

## **Basi di dati**

Riassunto dei principali argomenti  
ed esempi di esercizi

Autore:

**Danzi Matteo**

Matricola VR388529

# Indice

|           |   |           |
|-----------|---|-----------|
| <b>1</b>  | <b>Introduzione</b>                               | <b>3</b>  |
| <b>2</b>  | <b>Progettazione Concettuale</b>                  | <b>3</b>  |
| 2.1       | Modello Entità-Relazione . . . . .                | 3         |
| 2.2       | Entità . . . . .                                  | 4         |
| 2.3       | Relazione (o associazione tra Entità) . . . . .   | 5         |
| 2.4       | Attributo . . . . .                               | 7         |
| 2.5       | Identificatore di entità . . . . .                | 8         |
| 2.6       | Vincoli di cardinalità (Relazioni) . . . . .      | 9         |
| 2.7       | Attributo opzionale e multivalore . . . . .       | 10        |
| 2.8       | Attributo composto . . . . .                      | 10        |
| 2.9       | Generalizzazione . . . . .                        | 11        |
| 2.10      | Relazioni Ternarie . . . . .                      | 12        |
| <b>3</b>  | <b>Modello Relazionale</b>                        | <b>14</b> |
| 3.1       | Domini di base . . . . .                          | 14        |
| 3.2       | Costrutto Relazione . . . . .                     | 14        |
| 3.3       | Progettazione dei dati . . . . .                  | 16        |
| 3.4       | Valori nulli . . . . .                            | 18        |
| 3.5       | Vincoli di integrità . . . . .                    | 19        |
| <b>4</b>  | <b>Algebra Relazionale</b>                        | <b>23</b> |
| 4.1       | Operatori insiemistici . . . . .                  | 23        |
| 4.2       | Operatori specifici . . . . .                     | 23        |
| 4.3       | Operatori di giunzione . . . . .                  | 26        |
| <b>5</b>  | <b>Ottimizzazione di espressioni DML</b>          | <b>30</b> |
| 5.1       | Ottimizzatore . . . . .                           | 30        |
| 5.2       | Equivalenza tra espressioni algebriche . . . . .  | 31        |
| 5.3       | Trasformazioni di equivalenza . . . . .           | 31        |
| 5.4       | Ulteriori trasformazioni di equivalenza . . . . . | 32        |
| <b>6</b>  | <b>Soluzione dell'esercitazione</b>               | <b>35</b> |
| 6.1       | Esercizio 1.a . . . . .                           | 35        |
| 6.2       | Esercizio 1.b . . . . .                           | 35        |
| 6.3       | Esercizio 1.c . . . . .                           | 36        |
| 6.4       | Esercizio 1.d . . . . .                           | 36        |
| 6.5       | Esercizio 1.e . . . . .                           | 37        |
| <b>7</b>  | <b>Concetto di vista</b>                          | <b>38</b> |
| <b>8</b>  | <b>SQL - Structured Query Language</b>            | <b>38</b> |
| 8.1       | Sintassi . . . . .                                | 38        |
| 8.2       | Lista dei principali comandi . . . . .            | 39        |
| <b>9</b>  | <b>Transazioni</b>                                | <b>40</b> |
| 9.1       | Sintassi . . . . .                                | 40        |
| 9.2       | Proprietà . . . . .                               | 40        |
| 9.3       | Atomicità . . . . .                               | 41        |
| 9.4       | Consistenza . . . . .                             | 41        |
| 9.5       | Isolamento . . . . .                              | 41        |
| 9.6       | Persistenza . . . . .                             | 41        |
| <b>10</b> | <b>Architettura di un DBMS</b>                    | <b>42</b> |

|   |           |
|---|-----------|
| <b>11 Strutture fisiche e di accesso ai dati</b>          | <b>43</b> |
| 11.1 Gestore del buffer . . . . .                         | 43        |
| 11.2 Gestione delle pagine . . . . .                      | 44        |
| 11.3 Gestore dell'affidabilità . . . . .                  | 45        |
| 11.4 Gestore dei metodi di accesso . . . . .              | 48        |
| 11.5 B <sup>+</sup> -tree . . . . .                       | 54        |
| 11.6 Strutture ad accesso calcolato . . . . .             | 58        |
| <b>12 Esecuzione concorrente di Transizioni</b>           | <b>60</b> |
| 12.1 Anomalie di esecuzione concorrente . . . . .         | 60        |
| 12.2 Schedule . . . . .                                   | 61        |
| 12.3 Schedule seriale . . . . .                           | 61        |
| 12.4 Schedule serializzabile . . . . .                    | 61        |
| 12.5 Conflitto . . . . .                                  | 62        |
| 12.6 Locking a due fasi . . . . .                         | 64        |
| 12.7 Blocco di una transazione (starvation) . . . . .     | 67        |
| 12.8 Gestione della concorrenza in SQL . . . . .          | 67        |
| <b>13 Ottimizzazione di interrogazioni</b>                | <b>68</b> |
| 13.1 Ottimizzazione algebrica . . . . .                   | 68        |
| 13.2 Metodi di accesso disponibili . . . . .              | 68        |
| 13.3 Scelta finale del piano di esecuzione . . . . .      | 72        |
| <b>14 Interazione tra basi di dati e applicazioni</b>     | <b>73</b> |
| 14.1 Tipi di interazione . . . . .                        | 73        |
| 14.2 Interazione con DB in Java attraverso JDBC . . . . . | 73        |
| 14.3 7 Passi per interagire con un DBMS . . . . .         | 73        |
| 14.4 Transazioni in JDBC . . . . .                        | 75        |
| 14.5 Object Relational Mapping . . . . .                  | 76        |
| <b>15 Tecnologie NoSQL</b>                                | <b>76</b> |
| 15.1 Sistemi document-store . . . . .                     | 77        |






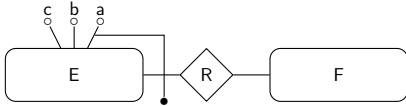


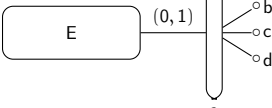
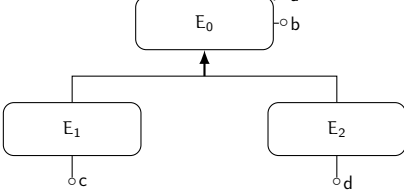
## 1 Introduzione

Questa dispensa contiene il riassunto dei principali argomenti svolti durante il corso di Basi di Dati (a.a. 2016/2017) e comprende un'esaurativa collezione di esempi ed esercizi presenti nelle slide del corso e svolti a lezione.

## 2 Progettazione Concettuale

### 2.1 Modello Entità-Relazione

Il modello Entità-Relazione (E-R) è un modello *concettuale* di dati, e come tale, fornisce una serie di strutture, dette *costrutti*, atte a descrivere la realtà di interesse in una maniera facile da comprendere e che prescinde dai criteri di organizzazione dei dati nei calcolatori. Questi costrutti vengono utilizzati per definire *schemi* che descrivono l'organizzazione e la struttura delle *occorrenze/istanze* dei dati, ovvero dei valori assunti dai dati al variare del tempo. Nella tabella vengono elencati tutti i costrutti che il modello E-R mette a disposizione, per ogni costrutto, c'è una relativa rappresentazione grafica.

| Costrutti                          | Rappresentazione Grafica   |
|------------------------------------|--|
| Entità                             |    |
| Relazione                          |   |
| Attributo                          |    |
| Identificatore di identità interno | <br> |
| Identificatore di identità esterno |    |
| Vincoli di cardinalità             |    |
| Attributo opzionale e multivalore  |    |
| Attributo composto                 |    |
| Generalizzazione                   |    |

## 2.2 Entità

### Semantica

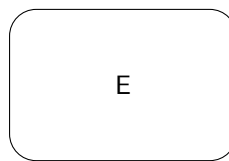
Rappresento una classe di oggetti con le seguenti caratteristiche:

- Hanno proprietà comuni
- Hanno esistenza autonoma
- Hanno identificazione univoca

Le proprietà nascono e muoiono insieme all'entità stessa.

### Sintassi

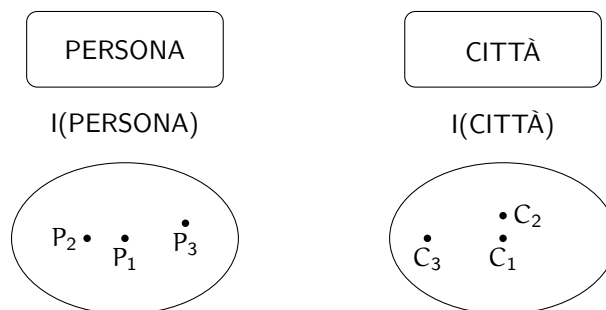
Una entità E dello schema si rappresenta graficamente nel seguente modo:



### Occorrenza/Istanza

L'occorrenza o istanza dell'entità è un oggetto appartenente alla classe rappresentata nello schema dall'entità E. L'insieme di tutte le occorrenze di E si indica con  $I(E)$ .

Esempi d'uso:



### Osservazioni:

- Inizialmente la base di dati è vuota:  $I(E) = \emptyset$  per ogni  $E_i \in \text{SCHEMA}$ .
- Le istanze delle entità dello schema hanno esistenza autonoma, nascono e muoiono in modo indipendente dal resto della base di dati.
- Ogni istanza di entità ha una vita indipendente dai valori assunti dalle sue proprietà.

## 2.3 Relazione (o associazione tra Entità)

### Semantica

Rappresento un legame logico tra due entità.

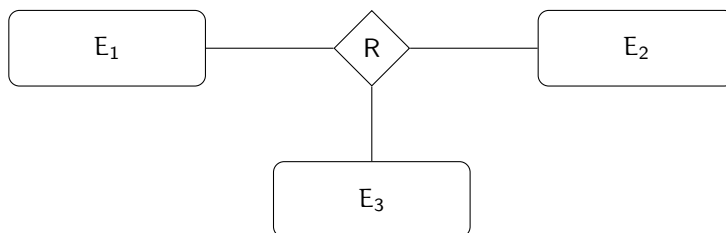
### Sintassi

Una rappresentazione grafica di una relazione si ottiene inserendo un rombo nello schema che va collegato attraverso spezzate a tutti i rettangoli che rappresentano entità coinvolte nella relazione. Casi possibili:

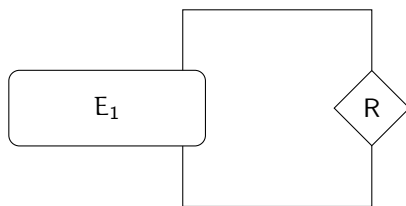
- Relazione Binaria



- Relazione Ternaria



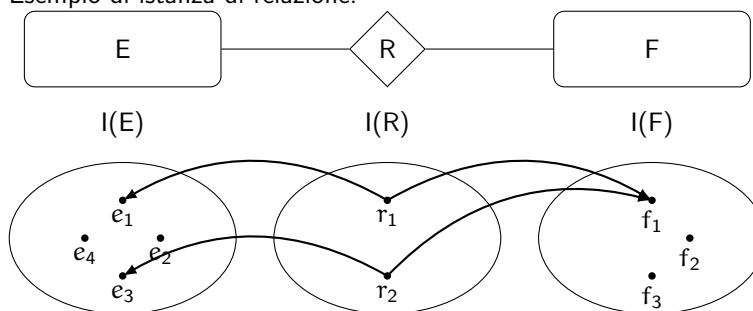
- Relazione Ricorsiva



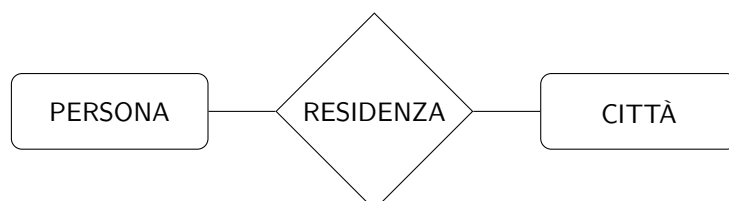
### Occorrenza/Istanza

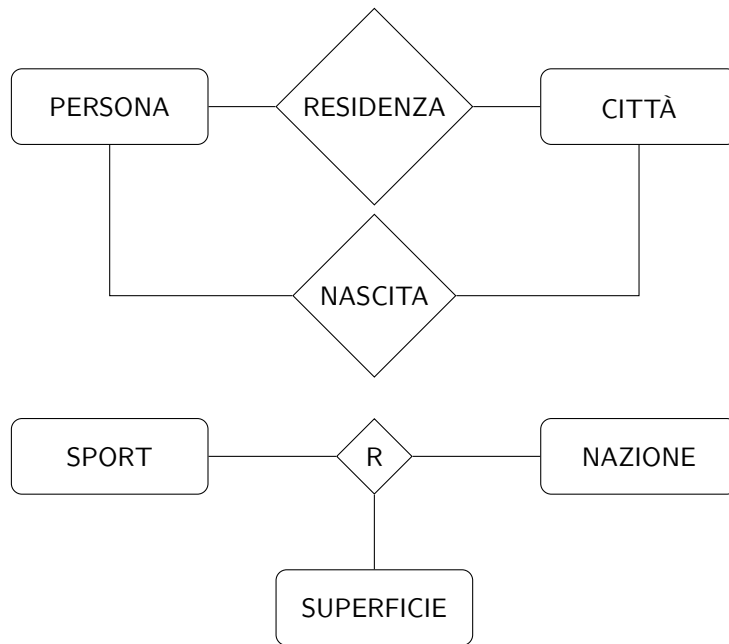
L'occorrenza di una relazione che coinvolge entità  $E_1, \dots, E_n$  è una ennupla di istanze di entità  $(e_1, \dots, e_n)$  dove  $e_i \in I(E_i) \quad \forall E_i \in E_1, \dots, E_n$ .

