è falso in ogni caso”.

$$\begin{aligned} \neg(\forall x \rightarrow \varphi(x)) &\neq \exists x \rightarrow \varphi(x) \\ \neg(\forall x \rightarrow \varphi(x)) &= \exists x \rightarrow \neg \varphi(x) \end{aligned}$$

#### Esempio:

Customers:

Nome	C#	Città
Rossi	1	Roma
Rossi	2	Milano
Bianchi	3	Roma

Admin:

C#	A#	Pezzi
1	1	50
1	1	200
2	2	150
3	3	125

Se vogliamo trovare i nomi e città delle persone che hanno sempre fatto ordini da più di 100 pezzi dobbiamo eliminare dal totale le persone che hanno fatto anche solo una volta un ordine da meno di 100 pezzi:

$$\pi_{\text{Nome, Città}}((\text{Customer} \bowtie \text{Order}) - \sigma_{\text{n}^\circ \text{ pezzi} \leq 100}(\text{Customer} \bowtie \text{Order}))$$

Employees:

Nome	C#	Sezione	Salario	Sup#
Rossi	1	B	100	3
Pirlo	2	A	200	3
Bianchi	2	A	500	NULL
Neri	4	B	150	1

Per trovare i dipendenti con stipendio maggiore dei supervisor bisogna fare prima un join della tabella con una copia di se stessa collegando il codice dei supervisor con i codici dei dipendenti. Per non confonderci rinominiamo tutti i valori della copia con una C davanti.

CEmployees = Employees

Employees2 = Employees  $\bowtie$  CEmployees:  
 $\text{Employees.Supervisor\#} = \text{CEmployees.C\#}$

Nome	C#	Sez	Sal	Sup#	CNome	C#	CSez	CSal	CSup#
Rossi	1	B	100	3	Bianchi	3	A	500	NULL
Pirlo	2	A	200	3	Bianchi	3	A	500	NULL
Neri	4	B	150	1	Rossi	1	B	100	3

Successivamente bisogna selezionare solo le tuple in cui lo stipendio dei supervisor è minore dei dipendenti e poi proiettare solo nome, codice e salario del dipendente.

$$\text{Final} = \pi_{\text{Nome, C\#, Salario}}(\sigma_{\text{Sal} > \text{CSal}}(\text{Employees2}))$$

Nome	C#	Sal
Neri	4	150

# 3

## Progettare un database relazionale

Per creare un database dobbiamo analizzare gli attributi che deve contenere il database e capire se è meglio fare uno schema singolo o multipli schemi collegati. È preferibile dividere gli attributi in più schemi per evitare problemi di inserimento (se alcuni dati possono essere aggiunti successivamente), cancellazione (se bisogna cancellare una tupla ma lasciare l'esistenza di alcuni attributi) o ridondanza. Per progettare un buon database bisogna separare i concetti in schemi diversi. Possiamo identificare i concetti rappresentati in una relazione dalla chiave.

### 3.1 Notazioni

- Uno schema relazionale viene definito con  $R$  ed è un insieme di attributi:  $R = \{A_1, A_2, \dots, A_n\}$  oppure  $R = A_1, A_2, \dots, A_n$
- Un attributo viene definito con le prime lettere dell'alfabeto:  $A, B, \dots$
- - Un insieme di attributi viene definito con le ultime lettere dell'alfabeto:  $X, Y, \dots$
- L'unione di due insiemi  $X \cup Y$  viene definita con  $XY$
- Una tupla  $t$  su  $R$  è una funzione che associa ad ogni attributo di  $A_i \in R$  un valore  $t[A_i]$  nel dominio di  $A_i$

### 3.2 Dipendenza funzionali

#### Dipendenza funzionali

Una dipendenza funzionale è un insieme di attributi:

$$X, Y \subseteq R | X \rightarrow Y$$

Si dice “X determina Y” in cui  $X$  è il determinante e  $Y$  è il dipendente. L'insieme delle dipendenze funzionali di uno schema si identifica con  $F$ . Un'istanza  $r$  è legale rispetto ad  $F$  se:

$$\forall (X \rightarrow Y) \in F \implies r \text{ SAT } (X \rightarrow Y)$$

Se  $F = \{A \rightarrow B, B \rightarrow C\}$  allora è come se in  $F$  ci fosse anche  $A \rightarrow C$ .



### Esempio:

Customers:

A	B	C
1	1	1
1	0	1
1	1	1

Soddisfa le dipendenza perché:

$$t_1[AB] = t_3[AB] \implies t_1[C] = t_3[C]$$

Admin:

A	B	C
1	1	1
1	0	1
1	1	0

Non soddisfa le dipendenza perché:

$$t_1[AB] = t_3[AB] \text{ ma } t_1[C] \neq t_3[C]$$

## 3.3 Chiusura di un insieme di dipendenza funzionali

### Definizione di $F^+$

Dato uno schema relazionale  $R$  e un insieme di dipendenze funzionali  $F$ , la chiusura di  $F$  è l'insieme di tutte le dipendenze funzionali soddisfatte da tutte le tuple in  $R$ . Viene denotata con  $F^+$ .

$$F^+ = \{X \rightarrow Y \mid \forall r \text{ legale} \implies r \text{ SAT } (X \rightarrow Y)\}$$

Da questo possiamo dire che:

$$F \subseteq F^+ \\ Y \subseteq X \implies (X \rightarrow Y) \in F^+$$

### 3.3.1 Assiomi di Armstrong

#### Definizione di $F^A$

Definiamo un altro insieme di dipendenze funzionali  $F^A$  creato con degli assiomi partendo da  $F$ :

- Inclusione iniziale:  $(X \rightarrow Y) \in F \implies (X \rightarrow Y) \in F^A$
- Riflessività:  $Y \subseteq X \subseteq R \implies (X \rightarrow Y) \in F^A$
- Aumento:  $(X \rightarrow Y) \in F^A \implies (XZ \rightarrow YZ) \in F^A \forall Z \in R$
- Transitività:  $\left. \begin{array}{l} (X \rightarrow Y) \in F^A \\ (Y \rightarrow Z) \in F^A \end{array} \right\} \implies X \rightarrow Z \in F^A$

Da questi primi 4 assiomi si ottengono implicitamente altre 3 proprietà:

1. **Unione:**

$$\left. \begin{array}{l} (X \rightarrow Y) \in F^A \\ (X \rightarrow Z) \in F^A \end{array} \right\} \Rightarrow (X \rightarrow YZ) \in F^A$$

**Dimostrazione:**

$$\left. \begin{array}{l} (X \rightarrow Y) \in F^A \xrightarrow{\text{aumento}} (XX \rightarrow XY) \in F^A \\ (X \rightarrow Z) \in F^A \xrightarrow{\text{aumento}} (XY \rightarrow ZY) \in F^A \end{array} \right\} \xrightarrow{\text{trans}} (X \rightarrow YZ) \in F^A$$

2. **Decomposizione:**

$$\left. \begin{array}{l} (X \rightarrow Y) \in F^A \\ Z \subseteq Y \end{array} \right\} \Rightarrow (X \rightarrow Z) \in F^A$$

**Dimostrazione:**

$$\left. \begin{array}{l} Z \subseteq Y \xrightarrow{\text{riff}} (Y \rightarrow Z) \in F^A \\ (X \rightarrow Y) \in F^A \end{array} \right\} \xrightarrow{\text{trans}} (X \rightarrow Z) \in F^A$$

3. **Pseudotrasitività:**

$$\left. \begin{array}{l} (X \rightarrow Y) \in F^A \\ (WY \rightarrow Z) \in F^A \end{array} \right\} \Rightarrow (WX \rightarrow Z) \in F^A$$

**Dimostrazione:**

$$\left. \begin{array}{l} (X \rightarrow Y) \in F^A \xrightarrow{\text{aumento}} (WX \rightarrow WY) \in F^A \\ (WY \rightarrow Z) \in F^A \end{array} \right\} \xrightarrow{\text{trans}} (WX \rightarrow Z) \in F^A$$

## 3.4 Chiusura di un insieme di attributi

### Definizione di $X^+$

Dato uno schema relazionale  $R$  con  $F$  insieme delle dipendenze su  $R$  e  $X \subset R$  la chiusura di  $X$  rispetto ad  $F$  definita come  $X_F^+$  o  $X^+$ :

$$X_F^+ = \{A | (X \rightarrow A) \in F^A\}$$

Ovviamente per riflessività:

$$X \subseteq X_F^+$$

Dato uno schema relazionale  $R$  con  $F$  insieme delle dipendenze funzionali su  $R$ :

$$(X \rightarrow Y) \in F^A \iff Y \subseteq X^+$$

**Dimostrazione:**

$$Y = A_1, A_2, \dots, A_n$$

- $Y \subseteq X^+ \implies (X \rightarrow A_i) \in F^A \quad \forall A_i \in Y \xrightarrow{\text{unione}} (X \rightarrow Y) \in F^A$
- $(X \rightarrow Y) \in F^A \xrightarrow{\text{decomp}} (X \rightarrow A_i) \in F^A \quad \forall A_i \in Y \implies A_i \in X^+ \forall i \implies Y \subseteq X^+$

### 3.4.1 Calcolare la chiusura di $X$

Dato uno schema relazionale  $R$  con  $F$  insieme della dipendenze funzionali,  $X^+$  si calcola tramite un algoritmo:

---

**Algoritmo:** Calcolo di  $X^+$

---

**Input:**

- $R$ : schema
- $F$ : insieme di dipendenze
- $X$ : insieme di attributi

**def** CalcolaChiusura( $R, F, X$ ):

```

  Z=X
  S = {A| $\exists(Y \rightarrow V) \in F, Y \subseteq Z, A \in V$ }
  while S  $\not\subseteq$  Z :
    Z=Z $\cup$ S
    S = {A| $\exists(Y \rightarrow V) \in F, Y \subseteq Z, A \in V$ }
  return Z

```

---

**Esempio:**

$R = ABCDEH$

$F = \{AB \rightarrow CD, EH \rightarrow D, D \rightarrow H\}$

Seguendo l'algoritmo calcoliamo la chiusura di  $A, D, AB$ :

- $A^+ = A$
- $D^+ = DH$
- $AB^+ = ABCDH$

### 3.4.2 Dimostrazione dell'algoritmo

Dato un insieme di attributi  $X$ , l'insieme  $Z$  uscito dall'algoritmo è uguale alla chiusura di  $X$ :

$$Z = X^+$$

**Dimostrazione per doppia inclusione:**

Definisco:

$$Z_i = \left\{ Z \text{ dopo } i \text{ cicli while} \right\} \implies Z_{i+1} = Z_i \cup S_i$$

$$Z_f = Z_{\text{finale}}$$

1.  $Z_f \subseteq X^+$

Dimostro per induzione che se  $Z_i \subseteq X^+$  allora anche  $Z_{i+1} \subseteq X^+$ :

1.1 Caso base:  $i = 0$

$$Z_0 = X \subseteq X^+$$

1.2 Passo induttivo:

$$Z_i \subseteq X^+ \implies Z_{i+1} \subseteq X^+$$

1.3 Dimostrazione induttiva:

$\forall A \in Z_{i+1}$  ci sono due casi possibili:

- $A \in Z_i \xrightarrow{\text{passo induttivo}} A \in X^+$
- $A \in S_i \implies \begin{cases} \exists(Y \rightarrow V) \in F \\ Y \subseteq Z_i \\ A \in V \end{cases} \implies Y \subseteq Z_i \subseteq X^+ \implies$   
 $(X \rightarrow Y) \in F^A \xrightarrow{\text{trans}} (X \rightarrow V) \in F^A \implies V \subseteq X^+ \implies$   
 $A \in X^+$

2.  $X^+ \subseteq Z_f$

	$Z_f$	$R \setminus Z_f$
$t_1$	11...1	01...0
$t_2$	11...1	10...1

uguali su  $Z_f$

Devo controllare se l'istanza è legale per ogni dipendenza di  $F$ .

$\forall (V \rightarrow W) \in F$  ci sono due casi possibili:

- $V \cap R \setminus Z_f \neq \emptyset \implies t_1[V] \neq t_2[V]$
- $V \subseteq Z_f \implies W \subseteq S_f \subseteq Z_f \implies t_1[W] = t_2[W]$

Ora che ho dimostrato che l'istanza è legale:

$$\forall A \in X^+ \implies (X \rightarrow A) \in F^A \implies X = Z_0 \subseteq Z_f \implies A \in Z_f$$

### 3.5 $F^+ = F^A$

Dato uno schema relazionale  $R$  con  $F$  insieme delle dipendenze funzionali:

$$F^+ = F^A$$

**Dimostrazione per doppia inclusione:**

1.  $F^A \subseteq F^+$

Preso una dipendenza funzionale  $(X \rightarrow Y) \in F^A$  definisco:

$$F_i^A = \begin{cases} \text{dipendenze funzionali} \\ \text{ottenute con } i \\ \text{applicazioni degli assiomi} \\ \text{di Armstrong} \end{cases}$$

$$F_0^A \subseteq F_1^A \subseteq F_2^A \subseteq \dots \subseteq F^A$$

Per induzione devo dimostrare che se una dipendenza ottenuta con  $i$  applicazioni degli assiomi di Armstrong appartiene ad  $F^+$  allora anche la dipendenza ottenuta con  $i + 1$  applicazioni degli assiomi di Armstrong appartiene ad  $F^+$ :

1.1 Caso base:  $i = 0$

$$F_0^A = F \subseteq F^+$$

1.2 Passo induttivo:

$$F_i^A \subseteq F^+ \implies F_{i+1}^A \subseteq F^+$$

1.3 Dimostrazione induttiva:

L'ultimo passaggio (per passare da  $i$  a  $i + 1$ ) può essere qualsiasi dei 3 assiomi:

- Riflessività:

$$Y \subseteq X \implies (X \rightarrow Y) \in F_{i+1}^A$$

$$t_1[X] = t_2[X] \implies t_1[Y] = t_2[Y] \implies (X \rightarrow Y) \in F^+$$

- Aumento:

$$(V \rightarrow W) \in F_i^A \implies (VZ \rightarrow WZ) \in F_{i+1}^A$$

$$t_1[VZ] = t_2[VZ] \implies \begin{cases} t_1[V] = t_2[V] \\ t_1[Z] = t_2[Z] \end{cases} \implies \begin{cases} t_1[W] = t_2[W] \\ t_1[Z] = t_2[Z] \end{cases} \implies$$

$$t_1[WZ] = t_2[WZ] \implies$$

$$(VZ \rightarrow WZ) \in F^+$$

- Transitività:

$$\left. \begin{array}{l} (X \rightarrow Y) \in F_i^A \\ (Y \rightarrow Z) \in F_i^A \end{array} \right\} \implies (X \rightarrow Z) \in F_{i+1}^A$$

$$t_1[X] = t_2[X] \implies t_1[Y] = t_2[Y] \implies t_1[Z] = t_2[Z] \implies (X \rightarrow Z) \in F^+$$

$$2. F^+ \subseteq F^A$$

	$X^+$	$R \setminus X^+$
$t_1$	11...1	01...0
$t_2$	11...1	10...1

uguali su  $X^+$

Devo controllare se l'istanza è legale per ogni dipendenza di  $F$ .

$\forall (V \rightarrow W) \in F$  ci sono due casi possibili:

- $V \cap R \setminus X^+ \neq 0 \implies t_1[V] \neq t_2[V]$
- $\left. \begin{array}{l} V \subseteq X^+ \implies (X \rightarrow V) \in F^A \\ (V \rightarrow W) \in F^A \end{array} \right\} \xRightarrow{\text{trans}} (X \rightarrow W) \in F^A \implies W \subseteq X^+$

Ora che ho dimostrato che l'istanza è legale:

$$\forall (X \rightarrow Y) \in F^+ \implies Y \subseteq X^+ \implies (X \rightarrow Y) \in F^A$$

## 3.6 Chiavi di uno schema

### Definizione di chiave

Una chiave è un insieme di attributi  $X \subseteq R$  in cui:

- $(X \rightarrow R) \in F^+$
- $\nexists X' \subseteq X | (X' \rightarrow R) \in F^+$

# 4

## Terza forma normale

### Definizione di schema in 3NF

Dato uno schema relazionale  $R$  con  $F$  insieme delle dipendenze funzionali su  $R$ ,  $R$  è in **terza forma normale** (3NF) se:

$$\forall (X \rightarrow A) \in F^+, A \notin X$$

Una dipendenza è in 3NF se  $X$  contiene una chiave o  $Y$  è contenuto in una chiave.

$$X \supseteq K \vee Y \subseteq K$$

Per controllare se uno schema è scritto in 3NF devo controllare tutte le dipendenze ottenute decomponendo quelle in  $F$  in modo che il dipendente sia un attributo singolo, basta che una non vada bene e tutto lo schema non è scritto in 3NF.

Possiamo dire anche che uno schema è in 3NF se non contiene dipendenze **parziali** né dipendenze **transitive**:

$$\begin{cases} F = \{AB \rightarrow C, B \rightarrow C\} \\ K = AB \end{cases} \implies B \rightarrow C \text{ è una dipendenza parziale perché dipende parzialmente da una chiave}$$

$$\begin{cases} F = \{A \rightarrow B, B \rightarrow C\} \\ K = A \end{cases} \implies B \rightarrow C \text{ è una dipendenza transitiva perché dipende in modo transitivo da una chiave}$$

### Esempi:

$R = ABCD$

$F = \{A \rightarrow B, B \rightarrow CD\}$

$A$  è l'unica chiave perché determina  $B$  e per transitività anche  $CD$

Devo controllare:

- $A \rightarrow B$ : va bene perché  $A$  è chiave
- $B \rightarrow C$ : non va bene perché  $B$  non è chiave e  $C$  non è contenuto in una chiave
- $B \rightarrow D$

$R = ABCD$

$F = \{AC \rightarrow B, B \rightarrow AD\}$

$AC$  e  $BC$  sono chiavi

Devo controllare:

- $AC \rightarrow B$ : va bene perché  $AC$  è chiave
- $B \rightarrow A$ : va bene perché  $A$  è contenuto nella chiave  $AC$
- $B \rightarrow D$ : non va bene perché  $B$  non è chiave e  $D$  non è contenuto in una chiave

## 4.1 Forma normale di Boyce-Codd

### Definizione di schema in BCNF

Dato uno schema relazionale  $R$  con  $F$  insieme delle dipendenze funzionali,  $R$  è in Boyce-Codd NF (BCNF) se tutti i determinanti contengono una chiave.

$$(X \rightarrow Y) \in F^+ | K \subseteq X$$

Tutti gli schemi in Boyce-Codd NF sono anche in 3NF ma non è sempre vero il contrario. A differenza della 3NF non è sempre possibile decomporre uno schema non in BCNF in sottoschemi in BCNF preservando tutte le dipendenze.