Esempio di istanza di relazione:



### Esempi d'uso





$I(R) = \{(\text{tennis, erba, Inghilterra}), (\text{tennis, terra, Italia})\}$

### Osservazioni

- Per far nascere una relazione è necessaria la presenza di almeno un'entità.
- Data una  $R$  e delle entità  $E_1, \dots, E_n$  l'insieme delle istanze di  $R$  è sempre un sottoinsieme del prodotto cartesiano  $I(E_1) \times \dots \times I(E_n)$  vale a dire:

$$I(R) \subset I(E_1) \times \dots \times I(E_n)$$

NON esistono in  $I(R)$  enuple duplicate.

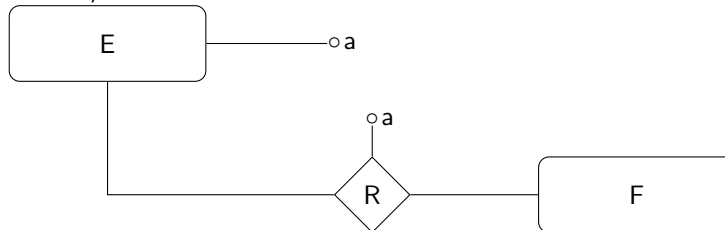
## 2.4 Attributo

### Semantica

Rappresenta una proprietà elementare di una entità (o relazione). Ogni attributo  $a$  di un'entità  $E$  (o relazione  $R$ ) può essere visto come una funzione che associa ad ogni istanza di entità (o relazione) uno e un solo valore appartenente ad un dominio (insieme di valori ammissibili).

### Sintassi

La rappresentazione grafica degli attributi è rappresentata da un pallino vuoto collegato con una spezzata all'entità  $E$ /relazione  $R$ .

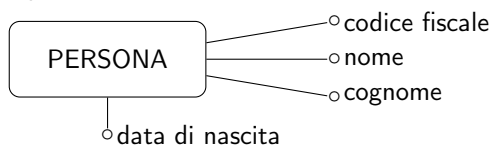


### Occorrenza/Istanza

L'istanza di un attributo  $a$  di un'entità  $E$  (o relazione  $R$ ) è costituita dal valore che la funzione rappresentata dall'attributo assume per ogni istanza di  $E$  (o  $R$ ).

$$v = f_0(e) \quad \text{dove} \quad e \in I(E) \quad \text{e} \quad f_0 : I(E) \rightarrow \text{dominio}$$

Esempio:



$$\begin{aligned}
 I(\text{PERSONA}) &= \{p_1, p_2, p_3\} \\
 f_{\text{nome}}(p_1) &= \text{'GIOVANNI'} \\
 f_{\text{nome}}(p_2) &= \text{'FRANCESCA'} \\
 f_{\text{nome}}(p_3) &= \text{'LUCA'}
 \end{aligned}$$



## 2.5 Identificatore di entità

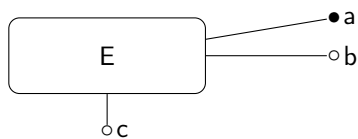
Viene utilizzato per indicare proprietà che devono essere soddisfatte dalla Base di dati. *Non esistono identificatori sulle relazioni.*

### Semantica

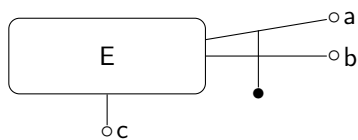
Rappresenta l'insieme di proprietà dell'entità (attributi o relazioni a cui l'entità partecipa) che consentono di identificare univocamente l'istanza dell'entità.

### Sintassi

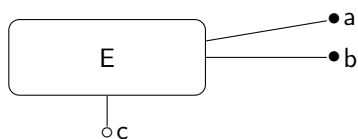
- Identificatore interno (solo attributi)



{a} è identificatore per E:

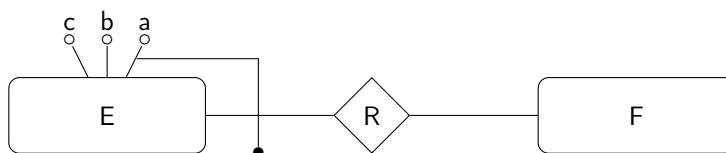


{a, b} è identificatore per E



{a} è identificatore per E {b} è identificatore per E

- Identificatore esterno (ci sono anche relazioni dell'identificatore):



{a, R} è un identificatore per E.

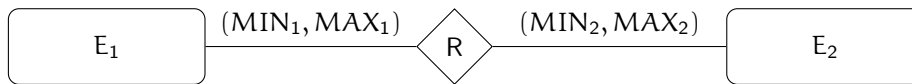
È necessario che R sia una funzione per far parte di un identificatore.

## 2.6 Vincoli di cardinalità (Relazioni)

### Semantica

Data una relazione  $R$  i vincoli di cardinalità si precisano per ogni entità  $E$  coinvolta nella relazione e indicano il numero minimo e il numero massimo di occorrenze della relazione  $R$  a cui ogni occorrenza  $E_i$  deve/può partecipare.

### Sintassi



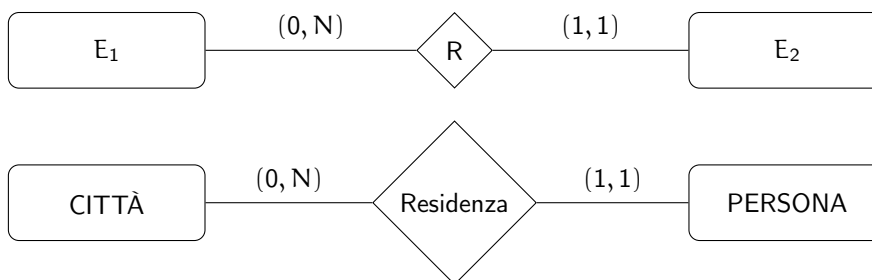
Valori tipici per  $MIN_1$ :

- 0 : indica che la partecipazione alla relazione  $R$  da parte dell'occorrenza di  $E_1$  è *opzionale*.
- 1 : indica che la partecipazione alla relazione  $R$  da parte dell'occorrenza di  $E_1$  è *obbligatoria*.
- $num > 1$  : indica che ogni occorrenza di  $E_1$  deve partecipare almeno a  $< num >$  occorrenze della relazione.

Valori tipici per  $MAX_1$ :

- 1 : indica che ogni occorrenza di  $E_1$  può partecipare al massimo a una occorrenza della relazione  $R$  (indica che  $R$  è una funzione dal punto di vista algebrico).
- "N" : indica che ogni occorrenza di  $E_1$  può partecipare ad un numero qualsiasi di occorrenze della relazione  $R$ .
- $num > 1$  : indica che ogni occorrenza di  $E_1$  può partecipare a  $< num >$  occorrenze della relazione  $R$ .

### Esempi



- $(0, N)$  posso avere al massimo  $N$  valori di città come residenza, quando la base di dati è vuota ho 0 città come residenza.
- $(1, 1)$  ogni persona ha esattamente un valore come residenza.

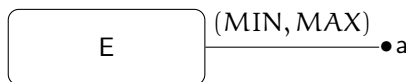
## 2.7 Attributo opzionale e multivalore

### Semantica

L'attributo opzionale/multivalore si ottiene precisando sull'attributo un vincolo di cardinalità che indica il numero minimo e massimo di valori che l'attributo deve/può assumere.

### Sintassi

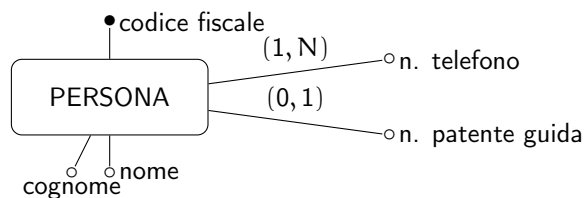
Il vincolo di default è ovviamente (1, 1). Negli altri casi:



Valori tipici per MIN e MAX:

- (0, 1) Attributo *opzionale*.
- (1, N) Attributo *multivalore*.
- (0, N) Attributo *opzionale e multivalore*.

### Esempi

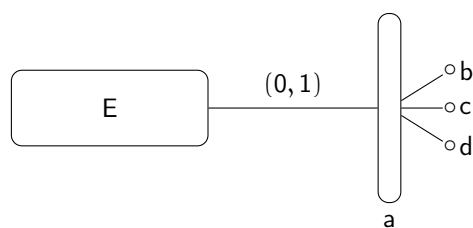


## 2.8 Attributo composto

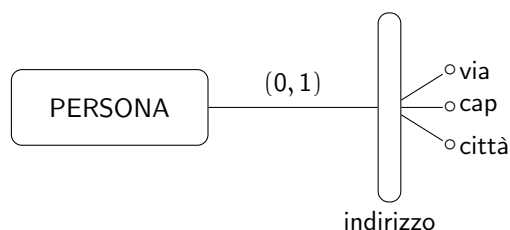
### Semantica

Consente di raggruppare attributi di entità o di una relazione quando presentano affinità di significato.

### Sintassi



### Esempi



Posso mettere anche (1, N)(multivalore), dipende dalla situazione.

## 2.9 Generalizzazione

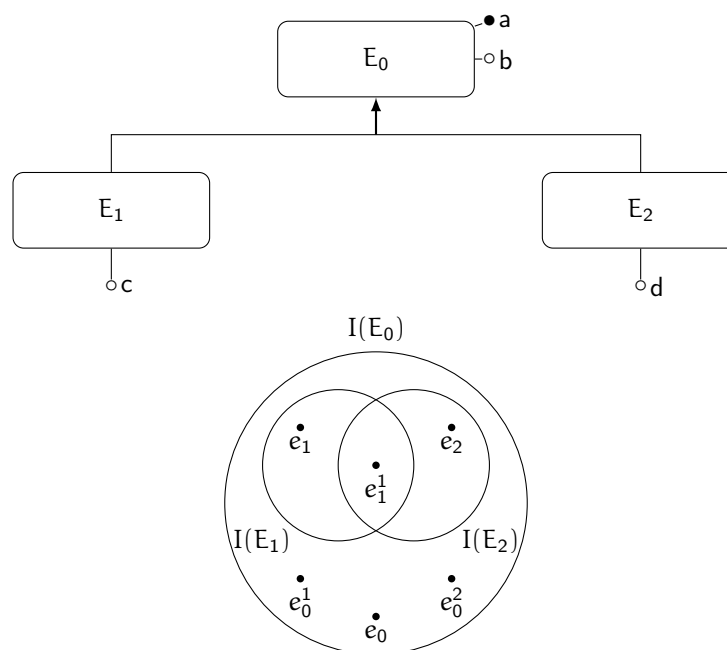
### Semantica

Rappresenta un legame logico simile ad una ereditarietà tra classi che coinvolge un'entità padre  $E_0$  e una o più entità figlie  $E_1, \dots, E_n$ , dove  $E_0$  rappresenta una classe di oggetti più generale rispetto alle classi di oggetti rappresentate dalle entità figlie  $E_1, \dots, E_n$ .

Proprietà delle occorrenze delle entità che partecipano ad una generalizzazione:

- **Ereditarietà:** ogni occorrenza delle entità figlie della generalizzazione eredita tutte le proprietà (attributi, relazioni, identificatori) specificate sull'entità padre.
- **Condivisione delle occorrenze:** ogni occorrenza delle entità figlie della generalizzazione è anche occorrenza dell'entità padre.