## 4.2 Trasformare uno schema in 3NF

Uno schema non in 3NF può essere decomposto in un insieme di schemi in 3NF che devono rispettare delle regole:

- devono preservare le dipendenze funzionali applicate ad ogni istanza dello schema originale (preservare  $F$ )
- bisogna poter ricostruire tramite join naturale ogni istanza legale dello schema originale senza aggiungere informazioni (join senza perdita)

### Esempio:

$F = \{A \rightarrow B, B \rightarrow C\}$  non in forma normale perché  $B \rightarrow C$  è una dipendenza transitiva  
Posso scomporlo in due modi:

- $\begin{cases} R_1 = AB \text{ con } F_1 = \{A \rightarrow B\} \\ R_2 = BC \text{ con } F_2 = \{B \rightarrow C\} \end{cases}$
- $\begin{cases} R_1 = AB \text{ con } F_1 = \{A \rightarrow B\} \\ R_2 = AC \text{ con } F_2 = \{A \rightarrow C\} \end{cases}$

Il secondo modo non va bene perché quando li mettiamo insieme la dipendenza  $B \rightarrow C$  non è soddisfatta.



## 4.3 Preservare F

### Definizione di decomposizione

Dato uno schema relazionale  $R$ , una decomposizione è un insieme di sottinsiemi di  $R$  tali che:

$$R = \bigcup_{i=1}^k R_i$$

Questa non è una partizione di  $R$  perché i sottoschemi non sono disgiunti.

Presa un'istanza  $r$  l'istanza di un sottoschema  $r_i$ :

$$r_i = \pi_{R_i}(r)$$

### Decomposizione che preserva F

Dato uno schema relazionale  $R$ , una decomposizione  $\rho = R_1, \dots, R_k$  preserva  $F$  se:

$$F \equiv G = \bigcup_{i=1}^k \pi_{R_i}(F)$$

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

Una singola dipendenza  $(X \rightarrow Y) \in F$  è preservata se  $Y \subseteq X_G^+$ .

### 4.3.1 Teorema sulla chiusura

#### Teorema sulla chiusura

Dati due insiemi di dipendenze funzionali  $F, G$ :

$$F \subseteq G^+ \iff F^+ \subseteq G^+$$

**Dimostrazione per doppia implicazione:**

$$1. \quad F^+ \subseteq G^+ \implies F \subseteq G^+ \\ F \subseteq F^+ \subseteq G^+ \implies F \subseteq G^+$$

$$2. \quad F \subseteq G^+ \implies F^+ \subseteq G^+$$

Scrivo  $g \xrightarrow{A} f$  per dire che posso ottenere  $f$  applicando gli assiomi di Armstrong su  $g$

$$\forall f \in F \implies \exists g \mid g \xrightarrow{A} f \implies G \xrightarrow{A} F \xrightarrow{A} F^+ \implies F^+ \subseteq G^+$$

## 4.4 Calcolare $X_G^+$

Dato uno schema relazionale  $R$  con  $F$  insieme della dipendenze funzionali e  $G$  rispetto ad  $F$ ,  $X^+$  si calcola tramite un algoritmo (non conosciamo  $G$ ):

---

**Algoritmo:** Calcolo di  $X_G^+$

---

**Input:**

- $R$ : schema
- $F$ : insieme di dipendenze
- $X$ : insieme di attributi
- $\rho$ : decomposizione

**def** CalcolaChiusuraG( $R, F, X, \rho$ ):

```

  Z=X
  S={}
  for i in range(1,k) :
    |  $S = S \cup (Z \cap R_i)_F^+ \cap R_i$ 
  while  $S \not\subseteq Z$  :
    | Z=Z $\cup$ S
    | for i in range(1,k) :
    | |  $S = S \cup (Z \cap R_i)_F^+ \cap R_i$ 
  return Z

```

---

**Dimostrazione per doppia inclusione:**

1.  $Z_f \subseteq X_G^+$

1.1 Caso base:  $i = 0$

$$Z_0 = X \subseteq X_G^+$$

1.2 Passo induttivo:

$$Z_i \subseteq X_G^+ \implies Z_{i+1} \subseteq X_G^+$$

1.3 Dimostrazione induttiva:

$\forall A \in Z_{i+1}$  ci sono due casi possibili:

- $A \in Z_i \xRightarrow{\text{passo induttivo}} A \in X_G^+$

$$\begin{aligned}
 & \bullet A \in S_i \implies A \in [(Z_i \cap R_j)_F^+ \cap R_j] \implies \left\{ \begin{array}{l} A \in R_j \\ A \in (Z_i \cap R_j)_F^+ \end{array} \right. \implies (Z_i \cap R_j \rightarrow A) \in F^A \\
 & \implies (Z_i \cap R_j \rightarrow A) \in \pi_{R_j}(F) \subseteq G \implies \left\{ \begin{array}{l} (X \rightarrow Z_i \cap R_j) \in G^A \\ (Z_i \cap R_j \rightarrow A) \in G^A \end{array} \right\} \xRightarrow{\text{trans}} (X \rightarrow A) \in G^A \\
 & \implies A \in X_G^+
 \end{aligned}$$

2.  $X_G^+ \subseteq Z_f$

$X \subseteq Z_f \implies X_G^+ \subseteq (Z_f)_G^+$ , se supponiamo di conoscere  $G$  possiamo applicare l'algoritmo per calcolare  $X_F^+$  ma su  $G$ .

Ogni elemento  $A \in S$  è anche in  $Z$  perché:

Se esiste  $(Y \rightarrow V) \in G \implies YV \subseteq R_i \implies Y \subseteq (R_i \cap Z_f) \xrightarrow{\text{riff}}$

$(R_i \cap Z_f \rightarrow Y) \in G^A \xrightarrow{\text{decomp}} (R_i \cap Z_f \rightarrow A) \in G^A \implies$

$A \in (R_i \cap Z_f)^+ \cap R_i \implies A \in Z_f$

Quindi  $Z_f = (Z_f)_G^+ \implies X_G^+ \subseteq Z_f$

## 4.5 Equivalenza tra insiemi di dipendenze funzionali

$$F^+ = G^+$$

Dati due insiemi di dipendenze funzionali  $F, G$ :

$$F \equiv G \iff F^+ = G^+$$

$G$  e  $F$  non sono uguali.

$$\left. \begin{array}{l} F^+ \subseteq G^+ \\ G^+ \subseteq F^+ \end{array} \right\} \iff \left\{ \begin{array}{l} F \subseteq G^+ \\ G \subseteq F^+ \end{array} \right.$$

**Dimostrazione tramite doppia inclusione:**

1.  $G \subseteq F^+$

$$G = \bigcup_{i=0}^k \pi_{R_i}(F) \implies G \subseteq F^+$$

2.  $F \subseteq G^+$

per dimostrarlo utilizziamo un algoritmo:

---

**Algoritmo:** Calcolo se  $F \subseteq G^+$

---

**Input:**

- R: schema
- F: insieme di dipendenze
- X: insieme di attributi
- $\rho$ : decomposizione

**def** IsSottoinsieme(R, F, X,  $\rho$ ):

```

  for (X → Y) in F :
    X_G^+ = CalcolaChiusuraG(R, F, X,  $\rho$ )
    if Y ∉ X_G^+ :
      return False
  return True

```

---

## 4.6 Trovare le chiavi di uno schema

Per trovare la chiavi bisogna trovare un attributo o un insieme di attributi tali che la sua chiusura sia  $R$  e non ci siano sottoinsiemi tali che la loro chiusura sia  $R$ :

$$X = K \iff X^+ = R$$

$$\nexists X' \subseteq X | (X')^+ = R$$

Ci sono delle chiusure che si possono saltare:

- Se un attributo non è determinato da nulla appartiene a tutte le chiavi
- Se un attributo non determina niente, ma è solo determinato non appartiene a nessuna chiave

Per provare a calcolarle in modo facile seguiamo dei passaggi:

1. Calcoliamo  $X_i = R - (W - V)$  per ogni dipendenza  $V \rightarrow W$
2. Calcoliamo  $X = (X_1 \cap \dots \cap X_n)$
3. Calcoliamo  $X^+$  e se il risultato è:
  - $R$  allora  $X$  è l'unica chiave
  - diverso da  $R$  allora  $X$  non è chiave ed esiste più di una chiave

### 4.6.1 Teorema di unicità della chiave

#### Teorema di unicità della chiave

Dato uno schema  $R$ , possiamo controllare se una chiave è unica scrivendo:

$$X = \bigcap_{(V \rightarrow W) \in F} R - (W - V)$$

Calcoliamo  $X^+$  e se:

- $X^+ = R \implies X$  unica chiave
- $X^+ \neq R \implies$  esiste più di una chiave e  $X$  non è superchiave

**Esempio:**

$R = ABCDE$

$F = \{AB \rightarrow C, AC \rightarrow B, D \rightarrow E\}$

$X = (ABCDE - (C - AB)) \cap (ABCDE - (B - AC)) \cap (ABCDE - (E - D)) = ABDE \cap ACDE \cap ABCD = AD$

$X^+ = AD^+ = ADE \neq R \implies$  esisto più chiavi

## 4.7 Join senza perdite

### Decomposizione don join senza perdite

Dato uno schema relazionale  $R$ , una decomposizione  $\rho = R_1, R_2, \dots, R_k$  ha un join senza perdita se per ogni istanza  $r$  di  $R$ :

$$r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \dots \bowtie \pi_{R_k}(r) = m_\rho(r)$$

### Proprietà di una decomposizione

La decomposizione di  $R$ ,  $m_\rho(r)$  ha delle proprietà:

- $r \subseteq m_\rho(r)$
- $\pi_{R_i}(m_\rho(r)) = \pi_{R_i}(r)$
- $m_\rho(m_\rho(r)) = m_\rho(r)$

**Dimostrazione:**

1.  $\forall t \in r \implies t \in (t[R_1] \bowtie \dots \bowtie t[R_k]) \subseteq (\pi_{R_1}(r) \bowtie \dots \bowtie \pi_{R_k}(r)) = m_\rho(r)$
2.  $\pi_{R_i}(m_\rho(r)) = \pi_{R_i}(r)$ 
  - $\pi_{R_i}(r) \subseteq \pi_{R_i}(m_\rho(r)) \implies$  punto 1
  - $\pi_{R_i}(m_\rho(r)) \subseteq \pi_{R_i}(r)$   
 $\forall t \in \pi_{R_i}(m_\rho(r)) \implies \exists t' \in m_\rho(r) | t'[R_i] = t \implies \exists t_i \in r | t_i[R_i] = t'[R_i] \implies$   
 $t = t'[R_i] = t_i[R_i] \in \pi_{R_i}(r)$
3.  $m_\rho(m_\rho(r)) = \pi_{R_1}(m_\rho(r)) \bowtie \pi_{R_2}(m_\rho(r)) \bowtie \dots \bowtie \pi_{R_k}(m_\rho(r)) = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \dots \bowtie \pi_{R_k}(r) = m_\rho(r)$

### 4.7.1 Controllare un join senza perdite

Per controllare se una decomposizione ha un join senza perdite si usa una serie di passaggi:

1. Si costruisce una tabella con  $|R|$  numero di colonne e  $k$  righe
2. In ogni casella della tabella con colonna  $j$  e riga  $i$  scriviamo  $\begin{cases} a & A_j \in R_i \\ b_i & A_j \notin R_i \end{cases}$
3. Per ogni dipendenza  $(X \rightarrow Y) \in F$  se ci sono due tuple  $t_1[X] = t_2[X]$  ma  $t_1[Y] \neq t_2[Y]$  allora se una delle due è uguale ad  $a$  allora cambia anche l'altra in  $a$  mentre se nessuno dei due è uguale a  $a$  allora cambia uno delle due facendola diventare uguale all'altra.

4. Se in qualunque momento una riga avesse tutte  $a$  allora possiamo fermarci e sapere che ha un join senza perdite
5. Alla fine del controllo di tutte le dipendenze se nessuna riga ha tutte  $a$  allora ripartiamo a controllare su tutte le dipendenze, fino a quando un'intero ciclo di dipendenze non cambia nulla sulla tabella

I passaggi precedenti sono derivati da un algoritmo: **Dimostrazione per assurdo:**

---

**Algoritmo:** Calcolo Join senza perdite

---

**Input:**

- R: schema
- F: insieme di dipendenze
- $\rho$ : decomposizione

**def** JoinLossless(R,F, $\rho$ ):

```

r=Tabella
changed=True
while changed and  $\nexists t \text{ in } r | t = \{a...a\}$  :
    changed=False
    for  $(X \rightarrow Y)$  in F :
        if  $t1[X] == t2[Y]$  and  $t1[X] != t2[Y]$  :
            changed=True
            for A in Y :
                if  $t1[A] == a$  :
                    t2[A]=a
                else
                    t1[A]=t2[A]
    if  $\exists t \text{ in } r | t = \{a...a\}$  :
        return True
return False

```

---

Supponiamo che  $\rho$  abbia un join senza perdita  $m_\rho(r) = r$  ma  $r$  non contenga nessuna tupla con solamente  $a$ .

Visto che non succede mai che  $a \rightarrow b_i$  allora  $\exists t_i \in \pi_{R_i}(r) | t_i = "a...a" \forall i$ , precisamente nella riga corrispondente al sottoschema  $R_i$ .

Quindi  $m_\rho(r)$  contiene una riga con tutte  $a$ .

## 4.8 Copertura minimale

### Definizione di copertura minimale

Dato un insieme di dipendenze funzionali  $F$ , una copertura minimale di  $F$  è un insieme di dipendenze funzionali  $G$  tale che:

1. Tutte le dipendenze in  $G$  hanno un attributo singolo come dipendente (a destra)
2.  $\forall (X \rightarrow Y) \in G \nexists X' \subset X | G \equiv [G - \{(X \rightarrow Y)\}] \cup \{(X' \rightarrow Y)\}$
3.  $\forall (X \rightarrow Y) \in G \implies G \not\equiv G \setminus \{(X \rightarrow Y)\}$

Preso un insieme  $F$  possiamo calcolare la copertura minimale tramite 3 passaggi:

1. Con la regola di decomposizione divido tutti i dipendenti ad attributi singoli
2. Controllo se le dipendenze possono essere ridotte, cioè  $\exists X' \subset X | (X' \rightarrow Y)$ , in caso si possa sostituire la dipendenza  $(X \rightarrow Y)$  con  $(X' \rightarrow Y)$ , per farlo devo calcolare la chiusura di tutti i sottoinsiemi di  $X$  e controllare se contengono  $Y$
3. Controllo per ogni dipendenza se  $(X)_{F \setminus \{(X \rightarrow Y)\}}^+$  ( $X^+$  calcolato sull'insieme  $F$  escludendo la dipendenza stessa) contiene  $Y$ , se si elimina la dipendenza  $(X \rightarrow Y)$

## 4.9 Algoritmo di decomposizione

Dato uno schema relazionale  $R$  con insieme di dipendenze funzionali  $F$  esiste sempre una decomposizione  $\rho$  tale che:

1. Ogni sottoschema è in 3NF
2. Preserva  $F$
3. Ha un join senza perdita

Per trovare la decomposizione possiamo usare un algoritmo composto da 3 passaggi:

1. Aggiungere a  $\rho$  un sottoschema  $S$  con tutti gli attributi che non appaiono in nessuna dipendenza
2. Tolgo questi attributi a  $R$  e controllo se c'è una dipendenza che contiene tutti gli attributi rimasti in  $R - S$ , se c'è aggiungo tutti questi attributi come unico sottoschema a  $\rho$
3. Se non c'è per ogni dipendenza aggiungo come sottoschema tutti gli attributi che la compongono
4. Per avere un join senza perdita un sottoschema deve contenere una chiave, se non c'è allora dobbiamo aggiungerne una.