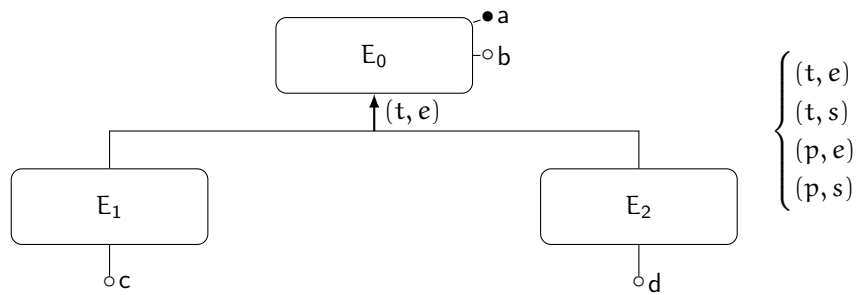
### Sintassi



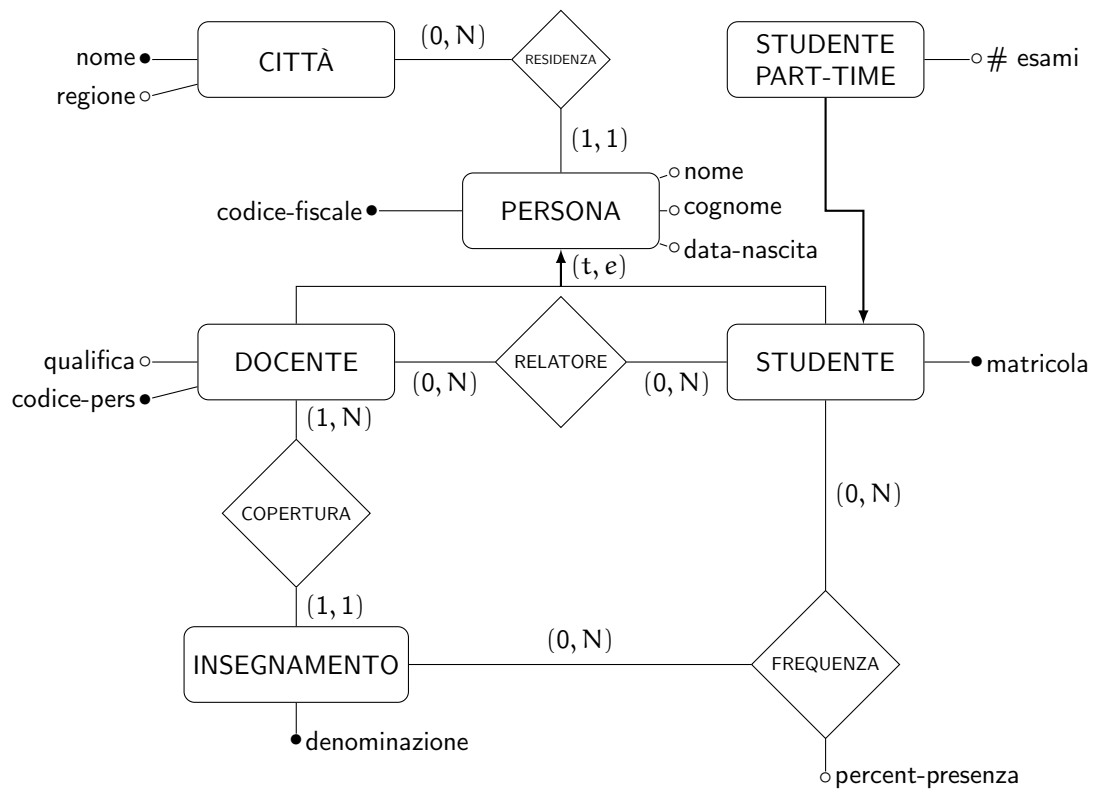
### Proprietà delle generalizzazioni

- Una generalizzazione si dice **totale** se ogni occorrenza dell'entità padre è anche occorrenza di almeno un'entità figlia.
- Se una generalizzazione non è totale, si dice **parziale**.
- Una generalizzazione si dice **esclusiva** se ogni occorrenza dell'entità padre è al più occorrenza di un'entità figlia.
- Se una generalizzazione non è esclusiva si dice **sovrapposta**.

Esempio:



### Esempio di generalizzazione

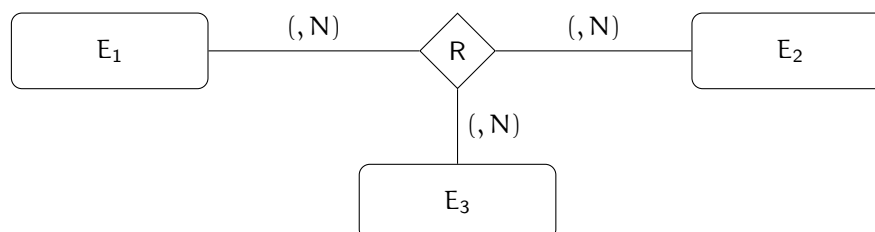


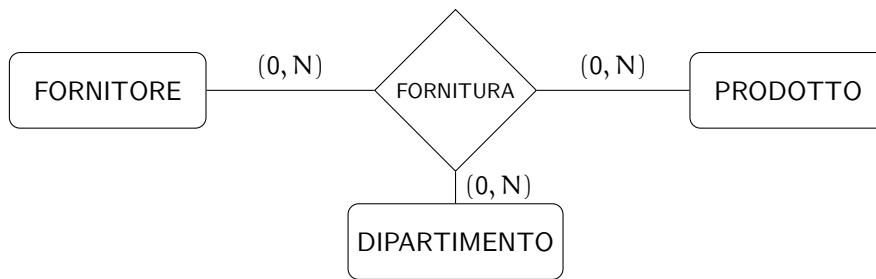
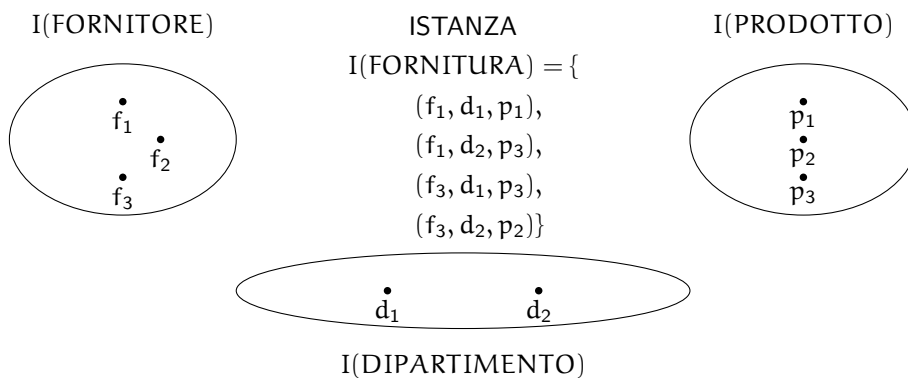
## 2.10 Relazioni Ternarie

### Semantica

Una relazione ternaria è una relazione che coinvolge tre entità.

### Sintassi



**Esempio****Istanza**

## 3 Modello Relazionale

Il *modello relazionale* si basa su due concetti, *relazione* e *tabella*, di natura diversa ma facilmente riconducibili l'uno all'altro. La nozione di *relazione* viene dalla matematica, in particolare dalla teoria degli insiemi, mentre il concetto di *tabella* è semplice e intuitivo.

Il modello relazionale risponde al requisito di indipendenza dei dati, che prevede una distinzione, nella descrizione dei dati, tra livello *fisico* e livello *logico*; gli utenti che accedono ai dati e i programmatori che sviluppano le applicazioni fanno riferimento solo al livello logico; i dati descritti a livello logico sono poi realizzati per mezzo di opportune strutture fisiche, ma per accedere ai dati non è necessario conoscere le strutture fisiche stesse.

### 3.1 Domini di base

Sono i domini da cui si scelgono i valori delle proprietà delle istanze di informazione da rappresentare. I domini tipici sono:

- Caratteri
- Stringhe di caratteri
- Numeri interi
- Numeri decimali a virgola fissa
- Numeri decimali a virgola mobile
- Domini del tempo: per rappresentare istanti e intervalli di tempo
- ecc. . .

### 3.2 Costrutto Relazione

Presentazione intuitiva: una relazione può essere vista come una tabella, ad esempio:

|         |       |         |
|---------|-------|---------|
| MILANO  | 20100 | 1300000 |
| VERONA  | 37100 | 350000  |
| BRESCIA | 25100 | 250000  |

Una tabella è un contenitore di dati a cui struttura è caratterizzata da una lista di colonne:

- I dati sono scritti nelle righe dove ogni riga descrive le caratteristiche di una istanza dell'informazione da rappresentare.
- I valori contenuti nelle colonne descrivono sempre la stessa proprietà delle istanze di informazione da rappresentare.

### Definizione di relazione come insieme di ennuple (lists)

**Definizione.** Dati  $n$  insiemi di valori (domini)  $D_1, \dots, D_n$  con  $n > 0$  e indicato con  $D_1 \times \dots \times D_n$  il loro prodotto cartesiano:

$$D_1 \times \dots \times D_n = \{(v_1, \dots, v_n) \mid v_1 \in D_1 \wedge \dots \wedge v_n \in D_n\}$$

Una relazione  $\rho$  di grado  $n$  è un qualsiasi sottoinsieme di  $D_1 \times \dots \times D_n$  :

$$\rho \subseteq D_1 \times \dots \times D_n$$

dove:

- $(v_1, \dots, v_n)$  è una ennupla della relazione
- $|\rho|$  è la cardinalità della relazione (numero di ennuple)

Si noti che:

- I domini  $D_1, \dots, D_n$  possono essere a cardinalità infinita, mentre le relazioni sono **SEMPRE a cardinalità finita**
- Dalla definizione si deduce che:
  - Non è definito alcun ordinamento sulle ennuple di una relazione
  - Non sono ammessi **DUPLICATI** di una ennupla
  - Nella definizione di relazione come insieme di ennuple, i valori nelle ennuple sono ordinati.

### Esempio

Relazione delle città:

$$\rho = \{(MILANO, 20100, 1.300.000), (VERONA, 37100, 350.000), (BRESCIA, 25100, 250.000)\}$$

$$\rho \subseteq D_1 \times D_2 \times D_3$$

- $D_1$  = Stringhe di caratteri
- $D_2$  = Numeri interi
- $D_3$  = Numeri interi

### Accesso ai valori di una ennupla

- Se  $t$  è una ennupla  $(v_1, \dots, v_n)$  il valore posto in  $i$ -esima posizione si indica con la notazione:  $t[i]$ .
- Questa modalità di accesso ai valori non è efficace per l'uso pratico delle relazioni si preferisce quindi assegnare un nome alle colonne; ciò conduce all'introduzione della definizione di relazione come insieme di tuple.

### Definizione di relazione come insieme di tuple (mappings)

**Definizione.** Sia  $X$  un insieme di nomi e sia  $\Delta$  l'insieme di tutti i domini di base ammessi dal modello. Si definisce la funzione:

$$\text{DOM} : X \rightarrow \Delta$$

Che associa ad ogni nome  $A$  di  $X$  un dominio  $\text{DOM}(A)$  di  $\Delta$ . I nomi di  $X$  si dicono *attributi*. Una *tupla* su  $X$  è una funzione:

$$t : X \rightarrow \bigcup_{A \in X} \text{DOM}(A)$$

dove:

$$t[A] = v \in \text{DOM}(A)$$

Una relazione su  $X$  è un insieme di tuple su  $X$ , dove  $X$  è l'insieme di attributi della relazione.

**Esempio** Relazione delle città:

- $X$  = Nome, CAP, Abitanti
- $\text{DOM}(\text{Nome})$  = Stringhe di caratteri
- $\text{DOM}(\text{CAP})$  = Numeri interi
- $\text{DOM}(\text{Abitanti})$  = Numeri interi



$$\rho_X = \{t_1, t_2, t_3\}$$

$t_1$  [Nome] = MILANO       $t_2$  [Nome] = VERONA       $t_3$  [...] = ...  
 $t_1$  [CAP] = 20100       $t_2$  [CAP] = 37100  
 $t_1$  [Abitanti] = 1.300.000       $t_2$  [Abitanti] = 350.000

Si noti che:

- Una relazione è un insieme di tuple e quindi **non può contenere tuple duplicate**.
- I domini per gli *attributi* possono essere solo **domini di base**, non sono ammessi altri domini, né il prodotto cartesiano di domini.
- In generale una base di dati relazionale è costituita da *più relazioni*.

### 3.3 Progettazione dei dati

Il progetto dei dati nel modello relazionale potrebbe seguire il seguente approccio:

- Identificare gli attributi elementari da rappresentare.
- Creare inizialmente un'unica relazione (tabella) che li contiene tutti.
- Analizzare il risultato.
- Rivedere e decomporre la tabella in caso di problemi.

Quali problemi genera la rappresentazione di tutti gli attributi in un'unica relazione (tabella)?