I passaggi precedenti sono derivati da un algoritmo: **Dimostrazione:**

---

**Algoritmo:** Calcolo decomposizione

---

**Input:**

- R: schema
- F: copertura minimale

**def** CalcolaDecomposizione(R,F):

---

```

  S={}
  ρ = {}
  S={A|∀(X → Y) ∈ F, A ∈ X, A ≠ Y}
  if S ≠ ∅ :
    R=R-S
    ρ = ρ ∪ {S}
  if ∃(X → Y) ∈ F | X ∪ Y == R :
    ρ = ρ ∪ {R}
  else
    for (X → Y) in F :
      ρ = ρ ∪ {X ∪ Y}
  return ρ

```

---

## 1. Preserva F

$$G = \bigcup_{i=1}^k \pi_{R_i}(F)$$

$$\forall (X \rightarrow Y) \in F \implies XY \in \rho \implies (X \rightarrow Y) \in G \implies F \equiv G$$

## 2. I sottoschemi sono in 3NF

Dividiamo i 3 casi possibili:

- $S \in \rho \implies$  tutti gli attributi fanno parte di una chiave quindi è in 3NF
- $R \in \rho \implies$  questa dipendenza ha forma  $(R - A) \rightarrow A \implies R - A$  è la chiave, quindi  $\forall (Y \rightarrow B) \in F \implies \begin{cases} B \neq A \implies B \in R - A \implies B \text{ primo} \\ B = A \implies Y = R - A \implies Y \text{ superchiave} \end{cases}$
- $XA \in \rho \implies$  essendo F copertura minimale  $\implies X$  è chiave di  $XA \implies \forall (Y \rightarrow B) \in F | YB \subseteq XA \implies \begin{cases} B = A \implies Y = X \implies Y \text{ superchiave} \\ B \neq A \implies B \in X \implies B \text{ primo} \end{cases}$



# 5

## Organizzazione fisica

L'organizzazione fisica di una database è divisa in più settori:

1. Hardware di archiviazione e progettazione fisica
  - 1.1 Gerarchia di archiviazione
  - 1.2 Interno di un hard disk
  - 1.3 Passaggio dai concetti logici a quelli fisici
2. Organizzazione record
  - 2.1 Puntatori
  - 2.2 Liste
3. Organizzazione file
  - 3.1 Organizzazione tramite file heap
  - 3.2 Organizzazione sequenziale
  - 3.3 Organizzazione tramite file hash (casuale)
  - 3.4 Organizzazione sequenziale indicizzata
  - 3.5 Organizzazione in lista
  - 3.6 Indici secondari e file invertiti
  - 3.7 B-Trees

### 5.1 Hardware di archiviazione e progettazione fisica

#### 5.1.1 Gerarchia di archiviazione

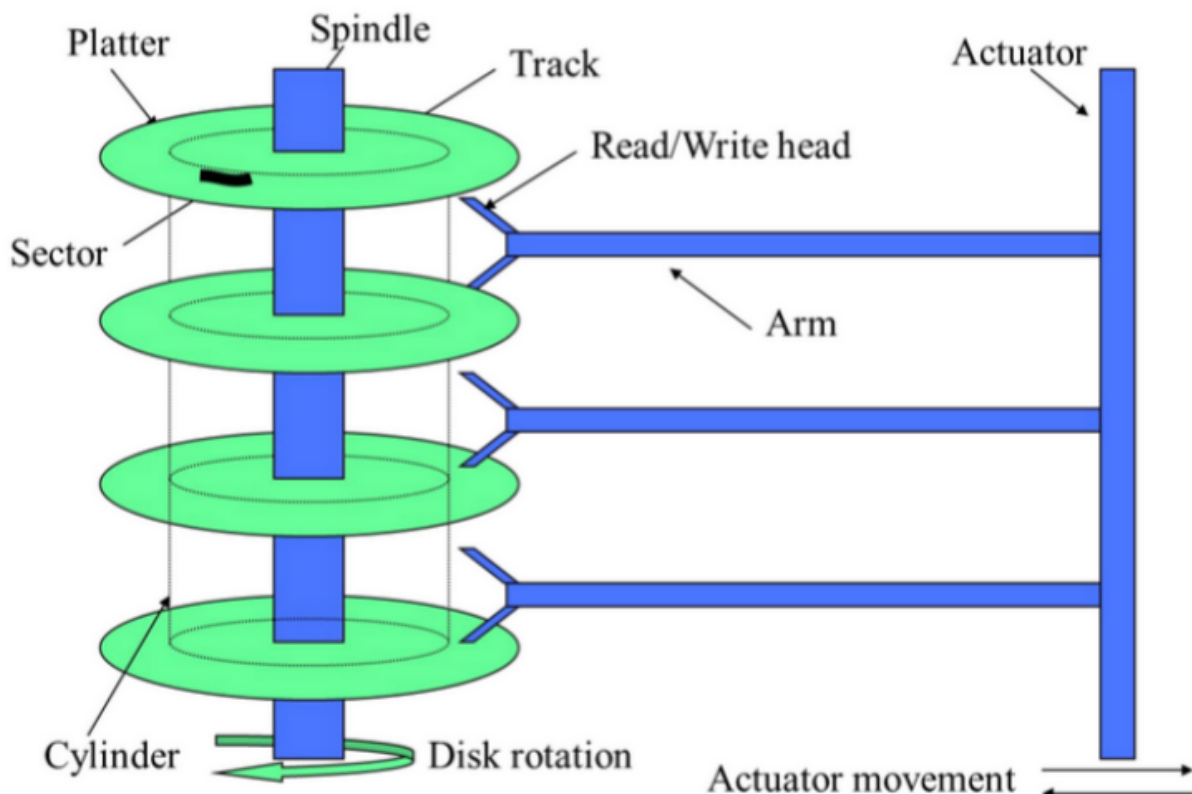
L'**Archiviazione primaria** di un DBMS contiene i buffer del database e i codici in esecuzione delle applicazioni e del DBMS. Essa è composta da CPU, Cache e Memoria principale (composta a sua volta da vari banchi di RAM).

L'**Archiviazione secondaria** si occupa dell'archiviazione permanente dei dati attraverso vari Hard Disk (HDD) e Solid-state drive (SSD), memorizzando i file fisici contenenti i dati del database.

### 5.1.2 Interno di un hard disk

Un hard disk è costituito da:

- Controller: gestisce gli altri componenti e le richieste di lettura o scrittura, mettendole in coda
- Piatti circolari magnetici collegati ad uno spindle (asse) che gira a velocità costante
- Testine di lettura o scrittura posizionate sui bracci di un attuatore, che si muove per posizionare le testine sul punto giusto dei piatti



Per leggere o scrivere un blocco di dati dai piatti bisogna:

- Posizionare l'attuatore, il cui tempo impiegato viene detto **Seek** time
- Attendere la rotazione del disco finché il settore richiesto non si trova sotto la testina di lettura e scrittura, il cui tempo impiegato viene detto **Rotation** time (ROT)
- Trasferire i dati, il cui tempo viene detto **Transfer** time, che dipende dalla grandezza del blocco da leggere (BIS) e dal rateo di trasferimento dei dati (TR), influenzato dalla densità delle particelle magnetiche presenti sul disco e la velocità di rotazione del disco stesso

Il tempo impiegato per completare una lettura o scrittura viene detto **Service** time:

$$\text{Service} = \text{Seek} + \text{ROT} + \frac{\text{BIS}}{\text{TR}}$$

Il tempo impiegato dall'hard disk per restituire la risposta viene detto **Response time** e dipende dal Service time e dal tempo per mettere in coda la richiesta, detto **Queueing time**:

$$\text{Response} = \text{Service} + \text{Queueing}$$

Visto che il tempo di lettura e scrittura dipende dalla posizione delle testine, possiamo dividere il tempo impiegato in due casi:

- **Random Block Access:** il tempo per accedere ad un blocco indipendentemente dalla posizione attuale delle testine

$$T_{\text{RBA}} = \text{Seek} + \frac{\text{ROT}}{2} + \frac{\text{BIS}}{\text{TR}}$$

- **Sequenzial Block Access:** il tempo per accedere ad un blocco con la testina già posizionata nel settore giusto

$$T_{\text{SBA}} = \frac{\text{ROT}}{2} + \frac{\text{BIS}}{\text{TR}}$$

### 5.1.3 Passaggio dai concetti logici a quelli fisici

Dobbiamo trasformare un database dal modello concettuale al modello relazionale e infine alla sua rappresentazione fisica, per capire meglio i termini possiamo usare questa tabella:

Modello concettuale	Modello relazionale	Modello interno
Tipo attributo e valore	Nome colonna e cella	Dati oggetti o campi
Record	Riga o tupla	Record archiviato
Tipo record	Tabella o relazione	File fisico o insieme di dati
Insieme di tipi di record	Insieme di tabelle	Database fisico o insieme di dati
Struttura dati logiche	Chiavi esterne	Strutture di archiviazione

Quindi a livello fisico:

- Un database consiste in un insieme di file
- Un file è un insieme di pagine con dimensione fissa
- Una pagina contiene dei record (tuple logiche)
- Un record contiene più campi, contenenti gli attributi delle tuple

## 5.2 Organizzazione dei file

I file all'interno di un hard disk possono essere organizzati in modo diverso per ottimizzare la velocità con cui si esegue una scrittura o lettura, per far ciò si cerca di aumentare i SBA e di diminuire i RBA più possibile.

### 5.2.1 Organizzazione tramite file heap

L'organizzazione tramite file heap è il modello più basico, in cui ogni record viene inserito alla fine del file, identificandolo con una chiave di ricerca. L'unico modo per cercare un record è tramite la ricerca lineare, il cui tempo medio avendo NBI blocchi è:

- $\lceil \frac{NBI}{2} \rceil$  SBA con chiave univoca
- $NBI \cdot SBA$  con chiave non univoca

### 5.2.2 Organizzazione tramite file sequenziali

L'organizzazione tramite file sequenziali inserisce i record in ordine crescente o decrescente in base alla loro chiave di ricerca, in modo che si possa usare anche la ricerca binaria per trovare un record, quindi il tempo di accesso con NBI blocchi è:

- $\lceil \log_2(NBI) \rceil$  RBA con ricerca binaria
- $\lceil \frac{NBI}{2} \rceil$  SBA con ricerca sequenziale

### 5.2.3 Organizzazione tramite file hash

L'organizzazione tramite file hash utilizza un algoritmo di hashing (di solito si utilizza la funzione  $\text{mod}$ ) per collegare la chiave di ricerca all'indirizzo fisico dove si trova il record. Visto che più di un record potrebbe essere mappato allo stesso valore hash, si utilizzano dei bucket per contenere tutti i record mappati nello stesso valore hash. Nella memoria principale viene salvata una bucket directory, dove ogni entrata corrisponde al puntatore al primo blocco del bucket associato.

Nel caso in cui i record mappati ad un bucket superino la capienza del bucket, il record va in overflow e va gestito, solitamente si usa l'indirizzamento aperto, cioè il record viene archiviato al primo posto disponibile.

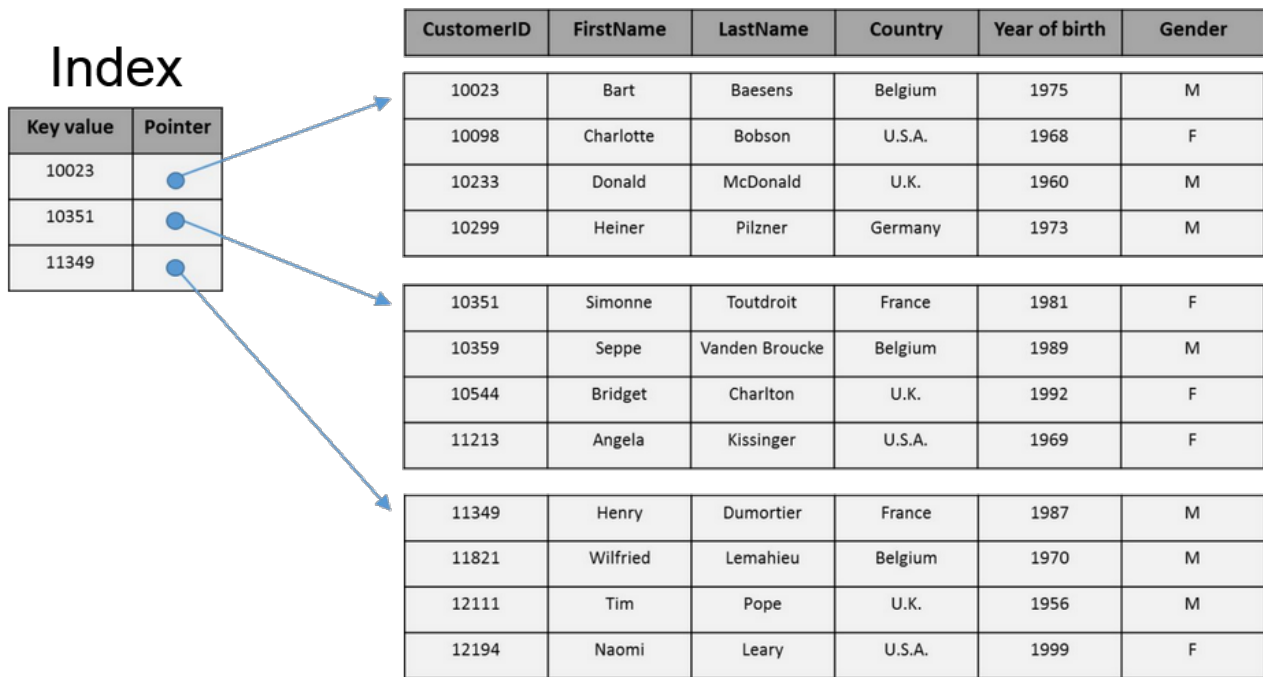
Il tempo di accesso è quindi diverso in base a se il record che ci interessa è in overflow o no:

- Non in overflow: Una RBA per raggiungere il primo blocco del bucket e diverse SBA per trovare il record all'interno del bucket, essendo archiviati in modo sequenziale
- In overflow: Oltre ai passaggi per trovare un record non in overflow dovremo accedere ad altri blocchi in base alla quantità di record in overflow

### 5.2.4 Organizzazione tramite ISAM

L'organizzazione tramite ISAM ha un file principale diviso in partizioni, ognuna con un'entrata del tipo `<Chiave di ricerca del primo record, puntatore indirizzo primo record>` archiviata nel file indice. Il file indice può essere di due tipi:

- **Denso**: esiste un'entrata per ogni possibile valore della chiave di ricerca
- **Sparso**: esiste solo l'entrata del primo record di ogni partizione, quindi ogni entrata fa riferimento ad un gruppo di record



Il tempo di accesso, con NBpFP e NBpFI rispettivamente i blocchi del file principale e del file indice, sarà quindi:

- Ricerca lineare sul file principale:  $\text{NBpFP} \cdot \text{SBA}$
- Ricerca binaria sul file principale:  $\lceil \log_2(\text{NBpFP}) \rceil \text{RBA}$
- Ricerca binaria tramite file indice:  $\lceil \log_2(\text{NBpFI}) \rceil + 1 \text{ RBA}$

### 5.2.5 Organizzazione tramite B-Tree (bilanciato)

#### Albero di ricerca ad $m$ vie

Un albero di ricerca ad  $m$ -vie è simile ad un albero binario di ricerca, con delle diverse proprietà:

- Un nodo ha da 1 fino a  $m - 1$  chiavi
- Il numero di figli va da 0 a  $k + 1$ , in cui  $k$  è il numero di chiavi del nodo, quindi il numero massimo di figli è  $m$
- Le chiavi nel sottoalbero puntato dal puntatore compreso tra due valori  $k$  e  $k'$  possono avere solo valori compresi tra  $k$  e  $k'$

**B-Tree**

Un B-Tree è un albero di ricerca ad  $m$ -vie con altre proprietà aggiuntive:

- Ogni nodo ha massimo  $m$  figli e  $m - 1$  chiavi
- Ogni nodo tranne la radice e le foglie ha minimo  $\lceil \frac{m}{2} \rceil$  figli e  $\lceil \frac{m}{2} \rceil - 1$  chiavi
- La radice ha minimo due figli tranne se è una foglia
- Tutte le foglie sono allo stesso livello

L'organizzazione tramite B-Tree ha un file indice strutturato ad albero in cui ogni nodo corrisponde ad un blocco, carico sempre per almeno metà e con almeno  $\lceil \frac{m}{2} \rceil$  figli.

Ogni blocco ha:

- Delle chiavi di ricerca
- I puntatori ai nodi figli

Nel file principale vengono archiviati i record e tutti i blocchi sono puntati dalle foglie dell'albero. I blocchi del file principale vengono archiviati separatamente e non fanno parte dell'albero.

La ricerca viene effettuata in maniera ricorsiva controllando ogni volta se il valore che stiamo cercando è nel nodo e continuando con il puntatore nell'intervallo corretto. Nel caso in cui il valore che stiamo cercando sia all'interno del nodo sia la chiave  $K_i$  il record corrispondente si accede tramite il puntatore ai dati  $P_{d_i}$ .

Il tempo di accesso è nel caso peggiore uguale all'altezza dell'albero quindi  $O(h)$ .

### Altezza di un B-Tree

Dato un B-Tree avente  $n$  chiavi totali, con  $m$  numero massimo di figli e  $d = \lceil \frac{m}{2} \rceil$  numero minimo di figli, l'altezza dell'albero sarà:

$$\lceil \log_m(n+1) \rceil \leq h \leq \lfloor \log_d(\frac{n+1}{2}) \rfloor + 1$$

#### Dimostrazione:

Il numero massimo di chiavi si ottiene se tutti i nodi hanno  $m$  figli e  $m-1$  chiavi, il numero di chiavi totali allora sarà:

$$n_{max} = (m-1) \sum_{i=0}^{h-1} m^i = (m-1) \frac{m^h - 1}{m - 1} = m^h - 1$$

Il numero minimo di chiavi si ottiene se tutti i nodi hanno  $d$  figli e  $d-1$  chiavi, il numero di chiavi totali allora sarà:

$$n_{min} = 1 + (d-1) \left( 2 \cdot \sum_{i=0}^{h-2} d^i \right) = 1 + (d-1) \left( 2 \frac{d^{h-1} - 1}{d - 1} \right) = 2d^{h-1} - 1$$

Da questo possiamo ricavare che:

$$2d^{h-1} - 1 \leq n \leq m^h - 1 \implies \lceil \log_m(n+1) \rceil \leq h \leq \lfloor \log_d(\frac{n+1}{2}) \rfloor + 1$$

# 6

## Concorrenza

In sistemi con una sola CPU i programmi vengono eseguiti in modo interfogliato, cioè che la CPU può:

- Eseguire alcune istruzioni di un programma
- Sospendere quel programma
- Eseguire istruzioni di altri programmi
- Riprendere ad eseguire il programma sospeso

In un DBMS la risorsa a cui i programmi accedono in modo concorrente è il DB. La concorrenza non crea problemi se vengono eseguite solo operazioni di lettura, invece se vengono eseguite anche operazioni di scrittura la concorrenza va controllata.