Caso pratico:

- Informazioni sui proprietari di appartamenti:  
codice fiscale, cognome, nome, data di nascita, codice catastale, via, numero civico, subalterno, tipo
- Tabella unica:  
PROPRIETA'(CodiceFiscale, Cognome, Nome, DataNas, CodiceCatasto, Via, NumeroCivico, Subalterno, Tipo)

Se in una relazione (tabella) si uniscono concetti disomogenei e con esistenza autonoma si presenta la seguente situazione:

- Ogni tupla rappresenta solitamente un'istanza della associazione che lega i concetti autonomi
- Per le istanze di informazione che sono coinvolte in più associazioni si produce una ripetizione inutile di valori in diverse tuple (**RIDONDANZA**)

La presenza di ridondanza produce le seguenti anomalie:

- **ANOMALIA di aggiornamento:** per aggiornare il valore di un attributo si è obbligati a modificare tale valore su più tuple della base di dati.
- **ANOMALIA di inserimento:** per inserire una nuova istanza di un concetto è necessario inserire valori al momento sconosciuti (sostituibili con valori NULLI) per gli attributi non disponibili.
- **ANOMALIA di cancellazione:** per cancellare un'istanza di un concetto è necessario cancellare valori ancora validi oppure inserire valori NULLI per gli attributi da cancellare.

Un buon progetto dei dati a livello logico non contiene ridondanza inutile

Un metodo per eliminare la ridondanza è quello di decomporre la relazione (tabella) unica in più tabelle

Tuttavia anche nella decomposizione è necessario seguire delle regole per non perdere informazione.

Ad esempio, nel caso di tabella unica mostrata in precedenza quale decomposizione possiamo applicare per eliminare la ridondanza e non perdere informazione?

**PROPRIETÀ**

|               |         |      |         |               |     |              |            |      |
|---------------|---------|------|---------|---------------|-----|--------------|------------|------|
| CodiceFiscale | Cognome | Nome | DataNas | CodiceCatasto | Via | NumeroCivico | Subalterno | Tipo |
|---------------|---------|------|---------|---------------|-----|--------------|------------|------|

si può decomporre in:

**PROPRIETARIO**

|               |         |      |         |
|---------------|---------|------|---------|
| CodiceFiscale | Cognome | Nome | DataNas |
|---------------|---------|------|---------|

**UNITÀ\_IMMOBILIARE**

|               |     |              |            |      |
|---------------|-----|--------------|------------|------|
| CodiceCatasto | Via | NumeroCivico | Subalterno | Tipo |
|---------------|-----|--------------|------------|------|

Si noti che questa decomposizione:

- elimina qualsiasi ridondanza, ma
- non conserva l'informazione che descrive l'associazione tra proprietario e unità immobiliare.

Come è possibile conservare tale informazione senza ritornare nella condizione di ridondanza che si produce nella tabella unica?

- Per conservare il legame logico è necessario replicare parte degli attributi, scegliendo tra questi quelli che hanno la proprietà di identificare il concetto verso il quale si vuole generare il legame.
- Nell'esempio precedente possiamo scegliere il codice fiscale come identificatore del proprietario e il codice catastale come identificatore dell'unità immobiliare.
- Quindi applichiamo una decomposizione dove replichiamo solo questi due attributi per rappresentare le istanze di associazione tra proprietario e unità immobiliare.

Quindi otteniamo la seguente decomposizione della tabella unica:

**PROPRIETÀ**

|               |         |      |         |               |     |              |            |      |
|---------------|---------|------|---------|---------------|-----|--------------|------------|------|
| CodiceFiscale | Cognome | Nome | DataNas | CodiceCatasto | Via | NumeroCivico | Subalterno | Tipo |
|---------------|---------|------|---------|---------------|-----|--------------|------------|------|

Si decompone così:

**PROPRIETARIO**

|               |         |      |         |
|---------------|---------|------|---------|
| CodiceFiscale | Cognome | Nome | DataNas |
|---------------|---------|------|---------|

**UNITÀ\_IMMOBILIARE**

|               |     |              |            |      |
|---------------|-----|--------------|------------|------|
| CodiceCatasto | Via | NumeroCivico | Subalterno | Tipo |
|---------------|-----|--------------|------------|------|

**PROPRIETÀ**

|               |               |
|---------------|---------------|
| CodiceFiscale | CodiceCatasto |
|---------------|---------------|

Questa decomposizione elimina tutta la ridondanza inutile e conserva tutta l'informazione rappresentata nella tabella unica.

Si noti che il modello relazionale è **VALUE-BASED**, e ciò significa che:

- È totalmente indipendente dalla rappresentazione fisica (tutta l'informazione è nei valori) e non ci sono meccanismi per gestire riferimenti o puntatori tra istanze di informazione.
- I legami logici tra tuple diverse si realizzano attraverso la replicazione di alcuni attributi (che hanno la proprietà di identificare il concetto e quindi di rappresentarlo): il legame tra due tuple si intende stabilito quando esse presentano gli stessi valori negli attributi replicati.
- È facile trasferire i dati da una base di dati all'altra.
- È rappresentato solo ciò che è rilevante per l'applicazione.

**Schema di una relazione:** è costituito dal nome della relazione e da un insieme di nomi per i suoi attributi:

$$R(A_1, \dots, A_n) \quad \text{oppure} \quad R(A_1 : D_1, \dots, A_n : D_n)$$

dove:  $\text{DOM}(A_i) = D_i$

**Schema di una base di dati:** è un insieme di schemi di relazione:

$$S = \{R_1(A_{1,1}, \dots, A_{1,n_1}), \dots, R_m(A_{m,1}, \dots, A_{m,n_m})\}$$

dove:  $R_1 \neq \dots \neq R_m$

**Istanza di una relazione** di schema  $R(A_1, \dots, A_n)$  con  $X = \{A_1, \dots, A_n\}$ :

è un insieme  $r$  di tuple su  $X$ .

**Istanza di una base di dati** di schema  $S = \{R_1(A_{1,1}, \dots, A_{1,n_1}), \dots, R_m(A_{m,1}, \dots, A_{m,n_m})\}$ :

è un insieme di istanze di relazioni  $db = \{r_1, \dots, r_m\}$  dove ogni  $r_i$  è un'istanza della relazione di schema  $R_i(A_{i,1}, \dots, A_{i,n_i})$ .

### 3.4 Valori nulli

Dalla definizione di relazione si ha che ogni tupla deve sempre contenere nei propri attributi valori significativi appartenenti ai domini di base del modello.

Non sempre in una base di dati reale esistono i valori per tutti gli attributi di una tupla.

Per quali motivi il valore di un attributo in certi casi manca?

Si possono presentare i seguenti casi:

- Il valore di un attributo  $A$  è **inesistente** (attributo opzionale a livello concettuale): non esiste per questa tupla un valore per l'attributo  $A$
- Il valore di un attributo  $A$  è **sconosciuto**: esiste un valore per l'attributo  $A$  di questa tupla ma non è noto alla base di dati.
- Il valore di un attributo  $A$  è sconosciuto o inesistente.

Per poter gestire tali situazioni viene introdotto nel modello relazionale un valore speciale detto **VALORE NULLO**.

Gli attributi di una tupla possono assumere un valore del dominio oppure il valore nullo

**Definizione. Tupla su  $X$  con valori nulli**

Una tupla su  $X$  è una funzione:

$$t : X \rightarrow \{\text{NULL}\} \cup \left( \bigcup_{A \in X} \text{DOM}(A) \right)$$

dove:

$$t[A] = v \in \text{DOM}(A) \vee t[A] = \text{NULL}$$

Si noti che:

- La presenza di valori nulli è accettabile solo in alcuni attributi (non è possibile avere tuple di soli valori nulli).
- In particolare, negli attributi replicati per rappresentare legami tra tuple la presenza di valori nulli può rendere inutilizzabile l'informazione.

### 3.5 Vincoli di integrità

Spesso è necessario introdurre vincoli (restrizioni) sulla popolazione (istanza) di una base di dati in quanto non tutte le possibili istanze sono corrette rispetto al sistema informativo considerato.

Allo scopo di indicare quali sono le istanze corrette si introduce il concetto di:

**Definizione. Vincolo di integrità** è una condizione (espressa da un **predicato**) che deve essere **sempre** soddisfatta da **ogni istanza** della base di dati.

#### Esempi di vincolo

##### TRENO

| Numero | OraPart | MinutoPart | Categoria | Destinazione | OraArr | MinutoArr |
|--------|---------|------------|-----------|--------------|--------|-----------|
|--------|---------|------------|-----------|--------------|--------|-----------|

##### FERMATA

| NumTreno | Stazione | OraFer | MinutoFer |
|----------|----------|--------|-----------|
|----------|----------|--------|-----------|

Predicati che esprimono possibili vincoli:

- $\forall t \in \text{TRENO} : t[\text{OraPart}] \in \{0, 1, \dots, 23\}$
- $\forall t \in \text{TRENO} : t[\text{MinutoPart}] \in \{0, 1, \dots, 59\}$
- $\forall t \in \text{TRENO} : t[\text{Numero}] > 0$
- $\forall t \in \text{TRENO} : t[\text{Numero}] > 5000 \Rightarrow t[\text{Categoria}] = \text{'regionale'}$
- $\forall t, t' \in \text{TRENO} : t \neq t' \Rightarrow t[\text{Numero}] \neq t'[\text{Numero}]$
- $\forall f \in \text{FERMATA} : \exists t \in \text{TRENO} : t[\text{Numero}] = f[\text{NumTreno}]$
- $\forall t \in \text{TRENO} : t[\text{Categoria}] = \text{'regionale'} \Rightarrow \exists f \in \text{FERMATA} : f[\text{NumTreno}] = t[\text{Numero}]$

Si introduce la seguente classificazione dei vincoli di integrità.

Si vogliono inoltre distinguere alcuni vincoli particolari che hanno una maggiore importanza degli altri in quanto garantiscono proprietà generali valide per tutti gli schemi relazionali ( "vincoli strutturali"):

- VINCOLI DI DOMINIO
- VINCOLI DI TUPLA
- VINCOLI INTRARELAZIONALI
  - ✧ CHIAVI
- VINCOLI INTERRELAZIONALI
  - ✧ VINCOLI DI INTEGRITÀ' REFERENZIALE

#### Vincoli di Dominio

Impongono una restrizione sul dominio dell'attributo di una relazione, ad esempio: data la relazione

##### TRENO

| Numero | OraPart | MinutoPart | Categoria | Destinazione | OraArr | MinutoArr |
|--------|---------|------------|-----------|--------------|--------|-----------|
|--------|---------|------------|-----------|--------------|--------|-----------|

I seguenti vincoli riguardano il dominio di un attributo di TRENO:

- $\forall t \in \text{TRENO} : t[\text{OraPart}] \in \{0, 1, \dots, 23\}$
- $\forall t \in \text{TRENO} : t[\text{MinutoPart}] \in \{0, 1, \dots, 59\}$
- $\forall t \in \text{TRENO} : t[\text{Numero}] > 0$

## Vincoli di tupla

Impongono una restrizione alla combinazione di valori che una tupla della relazione può assumere indipendentemente dalle altre tuple, ad esempio: data la relazione:

### TRENO

| Numero | OraPart | MinutoPart | Categoria | Destinazione | OraArr | MinutoArr |
|--------|---------|------------|-----------|--------------|--------|-----------|
|--------|---------|------------|-----------|--------------|--------|-----------|

il seguente è un vincolo di tupla:

$$\forall t \in \text{TRENO} : t[\text{Numero}] > 5000 \Rightarrow t[\text{Categoria}] = \text{'regionale'}$$

## Vincoli Intrarelazionali

Impongono una restrizione al contenuto di una relazione e specificano una condizione che ogni tupla della relazione deve soddisfare rispetto alle altre tuple della medesima relazione.

Una sottocategoria importante di tali vincoli include i **vincoli di chiave**:

- Superchiave
- Chiave candidata
- Chiave primaria

**Definizione. Superchiave:** Data una relazione di schema  $R(X)$ , un insieme di attributi  $K$ , sottoinsieme di  $X$ , è SUPERCHIAVE per  $R(X)$  se, per ogni istanza  $r$  di  $R(X)$  vale la seguente condizione:

$$\forall t, t' \in r : t \neq t' \Rightarrow t[K] \neq t'[K]$$

dove:

$$t[K] \neq t'[K] \equiv \exists A_i \in K : t[A_i] \neq t'[A_i]$$

**Definizione. Chiave Candidata:** Data una relazione di schema  $R(X)$ , un insieme di attributi  $K$ , sottoinsieme di  $X$ , è CHIAVE CANDIDATA (o CHIAVE) per  $R(X)$ , se  $K$  è superchiave per  $R(X)$  e vale la seguente condizione:

$$\neg \exists K' \subset K : K' \text{ è superchiave per } R(X)$$

**Teorema.** Esiste sempre una chiave candidata  $K$  per una relazione  $R(X)$ .

**Dim.**

- $X$  è superchiave per  $R(X)$
- se  $X$  non ha sottoinsiemi propri che sono superchiave, allora  $X$  è chiave
- altrimenti si riapplica il medesimo ragionamento ricorsivamente al sottoinsieme di  $X$  che è superchiave
- poiché la cardinalità di  $X$  è finita e poiché l'insieme vuoto non è superchiave si troverà in un numero finito di passi una chiave per  $R(X)$

**Definizione. Chiave Primaria:** Data una relazione di schema  $R(X)$  la sua CHIAVE PRIMARIA è la chiave candidata scelta per **identificare le tuple della relazione**.

Una chiave primaria  $K$  ha le seguenti caratteristiche:

- Non contiene mai valori nulli
- Su  $K$  il sistema genera una struttura d'accesso ai dati (o indice) per supportare le interrogazioni.

## Esempi

Data la relazione:

### TRENO

| Numero | OraPart | MinutoPart | Categoria | Destinazione | OraArr | MinutoArr |
|--------|---------|------------|-----------|--------------|--------|-----------|
|--------|---------|------------|-----------|--------------|--------|-----------|

Considerando i seguenti sottoinsiemi di X, dire se, rispetto al contesto applicativo, il sottoinsieme è superchiave o no e, se è superchiave, dire se è anche chiave:

- $K_1 = \{\text{Numero}\}$
- $K_2 = \{\text{Numero}, \text{Categoria}\}$
- $K_3 = \{\text{OraPart}, \text{MinutoPart}, \text{Destinazione}, \text{Categoria}\}$

Data la relazione:

### FERMATA

| NumTreno | Stazione | OraFer | MinutoFer |
|----------|----------|--------|-----------|
|----------|----------|--------|-----------|

Considerando i seguenti sottoinsiemi di X, dire se, rispetto al contesto applicativo, il sottoinsieme è superchiave o no e, se è superchiave, dire se è anche chiave:

- $K_1 = \{\text{NumTreno}\}$
- $K_2 = \{\text{NumTreno}, \text{Stazione}\}$
- $K_3 = \{\text{OraFer}, \text{MinutoFer}, \text{Stazione}\}$

## Vincoli Interrelazionali

Impongono una restrizione al contenuto di una relazione e specificano una condizione che ogni tupla della relazione deve soddisfare rispetto alle tuple di altre relazioni della base di dati.

Una sottocategoria importante di tali vincoli include i **vincoli di integrità referenziale (o vincoli sulle chiavi esportate)**

**Definizione. Vincolo di integrità referenziale:** Un vincolo di integrità referenziale tra un insieme di attributi  $Y = \{A_1, \dots, A_p\}$  di  $R_1$  e un insieme di attributi  $K = \{K_1, \dots, K_p\}$ , chiave primaria di un'altra relazione  $R_2$ , è soddisfatto se, per ogni istanza  $r_1$  di  $R_1$  e per ogni istanza  $r_2$  di  $R_2$  vale la seguente condizione:

$$\forall t \in r_1 : \exists s \in r_2 : \forall i \in \{1, \dots, p\} : t[A_i] = s[K_i]$$

**Definizione. Vincolo di integrità referenziale in presenza di valori nulli (legame opzionale)** Un vincolo di integrità referenziale tra un insieme di attributi  $Y = \{A_1, \dots, A_p\}$  di  $R_1$  e un insieme di attributi  $K = \{K_1, \dots, K_p\}$ , chiave primaria di un'altra relazione  $R_2$ , è soddisfatto se, per ogni istanza  $r_1$  di  $R_1$  e per ogni istanza  $r_2$  di  $R_2$  vale la seguente condizione:

$$\forall t \in r_1 : \exists s \in r_2 : (\forall i \in \{1, \dots, p\} : t[A_i] = s[K_i]) \vee (\exists i \in \{1, \dots, p\} : t[A_i] = \text{NULL})$$

Chiave primaria di una relazione  $R(X)$ : viene indicata sottolineando gli attributi della chiave:

Supposto che venga scelta come chiave primaria di TRENO l'insieme {Numero} la notazione per indicare tale chiave è la seguente:

### TRENO

| <u>Numero</u> | OraPart | MinutoPart | Categoria | Destinazione | OraArr | MinutoArr |
|---------------|---------|------------|-----------|--------------|--------|-----------|
|---------------|---------|------------|-----------|--------------|--------|-----------|

La chiave primaria di FERMATA sarà indicata invece come segue:

### FERMATA

| <u>NumTreno</u> | <u>Stazione</u> | OraFer | MinutoFer |
|-----------------|-----------------|--------|-----------|
|-----------------|-----------------|--------|-----------|

Vincolo di integrità referenziale: viene indicato riquadrando gli attributi soggetti al vincolo e collegando con una freccia il riquadro alla relazione (tabella) da cui la chiave proviene.

Supposto che esista un vincolo di integrità referenziale sull'attributo NumTreno della tabella FERMATA rispetto alla tabella TRENO, la notazione per indicare tale vincolo è la seguente:

**TRENO** ←

|               |         |            |           |              |        |           |
|---------------|---------|------------|-----------|--------------|--------|-----------|
| <u>Numero</u> | OraPart | MinutoPart | Categoria | Destinazione | OraArr | MinutoArr |
|---------------|---------|------------|-----------|--------------|--------|-----------|

**FERMATA**

|                 |                 |        |           |
|-----------------|-----------------|--------|-----------|
| <u>NumTreno</u> | <u>Stazione</u> | OraFer | MinutoFer |
|-----------------|-----------------|--------|-----------|

## 4 Algebra Relazionale

L'algebra relazionale è formata da un insieme di operatori (operazioni):

- Operatori **unari**:  $op_p(r_1) \rightarrow r$
- Operatori **binari**:  $r_1 op r_2 \rightarrow r$

Classificazione degli operatori dell'algebra:

- Operatori **insiemistici**
- Operatori **specifici**
- Operatori **di giunzione (o join)**

Per ognuno di questi tre gruppi ci sono operatori **di base** e operatori **derivati**.

### 4.1 Operatori insiemistici

**Osservazione.** le relazioni sono insiemi di *tuple omogenee*, quindi posso applicare alle relazioni le operazioni dell'algebra degli insiemi.

**Definizione.** Date due relazioni  $r_1$  e  $r_2$  di schema  $R_1(X)$  e  $R_2(X)$  si definiscono i seguenti operatori:

di Base

$$\textbf{Unione: } r_1 \cup r_2 = r \quad \begin{cases} \text{schema} & X \\ \text{istanza} & \{t \mid t \in r_1 \vee t \in r_2\} \end{cases}$$

$$\textbf{Differenza: } r_1 - r_2 = r \quad \begin{cases} \text{schema} & X \\ \text{istanza} & \{t \mid t \in r_1 \wedge t \notin r_2\} \end{cases}$$

Derivati

$$\textbf{Intersezione: } r_1 \cap r_2 = r_1 - (r_1 - r_2)$$

### 4.2 Operatori specifici

Caratterizzano l'algebra relazionale e sono:

- Ridenominazione  $\rightarrow \rho$
- Selezione  $\rightarrow \sigma$
- Proiezione  $\rightarrow \pi$

**Definizione.** Data una relazione  $r$  di schema  $R(X)$  con  $X = \{A_1 \dots A_n\}$  si definiscono i seguenti operatori:

#### Ridenominazione

È utile per **modificare lo schema di una relazione**.

Dato un insieme di attributi  $Y = \{B_1 \dots B_n\}$  con  $|Y| = |X|$  si definisce il seguente operatore:

$$\rho_{A_1 \dots A_n \rightarrow B_1 \dots B_n}(r) = \begin{cases} \text{schema} & X \\ \text{istanza} & \{t \mid \exists t' \in r : \forall i \in [1 \dots n] \forall B_i \in Y t[B_i] = t'[A_i]\} \end{cases}$$



### Esempio di Ridenominazione

Popolazione:

| Nome    | Cap   | Abitanti |
|---------|-------|----------|
| Verona  | 37100 | 35000    |
| Vicenza | 50100 | 15000    |

Città:

| Comune | Cap   | Popolazione |
|--------|-------|-------------|
| Milano | 20100 | 2500000     |

Non posso fare l'unione, devo prima ridenominare:

$$\rho_{\text{Nome, Cap, Abitanti} \rightarrow \text{Comune, Cap, Popolazione}}(\text{Popolazione}) \cup (\text{Città})$$

### Selezione

Consente di **estrarre da una relazione le tuple che soddisfano la condizione**  $F$ . (Riduzione in orizzontale della relazione)

$$\sigma_F(r) = \begin{cases} \text{schema} & X \\ \text{istanza} & \{t \mid \exists t' \in r : F(t) \text{ (la tupla } t \text{ rende vera } F)\} \end{cases}$$

$F$  è una formula proposizionale che si ottiene combinando attraverso i connettivi logici  $\wedge, \vee, \neg$  formule *atomiche* del tipo:

- $A \theta B$
- $A \theta c$

Dove:

- $\theta \in \{=, \neq, >, <, \geq, \leq\}$
- $A, B \in X$
- $c \in \text{DOM}(A)$  o è compatibile con  $\text{DOM}(A)$

### Esempi di $F$

TRENO(Numero, Cat, Dest, Arr, Port)

- $F: \text{Cat} = \text{'Regionale'}$
- $F: \text{Numero} = 12345$
- $F: \text{Cat} \neq \text{'Regionale'} \wedge \text{Dest} = \text{'Padova'}$

### Interpretazione di $F$

Una formula  $A \theta B$  è vera sulla tupla  $t$  se è vera la seguente condizione:

$$t[A] \theta c$$

Le formule  $F_1 \wedge F_2, F_1 \vee F_2, \neg F_1$  hanno valore di verità definiti dalle tabelle di verità dei connettivi logici  $\wedge, \vee, \neg$ .

### Proiezione

**Elimina alcuni attributi della relazione.**

Sia  $Y \subseteq X$  dove  $Y = \{A_1, \dots, A_m\}$  con  $m \leq n$

$$\Pi_Y(r) = \begin{cases} \text{schema} & Y \\ \text{istanza} & \{t \mid \exists t' \in r : t = t'[Y]\} \end{cases}$$

Dove:  $t'[Y] = \bar{t}$  essendo  $\bar{t}$  una tupla su  $Y$  tale che  $\forall A_i \in Y_i \quad \bar{t}[A_i] = t'[A_i]$

## Esempio di Proiezione

$$\Pi_{\{Cat\}}(Treno)$$

Questa interrogazione produce l'elenco di tutte le categorie per le quali esiste un treno nella tabella treno.

Posso riscriverla così:

$$\Pi_{Cat}(Treno)$$

## Osservazione sulle cardinalità

### TRENO

| Numero | Cat | Dest | Arr | Part |
|--------|-----|------|-----|------|
|--------|-----|------|-----|------|

$$\rightarrow \Pi_{Cat}(Treno)$$

Sapendo che la tabella treno contiene 150 tuple ( $|treno| = 150$ ), quante tuple contiene il risultato della produzione su Cat ?

$$|\Pi_{Cat}(Treno)| = ?$$

Creo sempre un insieme di tuple omogenee.

So che :  $|\Pi_{Cat}(Treno)| \leq 150$

alcuni duplicati(se ci sono) potrebbero essere eliminati

$$|\Pi_{Numero, Cat}(Treno)| = 150$$

Numero, Cat è una **superchiave**, quindi conservo la cardinalità di partenza.

Cardinalità della proiezione:

- $|\Pi_Y(r)| \leq |r|$  [sempre vero]
- $|\Pi_Y(r)| = |r|$  [se Y è una superchiave]

Cardinalità della selezione:

- $|\sigma_F(r)| \leq |r|$  [sempre vero]
- $|\sigma_F(r)| = 0$  [quando nessuna tupla di R rende vera F]
- $|\sigma_F(r)| = |r|$  [quando tutte le tuple di R rendono vera F]

### 4.3 Operatori di giunzione

Operatore di base: **Join Naturale**

#### Join Naturale

Dipende dallo schema, la **condizione è di uguaglianza tra attributi, richiede una denominazione**.