### 6.1 Transazioni

#### Definizione di transazione

Una **transazione** è l'esecuzione di una parte di un programma che rappresenta un'unità logica di accesso o modifica del DB.

Le transazioni hanno 4 proprietà, descritte dall'acronimo ACID:

- **Atomicità:** la transazione non può essere eseguita a metà, deve essere eseguita tutta o non essere eseguita per niente
- **Consistenza:** la transazione deve iniziare con un DB in stato consistente (non deve violare vincoli) e lo deve lasciare in stato consistente quando è finita l'esecuzione
- **Isolamento:** la transazione deve essere eseguita in modo indipendente dalle altre e un fallimento non deve interferire con le transazioni successive
- **Durabilità:** una volta che la transazione ha richiesto un **commit work**, i cambiamenti apportati al DB non devono essere persi, per evitare perdite di dati vengono tenuti dei log che annotano tutte le operazioni sul DB



## 6.2 Schedule di transazioni

### Definizione di schedule

Una schedule di un insieme  $T$  di transazioni è l'ordinamento delle operazioni nelle transazioni che conserva l'ordine delle operazioni all'interno delle singole transazioni. Se una transazione ha un'operazione  $O_1$  precedente ad un'operazione  $O_2$  le due operazioni in una schedule potranno essere separate da operazioni di altre transazioni ma non avremo mai  $O_2$  precedente a  $O_1$ .

### Schedule seriale

Uno schedule **seriale** è uno schedule ottenuto permutando le transazioni. Lo schedule seriale quindi eseguirà le transazioni in maniera sequenziale e non interfogliata. Tutti gli schedule seriali sono corretti.

Uno schedule non seriale è corretto solo se è **serializzabile**, cioè è equivalente (non uguale) ad uno schedule seriale.

Due schedule sono **equivalenti** solo se per ogni dato modificato producono valori uguali, dove due valori sono uguali solo se prodotti dalla stessa sequenza di operazioni

**Esempio:**

$T_1$	$T_2$
read(X) $X=X+N$ write(X)	read(X) $X=X-M$ write(X)

$T_1$	$T_2$
read(X) $X=X+N$ write(X)	read(X) $X=X-M$ write(X)

Le due schedule non sono equivalenti nonostante diano lo stesso risultato  $X$ , però sono seriali quindi corrette.

### 6.2.1 Garantire la serializzabilità

Per garantire la serializzabilità è impossibile determinare in anticipo come verranno eseguite le operazioni e non si possono neanche eseguire e poi cancellare tutte le transazioni se lo schedule non è serializzabile. I sistemi quindi utilizzano metodi che garantiscano la serializzabilità, per evitare il bisogno di testare lo schedule ogni volta.

Per garantire la serializzabilità bisogna:

- Imporre dei **protocolli**, cioè delle regole alle transazioni
- Usare i **timestamp**, degli identificatori delle transazioni generati dal sistema, in base ai quali vengono ordinate le operazioni

**Item**

Gli **item**, usati dai metodi precedenti, sono unità il cui accesso è controllato e le cui dimensioni sono definite per far sì che in media una transazione acceda a pochi item. La dimensione degli item viene detta **granularità** del sistema. Una granularità grande permette una gestione efficiente della concorrenza, mentre una granularità piccola permette l'esecuzione concorrente di più transazioni, rischiando però di sovraccaricare il sistema.

## 6.3 Lock

**Definizione di lock**

Un lock è un privilegio di accesso ad un item realizzato tramite una variabile associata all'item, che descrive lo stato dell'item rispetto alle operazioni che possono essere eseguite su di esso.

Un lock:

- Viene richiesto da una transazione tramite un'operazione di **locking**, in cui se la variabile di quell'item è **unlocked** (cioè nessuna altra transazione ha un lock su quell'item) la transazione può accedere all'item e assegnare alla variabile il valore **locked**.
- Viene rilasciato tramite un'operazione di **unlocking**, in cui la variabile assegnata all'item viene cambiata da **locked** a **unlocked**.

Nel periodo tra l'operazione di **locking** e quella di **unlocking** la transazione mantiene un lock sull'item.

Uno schedule è legale se una transazione effettua un **locking** ogni volta che legge o scrive un item e rilascia tutti i lock che ha ottenuto.

## 6.4 Lock binario

Un lock binario ha solo due valori, **locked** e **unlocked**. Le transazioni richiedono l'accesso ad un item  $X$  tramite un  $lock(X)$  e rilasciano l'item permettendo l'accesso ad altre transazioni tramite  $unlock(X)$ .

**Esempio:**

$T_1$	$T_2$
lock(X) read(X) $X=X+N$ write(X) unlock	lock(X) read(X) $X=X+M$ write(X) unlock(X)
lock(X) read(Y) $Y=Y+N$ write(Y) unlock(Y)	

### 6.4.1 Equivalenza di schedule con lock binario

Per verificare l'equivalenza di due schedule con lock binario utilizziamo un modello che non conta le specifiche operazioni ma solo quelle rilevanti per controllare la sequenza, cioè quelle di lock e unlock. Per fare questo associamo ad ogni operazione di *unlock* una funzione che ha come argomenti gli item letti dalla transazione prima dell'operazione di *unlock*.

Due schedule sono equivalenti se le formule che danno il valore sono uguali per **tutti gli item**.

### 6.4.2 Algoritmo per testare la serializzabilità

Per controllare se un schedule  $S$  è serializzabile:

1. Crea un grafo  $G$  in cui gli archi con l'etichette di un item  $X$  tra  $T_i$  e  $T_j$  indicano che  $T_j$  è la prima transazione che fa *lock*( $X$ ) dopo che  $T_i$  fa *unlock*( $X$ )
2. Se  $G$  ha un ciclo allora  $S$  non è serializzabile, sennò basta usare l'ordinamento topologico per ottenere uno schedule seriale  $S'$  equivalente a  $S$ . L'ordinamento topologico si ottiene eliminando ricorsivamente i nodi che non hanno archi entranti insieme ai suoi archi uscenti

Quindi:

$$G \text{ aciclico} \iff S \text{ serializzabile}$$

## 6.5 Lock a due fasi

Una transazione obbedisce al protocollo di locking a due fasi se prima esegue tutte le operazioni di lock e poi tutte le operazioni di unlock. Il lock a due fasi non è esclusivo del lock a due valori, potremmo avere lock a due fasi con tre valori.

**Teorema sul lock a due fasi**

Dato un insieme di transazioni  $T$  ogni schedule è serializzabile se tutte le transazioni sono a due fasi.

**Dimostrazione per assurdo:**

Supponiamo di avere uno schedule  $S$  composto solo da transazioni a due fasi ma il grado  $G$  sia ciclico.

$G$  crea uno schedule  $S$ :

$T_1$	$T_2$	$\dots$	$T_k$
unlock( $X_1$ )	lock( $X_1$ ) unlock( $X_2$ )		unlock( $X_k$ )
lock( $X_k$ )			

$T_1$  quindi non è a due fasi, il che è una contraddizione.

## 6.6 Lock a tre valori

Un lock a tre valori ha tre valori possibili: rlocked, wlocked e unlocked.

Una transazione che vuole solo leggere un item  $X$  esegue  $rlock(X)$ , impedendo alle altre transazioni di scrivere ma non di leggere. Una transazione che vuole sia leggere che scrivere esegue  $wlock(X)$  impedendo alle altre transazioni sia di leggere che di scrivere. Entrambi i lock vengono rilasciati con  $unlock(X)$ .

Questi lock agiscono secondo la tabella:

	X unlocked	X rlocked	X wlocked
T esegue $rlock(X)$	T ottiene un lock di lettura su X	T ottiene un lock di lettura su X	T aspetta
T esegue $wlock(X)$	T ottiene un lock di scrittura su X	T aspetta	T aspetta

### 6.6.1 Equivalenza di schedule con lock a tre valori

Per verificare l'equivalenza di due schedule con lock a tre valori associamo ad ogni operazione di  $unlock$  successiva ad un'operazione di  $wlock$  una funzione che ha come argomenti gli item letti e scritti dalla transazione prima dell'operazione di  $unlock$ .

Due schedule sono equivalenti se le formule che danno il valore sono uguali per **tutti gli item** su cui viene effettuato un  $wlock$  e vengono letti gli stessi valori ad ogni  $rlock$ .

### 6.6.2 Algoritmo per testare la serializzabilità

Per controllare se un schedule  $S$  è serializzabile:

1. Crea un grafo  $G$  in cui gli archi con l'etichette di un item  $X$  tra  $T_i$  e  $T_j$  indicano due cose possibili:

- $T_i$  ha eseguito  $rlock(X)$  o  $wlock(X)$  e  $T_j$  è la prima transazione che fa  $wlock(X)$
  - $T_i$  ha eseguito  $wlock(X)$  e  $T_j$  esegue  $rlock(X)$  prima che un'altra transazione esegua  $wlock(X)$
2. Se  $G$  ha un ciclo allora  $S$  non è serializzabile, sennò basta usare l'ordinamento tipologico per ottenere uno schedule seriale  $S'$  equivalente a  $S$ . L'ordinamento topologico si ottiene eliminando ricorsivamente i nodi che non hanno archi entranti insieme ai suoi archi uscenti.

## 6.7 Lock write-only e read-only

Un lock write-only e read-only ha tre valori possibili: *rlocked*, *wlocked* e *unlocked*.

Una transazione che vuole solo leggere un item  $X$  esegue  $rlock(X)$ , impedendo alle altre transazioni di scrivere ma non di leggere. Una transazione solo scrivere senza leggere esegue  $wlock(X)$  impedendo alle altre transazioni sia di leggere che di scrivere. Entrambi i lock vengono rilasciati con  $unlock(X)$ .

### 6.7.1 Equivalenza di schedule con lock write-only e read-only

Per verificare l'equivalenza di due schedule con lock write-only e read-only associamo ad ogni operazione di *unlock* successiva ad un'operazione di *wlock* una funzione che ha come argomenti solamente gli item letti dalla transazione prima dell'operazione di *unlock*.

Uno schedule  $S'$  è equivalente ad un altro schedule  $S$  se:

1. se in  $S$  ho  $T_1$  che esegue  $wlock(X)$  e dopo  $T_2$  che esegue  $rlock(X)$  allora in  $S'$  devo mettere prima  $T_1$  e dopo  $T_2$ .
2. se  $T_3$  esegue  $wlock(X)$  allora in  $S'$  posso mettere  $T_3$  prima di  $T_1$  oppure dopo  $T_2$ , ma non in mezzo.
3. se  $T_i$  esegue  $rlock(X)$  e legge il valore iniziale di  $X$  allora in  $S'$  deve precedere qualunque  $T_j$  che esegua  $wlock(X)$
4. su  $T_i$  esegue  $wlock(X)$  e scrive l'ultimo valore di  $X$  allora in  $S'$  deve seguire qualunque  $T_j$  che esegua  $rlock(X)$

Gli ultimi due vincoli vengono assorbiti dall'1 e dal 2 se aggiungiamo una transazione  $T_0$  che esegue  $wlock()$  su tutti gli item e una transazione  $T_f$  che esegue  $rlock()$  su tutti gli item.

### 6.7.2 Algoritmo per testare la serializzabilità

Per controllare se un schedule  $S$  è serializzabile:

1. Crea un poligrafo  $P$  in cui gli archi con l'etichette di un item  $X$  tra  $T_i$  e  $T_j$  indicano due cose possibili:
  - $T_i$  ha eseguito  $wlock(X)$  e  $T_j$  fa  $rlock(X)$
  - se  $T_k$  esegue  $wlock(X)$  su un item  $X$  che impone l'esistenza di un arco  $T_i \rightarrow T_j$ , in questo caso si possono fare 3 cose:
    - creo l'arco  $T_j \rightarrow T_k$  se  $T_i = T_0 \wedge T_j \neq T_f$

- creo l'arco  $T_k \rightarrow T_i$  se  $T_j = T_f \wedge T_i \neq T_0$
  - creo due archi alternativi  $T_k \rightarrow T_i \wedge T_j \rightarrow T_k$  se  $T_i \neq T_0 \wedge T_j \neq T_f$
2. Se esiste almeno un grafo  $P$  aciclico ottenuto prendendo uno solo tra le coppie di archi alternativi allora  $S$  è serializzabile. Per ottenere  $S'$  schedule seriale equivalente a  $S$  bisogna usare il sorting topologico.

## 6.8 Deadlock

### Definizione di deadlock

Un **deadlock** è uno stallo che si verifica quando ogni transazione in un insieme  $T$  sta aspettando di ottenere un lock su un item su cui un'altra transazione ha già un lock e quindi rimangono bloccate e potrebbero bloccare anche transazioni non in  $T$ .

### 6.8.1 Verificare e risolvere un deadlock

Per verificare se sta accadendo una situazione di deadlock si crea un grafo di attesa  $G$ :

1.  $G$  ha un arco  $T_1 \rightarrow T_2$  se  $T_1$  sta aspettando il lock su un item attualmente in lock da  $T_2$
2. Se  $G$  ha un ciclo allora c'è una situazione di stallo che coinvolge le transazioni nel ciclo

Per risolvere un deadlock si utilizza il **roll-back**, cioè presa una transazione nel ciclo:

1. La transazione viene abortita
2. I suoi effetti sul DB vengono annullati
3. Tutti i lock della transazione vengono rilasciati

### Evitare un deadlock

Per evitare che si verifichi un deadlock si possono utilizzare dei protocolli, come ordinare gli item e imporre alle transazioni di richiedere i lock sugli item in ordine. In questo modo non ci può essere un ciclo nel grafo di attesa. **Dimostrazione per assurdo:**

Supponiamo che le transazioni richiedano gli item in ordine ma nel grafo ci sia un ciclo. Per far apparire il ciclo dovrei avere che  $X_k$  precede  $X_1$ , ma per l'ordinamento è  $X_1$  a precedere  $X_k$ , quindi è una contraddizione.

## 6.9 Livelock

Un **livelock** è uno stallo che si verifica quando una transazione aspetta in modo indefinito che gli venga concesso un lock su un item.

Si può evitare che avvenga un livelock in diversi modi come:

- Usare una strategia first come-first served
- Eseguire le transazioni in base alla priorità e aumentare la priorità di una transazione in base al tempo in cui rimane in attesa

## 6.10 Definizioni varie

### Abort di una transazione

Una transazione viene abortita quando si verificano determinate situazioni:

1. La transazione esegue un'operazione non corretta (divisione per 0, accesso non consentito)
2. Lo scheduler rileva un deadlock
3. Lo scheduler fa abortire la transazione per garantire la serializzabilità
4. Si verifica un malfunzionamento hardware o software

Quando abortiamo una transazione dobbiamo annullare i cambiamenti al DB effettuati sia dalla transazione che a tutte quelle che abbiano letto dati sporchi.

### Punto di commit

Il **punto di commit** di una transazione è il momento in cui la transazione ha ottenuto tutti i lock e ha effettuato tutti i calcoli. Non può quindi essere abortita per i motivi 1 e 3.

### Dati sporchi

I dati sporchi sono dei dati scritti da una transazione sul DB prima che abbia raggiunto il punto di commit. Per evitare la lettura di dati sporchi si usa un protocollo più restrittivo rispetto a quello a due fasi, cioè il **lock a due fasi stretto**.

## 6.11 Tipi di protocolli

I protocolli si dividono in due tipi:

- Conservativi: cercano di evitare le situazioni di stallo
- Aggressivi: cercano di eseguire le transazioni il più velocemente possibile anche se potrebbe portare a situazioni di stallo

In base alla probabilità che due transazioni richiedano un lock su uno stesso item conviene usare tipi di protocolli diversi:

- Probabilità alta: conviene usare un protocollo conservativo per evitare il sovraccarico dovuto alla gestione dei deadlock
- Probabilità bassa: conviene usare un protocollo aggressivo per evitare il sovraccarico sulla gestione dei lock

### 6.11.1 Protocolli conservativi

Un protocollo conservativo vuole che una transazione  $T$  richieda tutti i lock che servono all'inizio e li ottiene solo se sono tutti disponibili, se anche uno solo non può essere ottenuto la transazione viene messa in attesa. Questo protocollo evita i deadlock ma non i livelock.

Se si volessero evitare anche i livelock, bisogna che la transazione  $T$  oltre che ottenere i lock solo se sono tutti disponibili, li ottenga solo se nessuna transazione precedente nella coda sia in attesa di un lock richiesto da  $T$ .

I vantaggi dei protocolli conservativi sono:

- Evitare il verificarsi di deadlock e livelock

Gli svantaggi sono:

- L'esecuzione di una transazione può essere ritardata
- La transazione deve chiedere un lock su qualsiasi item potrebbe servirgli anche se poi non lo usa

### 6.11.2 Protocolli aggressivi

Un protocollo aggressivo permette a una transazione di richiedere un lock prima di leggerlo o scriverlo.

## 6.12 Timestamp

### Definizione di timestamp

Un timestamp è un valore che viene assegnato ad una transazione dallo scheduler quando la transazione inizia e può essere il valore di un contatore o l'ora di inizio della transazione. Un timestamp minore implica che la transazione è iniziata prima di un'altra.

### 6.12.1 Serializzabilità

Uno schedule è serializzabile se è equivalente allo schedule seriale in cui le transazioni sono eseguite in ordine in base al timestamp. Questo vuol dire che per ciascun item acceduto da più transazioni, l'ordine con cui queste lo fanno è in base al timestamp.

**Esempio:**

$T_1$
read(X)
$X=X+10$
write(X)
$TS(T_1)=110$

$T_2$
read(X)
$X=X+5$
write(X)
$TS(T_2)=100$

Questo schedule sarà quindi serializzabile solo se equivalente allo schedule seriale  $T_2T_1$ .

### 6.12.2 Read timestamp e write timestamp