Siano  $r_1$  e  $r_2$  due relazioni di schema  $R_1(X_1)$  e  $R_2(X_2)$  rispettivamente, il join naturale tra  $r_1$  e  $r_2$  come segue:

$$r_1 \bowtie r_2 = \begin{cases} \text{schema: } X_1 \cup X_2 \\ \text{istanza: } \{t \mid \exists t_1 \in r_1 \wedge \exists t_2 \in r_2 : t_1 = t[X_1] \wedge t_2 = t[X_2]\} \end{cases}$$

#### Esempi di join naturale

| $\bar{r}_1$ | A     | B     | $\bar{r}_2$ | B     | C        |   |
|-------------|-------|-------|-------------|-------|----------|---|
|             | $a_1$ | $b_2$ |             | $b_1$ | $c_0$    | $r_1 \bowtie r_2 = r \rightarrow \text{schema:}\{A, B, C\}$ |
|             | $a_2$ | $b_2$ |             | $b_2$ | $c_3$    |   |
|             | $a_3$ | $b_1$ |             | $b_3$ | $c_{10}$ |   |
|             |       |       |             | $b_2$ | $c_4$    |   |

Se  $X_1 \cup X_2 = \emptyset$  allora il join naturale produce un risultato che corrisponde a considerare tutte le possibili combinazioni tra le tuple di  $R_1$  e le tuple di  $R_2$ .

Quindi nell'esempio di prima:

|                     | A     | B     | C     |
|---------------------|-------|-------|-------|
| $r_1 \bowtie r_2 =$ | $a_1$ | $b_1$ | $c_0$ |
|                     | $a_2$ | $b_2$ | $c_3$ |
|                     | $a_2$ | $b_2$ | $c_4$ |
|                     | $a_3$ | $b_1$ | $c_0$ |
|                     |       |       |       |

Altro esempio di join naturale:

| $\bar{\bar{r}}_1$                           | A     | B     | $\bar{\bar{r}}_2$ | D        | C        |
|---|-------|-------|-------------------|----------|----------|
|   | $a_1$ | $b_1$ |                   | $b_1$    | $c_0$    |
|   | $a_2$ | $b_2$ |                   | $b_2$    | $c_3$    |
|   | $a_3$ | $b_1$ |                   | $b_3$    | $c_{10}$ |
|   | A     | B     | D                 | C        |          |
| $\bar{\bar{r}}_1 \bowtie \bar{\bar{r}}_2 =$ | $a_1$ | $b_1$ | $b_1$             | $c_0$    |          |
|   | $a_1$ | $b_1$ | $b_2$             | $c_3$    |          |
|   | $a_1$ | $b_1$ | $b_3$             | $c_{10}$ |          |
|   | $a_2$ | $b_2$ | $b_1$             | $c_0$    |          |
|   | $a_2$ | $b_2$ | $b_2$             | $c_3$    |          |
|   | $a_2$ | $b_2$ | $b_3$             | $c_{10}$ |          |
|   | $a_3$ | $b_1$ | $b_1$             | $c_0$    |          |
|   | $a_3$ | $b_1$ | $b_2$             | $c_3$    |          |
|   | $a_3$ | $b_1$ | $b_3$             | $c_{10}$ |          |
|   |       |       |                   |          |          |

### Esempio - Esercizio

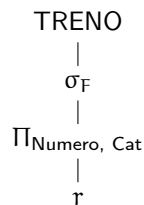
#### TRENO

| Numero | Categoria | Destinazione | Arrivo | Partenza |
|--------|-----------|--------------|--------|----------|
|--------|-----------|--------------|--------|----------|

#### FERMATA

| Numero | Stazione | Orario |
|--------|----------|--------|
|--------|----------|--------|

$Q_1$  : Trova numero e la categoria dei treni che partono dopo le 12 : 00 e prima delle 15 : 00 e non sono regionali.



Dove

$$F = \{\text{Part} > 12 : 00 \wedge \text{Part} < 15 : 00 \wedge \text{Cat} \neq \text{'REG'}\}$$

| r | Numero | Categoria |
|---|--------|-----------|
|   | 1513   | ICO       |
|   | 1514   | FB        |

Esempio di join naturale ( $\bowtie$ )

#### DOCENTE

| codice docente | nome       |
|----------------|------------|
| 123            | Belussi    |
| 456            | Bombieri   |
| 789            | Giacobazzi |

#### INSEGNAMENTO

| nome insegnamento | codice docente |
|-------------------|----------------|
| Basi di Dati      | 123            |
| Programmazione    | 456            |
| Linguaggi         | 789            |

#### DOCENTE $\bowtie$ INSEGNAMENTO

| codice docente | nome       | nome insegnamento |
|----------------|------------|-------------------|
| 123            | Belussi    | Basi di Dati      |
| 456            | Bombieri   | Programmazione    |
| 789            | Giacobazzi | Linguaggi         |

## Proprietà del join naturale

### 1. join completo

Il join naturale  $r_1 \bowtie r_2$ , dove  $r_1$  ha schema  $X_1$  e  $r_2$  ha schema  $X_2$ , si dice *completo* se ogni tupla di  $r_1$  e di  $r_2$  contribuisce a generare una tupla del join  $r_1 \bowtie r_2$ .

$$\forall t_1 \in r_1 : \exists t \in r_1 \bowtie r_2 : t[X_1] = t_1 \quad \wedge \quad \forall t_2 \in r_2 : \exists t' \in r_1 \bowtie r_2 : t'[X_2] = t_2$$

Se una tupla di  $r_1$  (o  $r_2$ ) non contribuisce al join allora si dice **dangling tuple (tupla appesa)**

### 2. cardinalità del join naturale:

$$0 \leq |r_1 \bowtie r_2| \leq |r_1| \cdot |r_2|$$

Se il join è *completo* allora:

$$\max(|r_1|, |r_2|) \leq |r_1 \bowtie r_2| \leq |r_1| \cdot |r_2|$$

Se  $X_1 \cap X_2$  è una *superchiave* per  $r_2$ :

$$0 \leq |r_1 \bowtie r_2| \leq |r_1|$$

Se  $X_1 \cap X_2$  è una *superchiave* per  $r_2$  e esiste un vincolo di integrità referenziale tra gli attributi  $X_1 \cap X_2$  di  $r_1$  (o una parte di  $X_1 \cap X_2$ ) e la chiave primaria di  $r_2$ :

$$|r_1 \bowtie r_2| = |r_1|$$

### 3. altre proprietà:

il join naturale è *commutativo*

$$r_1 \bowtie r_2 = r_2 \bowtie r_1$$

il join naturale è *associativo*

$$r_1 \bowtie (r_2 \bowtie r_3) = (r_1 \bowtie r_2) \bowtie r_3$$

Se  $X_1 = X_2$ , vale a dire che se le due relazioni hanno lo stesso schema:

$$r_1 \bowtie r_2 = r_1 \cap r_2$$

Se  $X_1 \cap X_2 = \emptyset$ , vale a dire che se non esistono attributi comuni tra le due relazioni:

$$r_1 \bowtie r_2 = r_1 \times r_2 \quad (\text{prodotto cartesiano})$$

Si osservi che  $|r_1 \times r_2| = |r_1| \cdot |r_2|$



## Algebra con valori nulli

È opportuno estendere l'algebra relazionale affinché possa manipolare anche i valori nulli. Le operazioni che devono essere raffinate in presenza di valori nulli sono:

- **Selezione:** le condizioni di selezione in presenza di valori nulli hanno i seguenti valori di verità:
  - $A \Theta B$  sulla tupla  $t$ : se  $t[A]$  o  $t[B]$  sono NULL allora  $t[A] \Theta t[B]$  è FALSO.
  - $A \Theta \text{const}$  sulla tupla  $t$ : se  $t[A]$  è NULL allora  $t[A] \Theta \text{const}$  è FALSO.
  - Condizioni atomiche aggiuntive:  $A \text{ is null } / A \text{ is not null}$
- **Join naturale:** la condizione di uguaglianza sugli attributi comuni alle due relazioni è falsa su  $t_1$  e  $t_2$  se almeno uno degli attributi comuni di  $t_1$  o  $t_2$  è NULL.

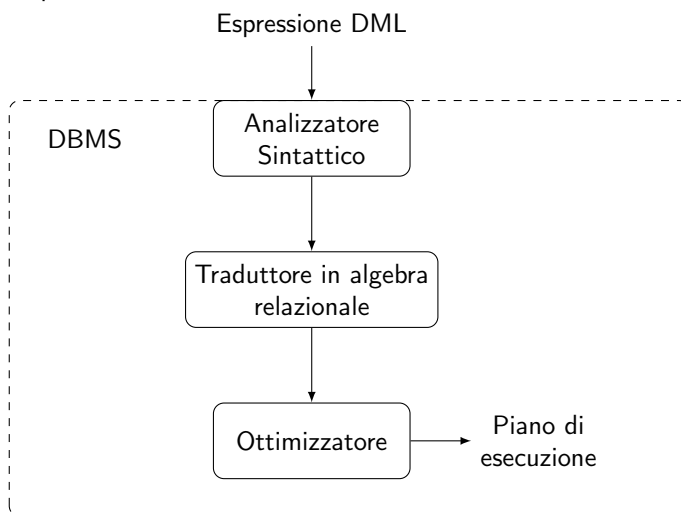
## Join Esterni

Consentono di ottenere nel risultato del join tutte le tuple (anche le tuple pendenti "dangling tuples") di una o di entrambe le relazioni coinvolte nel join, eventualmente estese con valori nulli.

- LEFT JOIN :  $r_1 \bowtie_{\text{LEFT}} r_2$
- RIGHT JOIN :  $r_1 \bowtie_{\text{RIGHT}} r_2$
- FULL JOIN :  $r_1 \bowtie_{\text{FULL}} r_2$

## 5 Ottimizzazione di espressioni DML

Ogni espressione DML (solitamente specificata in linguaggio dichiarativo) ricevuta dal DBMS è soggetta ad un processo di elaborazione.



### 5.1 Ottimizzatore

L'ottimizzatore genera un'espressione equivalente all'interrogazione di input e di costo inferiore.

Il costo viene valutato in termini di **dimensione dei risultati intermedi**.

L'ottimizzatore esegue **trasformazioni di equivalenza** allo scopo di RIDURRE LA DIMENSIONE DEI RISULTATI INTERMEDI.

## 5.2 Equivalenza tra espressioni algebriche

- Equivalenza dipendente dallo schema: dato uno schema  $R$

$$E_1 \equiv_R E_2 \quad \text{se } E_1(r) = E_2(r) \quad \text{per ogni istanza } r \text{ di schema } R$$

- Equivalenza assoluta: è indipendente dallo schema

$$E_1 \equiv E_2 \quad \text{se } E_1 \equiv_R E_2 \quad \text{per ogni schema } R \text{ compatibile con } E_1 \text{ e } E_2$$

## 5.3 Trasformazioni di equivalenza

Sia  $E$  un'espressione di schema  $X$ , si definiscono le seguenti trasformazioni di equivalenza:

- **Atomizzazione delle selezioni**

$$\sigma_{F_1 \wedge F_2}(E) \equiv \sigma_{F_1}(\sigma_{F_2}(E))$$

È propedeutica ad altre trasformazioni. Non ottimizza se non è seguita da altre trasformazioni.

- **Idempotenza delle proiezioni**

$$\Pi_Y(E) \equiv \Pi_Y(\Pi_{YZ}(E)) \quad \text{dove } Z \subseteq X$$

È propedeutica ad altre trasformazioni. Non ottimizza se non è seguita da altre trasformazioni.

Siano  $E_1$  e  $E_2$  espressioni di schema  $X_1$  e  $X_2$ , si definiscono le seguenti trasformazioni di equivalenza:

- **Anticipazione delle selezioni rispetto al join**

$$\sigma_F(E_1 \bowtie E_2) \equiv E_1 \bowtie \sigma_F(E_2)$$

Applicabile solo se  $F$  si riferisce solo ad attributi di  $E_2$ .

- **Anticipazione della proiezione rispetto al join**

$$\Pi_{X_1 Y}(E_1 \bowtie E_2) \equiv_R E_1 \bowtie \Pi_Y(E_2)$$

Applicabile solo se  $Y \subseteq X_2$  e  $(X_2 - Y) \cup X_1 = \emptyset$

Combinando l'anticipazione della proiezione con l'idempotenza delle proiezioni otteniamo:

$$\Pi_Y(E_1 \bowtie_F E_2) \equiv \Pi_Y(\Pi_{Y_1}(E_1) \bowtie_F \Pi_{Y_2}(E_2))$$

$$\Pi_Y(E_1 \bowtie E_2) \equiv \Pi_Y(\Pi_{Y_1}(E_1) \bowtie \Pi_{Y_2}(E_2))$$

dove :

- $Y_1 = (X_1 \cap Y) \cup J_1$
- $Y_2 = (X_2 \cap Y) \cup J_2$
- $J_1/2$  sono gli attributi di  $E_1/2$  coinvolti nel join (vale a dire presenti in  $F$  per il theta-join, mentre in caso di join naturale  $J_1 = J_2 = X_1 \cap X_2$ )



## 5.4 Ulteriori trasformazioni di equivalenza

Siano  $E_1$  e  $E_2$  espressioni di schema  $X_1$  e  $X_2$ , si definiscono le seguenti trasformazioni di equivalenza:

- Inglobamento di una selezione in un prodotto cartesiano (**attenzione questa regola si applica solo dopo aver verificato che non sia possibile anticipare selezioni rispetto al join**):

$$\sigma_F(E_1 \bowtie E_2) \equiv E_1 \bowtie_F E_2$$

dove  $X_1 \cap X_2 = \emptyset$ .

- Applicazione delle proprietà commutativa e associativa di: unione, prodotto cartesiano, intersezione.
- Applicazione della proprietà distributiva:

$$\sigma_F(E_1 \cup E_2) \equiv \sigma_F(E_1) \cup \sigma_F(E_2)$$

$$\sigma_F(E_1 - E_2) \equiv \sigma_F(E_1) - \sigma_F(E_2)$$

$$\Pi_Y(E_1 \cup E_2) \equiv \Pi_Y(E_1) \cup \Pi_Y(E_2)$$

$$E_1 \bowtie (E_2 \cup E_3) \equiv (E_1 \bowtie E_2) \cup (E_1 \bowtie E_3)$$

- Applicazione di altre trasformazioni:

$$\sigma_{F_1 \vee F_2}(E) \equiv \sigma_{F_1}(E) \cup \sigma_{F_2}(E)$$

$$\sigma_{F_1 \wedge F_2}(E) \equiv \sigma_{F_1}(E) \cap \sigma_{F_2}(E) \equiv \sigma_{F_1}(E) \bowtie \sigma_{F_2}(E)$$

$$\sigma_{F_1 \wedge \neg F_2}(E) \equiv \sigma_{F_1}(E) - \sigma_{F_2}(E)$$

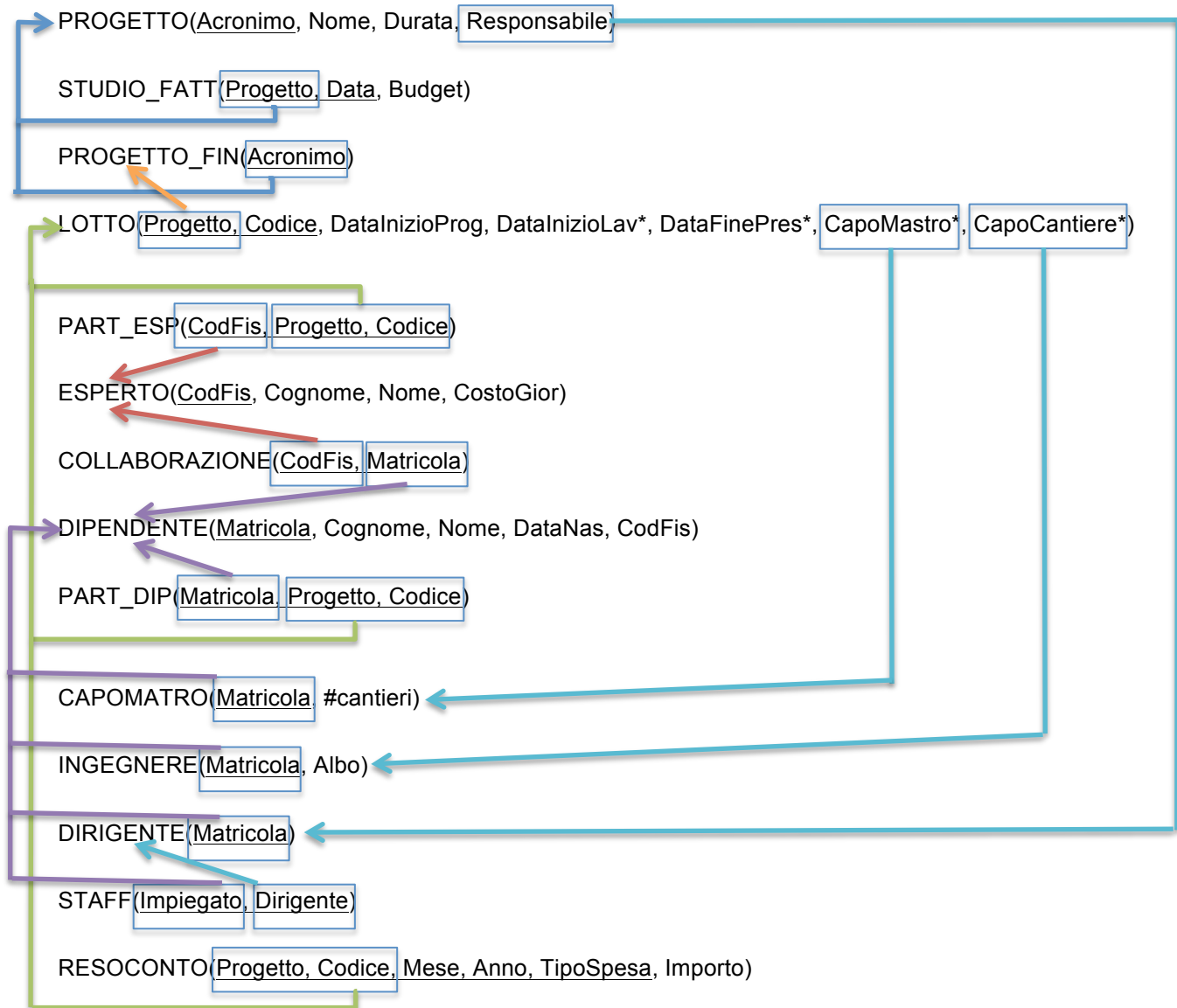
# Basi di dati

## Esercitazione in preparazione della seconda prova intermedia

2016/2017

Sia assegnato il seguente schema risultato della progettazione logica:

### Schema relazionale



1. Dato lo schema relazionale sopra riportato, esprimere in algebra relazionale ottimizzata le seguenti interrogazioni:
  - a) Trovare i progetti che hanno almeno due lotti con data inizio progettazione nel mese di gennaio 2016 e data di inizio cantiere nel mese di giugno 2016 riportando l'acronimo del progetto e il nome e cognome del responsabile del progetto
  - b) Trovare gli ingegneri che non sono mai stati assegnati come capo cantiere del lotto di un progetto, riportando la matricola, il nome, il cognome dell'ingegnere e il codice fiscale degli esperti esterni con cui collabora.
  - c) Trovare per ogni lotto, a cui partecipa un esperto esterno di cognome "Rossi", l'elenco delle spese sostenute nel mese di gennaio 2016 riportando l'acronimo del progetto, il codice del lotto, il tipo di spesa e l'importo.
  - d) Trovare il codice fiscale e il cognome dei dipendenti che hanno partecipato a tutti i lotti di almeno un progetto, riportando anche l'acronimo di tale progetto.
  - e) Trovare il dirigente più giovane, riportando il suo nome, cognome e data di nascita

2. Dato il seguente schema relazionale:

LABORATORIO(codLab, nome, numeroAddetti);

ESAME(codLab, paziente, resoconto, urgenza, dataEsame, oraEsame)

PAZIENTE(tesseraSanitaria, nome, cognome, nazionalità)

Vincoli di integrità: ESAME.codLab → LABORATORIO,  
ESAME.paziente → PAZIENTE

Supponendo che le relazioni abbiano le seguenti cardinalità:

- LABORATORIO: 150
- ESAME: 120000
- PAZIENTE: 85000

e che per ogni paziente si sia registrato almeno un esame, indicare la cardinalità minima e massima dei risultati delle seguenti interrogazioni:

Q<sub>1</sub>  $\Pi_{\{\text{codLab}, \text{paziente}, \text{urgenza}\}} (\text{LABORATORIO} \bowtie \text{ESAME})$

Q<sub>2</sub>  $\Pi_{\{\text{nome}\}} (\text{LABORATORIO}) \cup \Pi_{\{\text{nome}\}} (\text{PAZIENTE})$

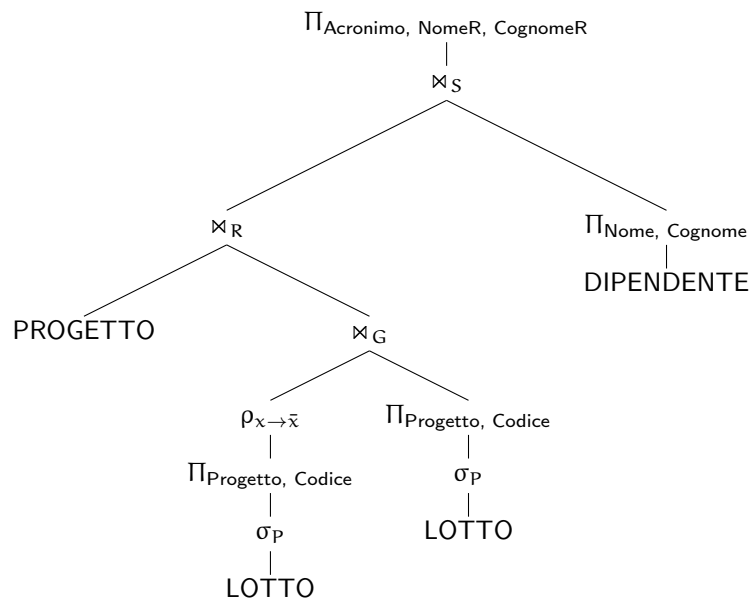
Q<sub>3</sub>  $\Pi_{\{\text{nazionalità}\}} ( (\text{ESAME} \bowtie_{\text{paziente} = \text{tesseraSanitaria}} \text{PAZIENTE}) - ( \text{ESAME} \bowtie_{\text{paziente} = \text{tesseraSanitaria}} \sigma_{\text{NOT nazionalità}='italiana'} \text{PAZIENTE} ) )$

3. Dato lo schema relazionale dell'esercizio 2 scrivere in SQL le seguenti interrogazioni:

- Trovare il nome e il cognome dei pazienti di nazionalità non italiana che ieri hanno fatto almeno due esami urgenti in due laboratori diversi.
- Trovare il nome e il numero di addetti dei laboratori che il giorno 1/1/2017 non hanno eseguiti alcun esame.
- Trovare il paziente che ha fatto il numero massimo di esami, riportando nel risultato il cognome e la nazionalità del paziente.
- Trovare per ogni laboratorio il numero di esami fatti e il numero di pazienti esaminati in ogni mese del 2016, riportando nel risultato il codice e il nome del laboratorio insieme ai conteggi richiesti.

## 6 Soluzione dell'esercitazione

### 6.1 Esercizio 1.a



dove:

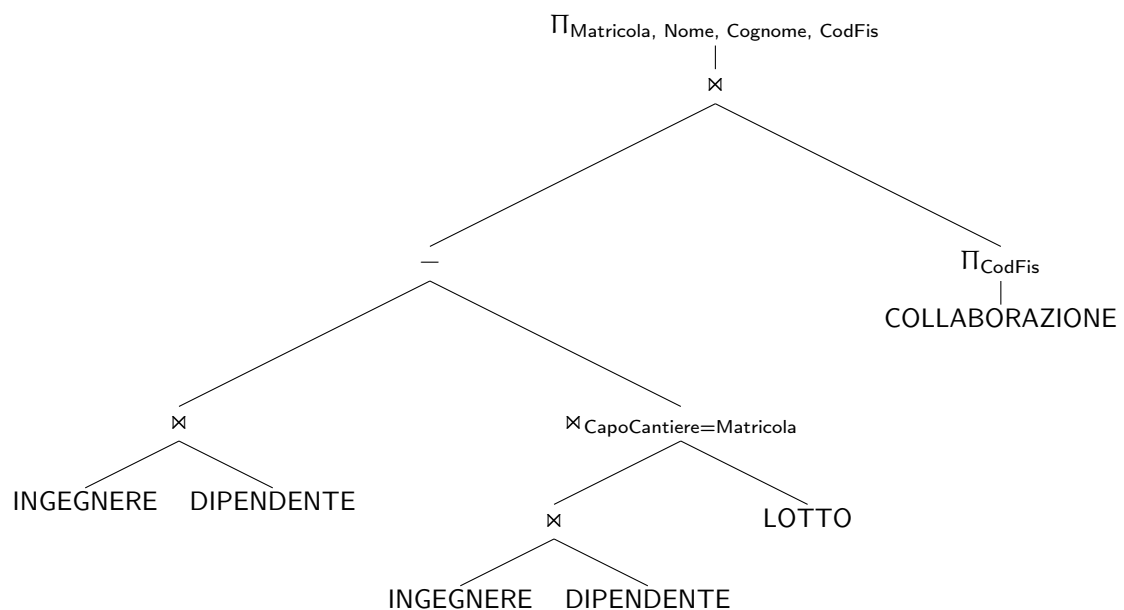
$S = \text{Matricola, Nome, Cognome} \rightarrow \text{Responsabile, NomeR, CognomeR}$

$R = \text{Progetto} = \text{Acronimo}$

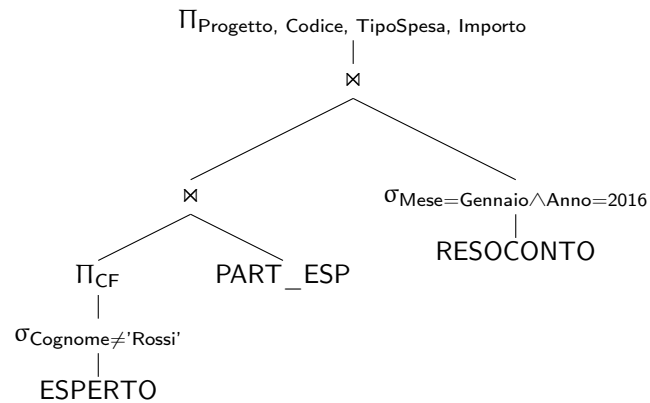
$G = \text{Progetto} = \overline{\text{Progetto}} \wedge \text{Codice} \neq \overline{\text{Codice}}$

$$P = \begin{cases} \text{DataInizioProg} \geq '1/1/2016' \\ \text{DataInizioProg} \leq '31/1/2016' \\ \text{DataInizioLav} \geq '1/6/2016' \\ \text{DataInizioLav} \leq '30/6/2016' \end{cases}$$

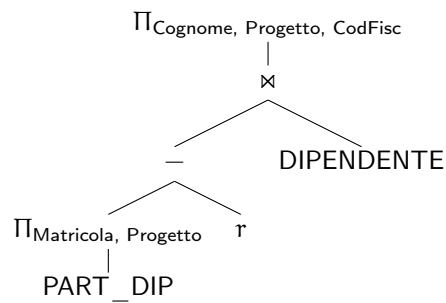
### 6.2 Esercizio 1.b



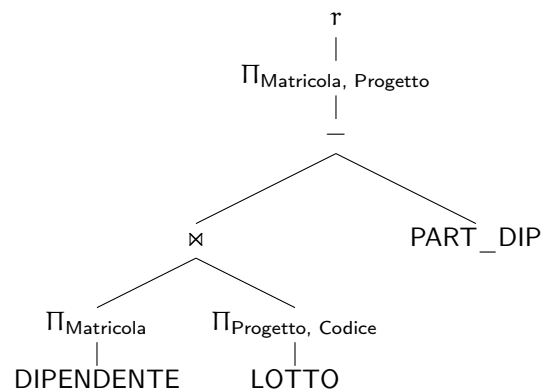
## 6.3 Esercizio 1.c



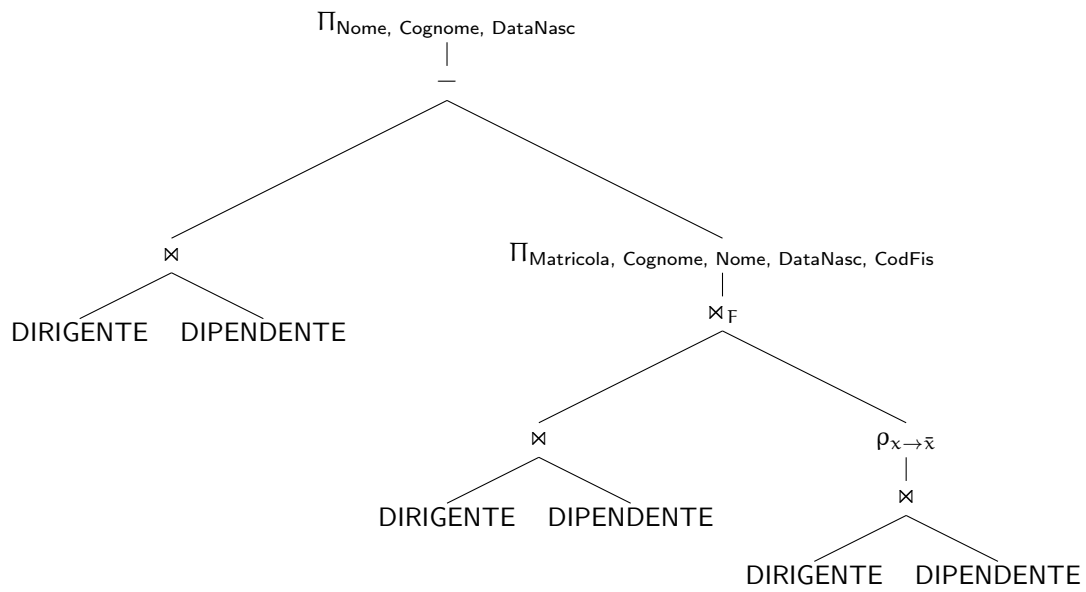
## 6.4 Esercizio 1.d



Dove  $r$  rappresenta tutte le tuple Matricola, Progetto di dipendenti che non hanno partecipato ad almeno uno dei lotti del progetto.



## 6.5 Esercizio 1.e



Dove

$$F = \overline{\text{DataNasc}} > \text{DataNasc} \wedge \text{Matricola} \neq \overline{\text{Matricola}}$$

## 7 Concetto di vista

È una relazione derivata. Si specifica l'espressione che genera il suo contenuto. Esso quindi dipende dalle relazioni che compaiono nell'espressione.

**Osservazione.** Una vista può essere funzione di altre viste purché esista un ordinamento in grado di guidare il calcolo delle relazioni derivate.

### Vista virtuale

Una vista si dice "virtuale" se viene calcolata ogni volta che serve. Conviene quando:

- L'aggiornamento è frequente
- L'interrogazione che genera la vista è semplice

Utilità delle viste:

- Consentono di personalizzare l'interfaccia utente
- Facilitano la gestione della privacy dei dati
- Permettono di memorizzare nel DBMS interrogazioni complesse condivise
- Sono utili per rendere l'interfaccia delle applicazioni indipendente dallo schema logico (INDIPENDENZA LOGICA)

### Vista materializzata

Una vista si dice "materializzata" se viene calcolata e memorizzata esplicitamente nella base di dati. Conviene materializzare quando:

- L'aggiornamento è raro
- L'interrogazione che genera la vista è complessa

Se la vista viene materializzata è necessario ricalcolarne il contenuto ogni volta che le relazioni da cui dipende vengono aggiornate.

## 8 SQL - Structured Query Language

SQL è un linguaggio che è stato definito negli anni '70; è stato standardizzato negli anni '80 e '90 ed è oggi il linguaggio più diffuso per i DBMS relazionali. rappresenta uno strumento per l'interazione e la gestione della base di dati. È un linguaggio di **interrogazione dichiarativo**, cioè precisa le proprietà che devono essere rispettate dal risultato dell'interrogazione (query). Si basa sul calcolo relazionale e sulla logica del prim'ordine, in cui ogni espressione è dichiarativa e quindi definisce un insieme di proprietà.

### 8.1 Sintassi

```
SELECT <Lista Attributi>  
FROM <Lista Tabelle>  
[ WHERE <Condizione> ]
```

La sua esecuzione produce una relazione risultato con le seguenti caratteristiche:

- **Schema:** è costituito da tutti gli attributi indicati in <ListaAttributi>
- **Contenuto:** è costituito da tutte le tuple  $t$  ottenute proiettando sugli attributi di <ListaAttributi> (clausola SELECT) le tuple  $t'$  appartenenti alle tabelle indicate in <ListaTabelle> (clausola FROM) che soddisfano l'eventuale condizione <Condizione> (clausola WHERE).

## 8.2 Lista dei principali comandi

```
SELECT DISTINCT <...> AS <alias> | * |
                COUNT (* | DISTINCT ... | ALL ...)
FROM <table/s> [AS] <alias>
[ WHERE <espr> ]
```

### Clausola ORDER BY

```
ORDER BY <attr> [ASC | DESC] {, <attr> [ASC | DESC]}
```

### Interrogazioni nidificate

```
SELECT ...
FROM ...
WHERE <espr> <predicato> <ALL | ANY> (SELECT ... FROM ... WHERE)
```

dove predicato complesso è dell'insieme {=, <>, <, >, <=, >=} più ALL o ANY:

- A op ALL : questo predicato è soddisfatto dalla tupla t se esiste almeno un valore v nel risultato dell'interrogazione che verifica la condizione.  
= **ANY** si può scrivere IN.
- A op ANY : questo predicato è soddisfatto dalla tupla t se per ogni valore v nel risultato dell'interrogazione è verificata la condizione  
<> **ALL** si può scrivere NOT IN.

### Clausola EXISTS

```
SELECT ...
FROM ...
WHERE <espr> AND <EXISTS | NOT EXISTS> (SELECT 1 FROM ... WHERE)
```

### Operatori aggregati

```
COUNT(* | [ DISTINCT | ALL ] <listaAttributi> )
SUM | MAX | MIN | AVG ([ DISTINCT | ALL ] <espressione>)
```

### Clausola GROUP BY e HAVING

```
SELECT <listaAttributi> FROM ... WHERE ...
GROUP BY <Attributo> {, <Attributo>}
HAVING <condizione_sel_gruppi>
ORDER BY ...
```

### Interrogazioni insiemistiche

```
<selectSQL> { UNION | INTERSECT | EXCEPT [all] <selectSQL> }
```

### Viste

```
CREATE VIEW <nomeVista> [( <lista attributi> )] AS <selectSQL>
```

- Non è possibile definire viste ricorsive.
- È possibile definire una vista utilizzando altre viste, ma evitando dipendenze circolari (esempio di dipendenza circolare: V1 definita da V2 definita da V3 definita da V1).
- Le viste non sono in generale aggiornabili (solo in casi particolari i DBMS ammettono l'aggiornamento di viste).



## 9 Transazioni

### Principale caratteristica di un DBMS:

Un DBMS è un sistema transazionale, cioè fornisce un meccanismo per la definizione ed esecuzione di **transazioni**.

**Definizione. Transazione:** È un'unità di lavoro svolto da un programma applicativo (che interagisce con una base di dati) per la quale si vogliono garantire alcune proprietà.

Una transazione o va a buon fine e ha effetto sulla base di dati o abortisce e non ha nessun effetto sulla base di dati.

**O tutto o niente!**

### 9.1 Sintassi

transazione:

```
begin transaction
<programma>
end transaction
```

programma:

```
<istruzione> | commit work | rollback work
{<istruzione> | commit work | rollback work}
```

La transazione va a buon fine all'esecuzione di un `commit work`. Non ha invece effetto se viene eseguito un `rollback work`.

Una transazione è **ben formata** se:

- Inizia con un `begin transaction`.
- Termina con un `end transaction`.
- La sua esecuzione comporta il raggiungimento di un `commit` o di un `rollback work` e dopo il `commit/rollback` non si eseguono altri accessi alla base di dati.

Esempio di transazione ben formata:

```
begin transaction;
    update CONT0 set saldo = saldo - 1200
        where filiale = '005' and numero = 15;
    update CONT0 set saldo = saldo + 1200
        where filiale = '005' and numero = 105;
commit work;
end transaction;
```

### 9.2 Proprietà

Una transazione ha quattro proprietà:

|                |                    |
|----------------|--------------------|
| 1. Atomicità   | <b>Atomicity</b>   |
| 2. Consistenza | <b>Consistency</b> |
| 3. Isolamento  | <b>Isolation</b>   |
| 4. Persistenza | <b>Durability</b>  |

Un DBMS che gestisce transazioni dovrebbe garantire per ogni transazione che esegue tutte queste proprietà.

### 9.3 Atomicità

Una transazione è una unità di esecuzione **indivisibile**. O viene eseguita completamente o non viene eseguita affatto.

Implicazioni:

- Se una transazione viene interrotta prima del commit, il lavoro fin qui eseguito dalla transazione deve essere disfatto ripristinando la situazione in cui si trovava la base di dati prima dell'inizio della transazione.
- Se una transazione viene interrotta all'esecuzione del commit (commit eseguito con successo), il sistema deve assicurare che la transazione abbia effetto sulla base di dati.

### 9.4 Consistenza

L'esecuzione di una transazione non deve violare i vincoli di integrità.

Implicazioni: Al verificarsi della violazione di un vincolo il sistema può:

- **verifica immediata:**

Viene abortita l'ultima operazione e il sistema restituisce all'applicazione una segnalazione d'errore. L'applicazione può quindi reagire alla violazione.

- **verifica differita:**

Al commit se un vincolo di integrità viene violato la transazione viene abortita senza possibilità da parte dell'applicazione di reagire alla violazione.

### 9.5 Isolamento

L'esecuzione di una transazione deve essere **indipendente** dalla contemporanea esecuzione di altre transazioni.

Implicazioni:

- Il rollback di una transazione non deve creare rollback a catena di altre transazioni che si trovano in esecuzione contemporaneamente.
- Il sistema deve regolare l'esecuzione concorrente

### 9.6 Persistenza

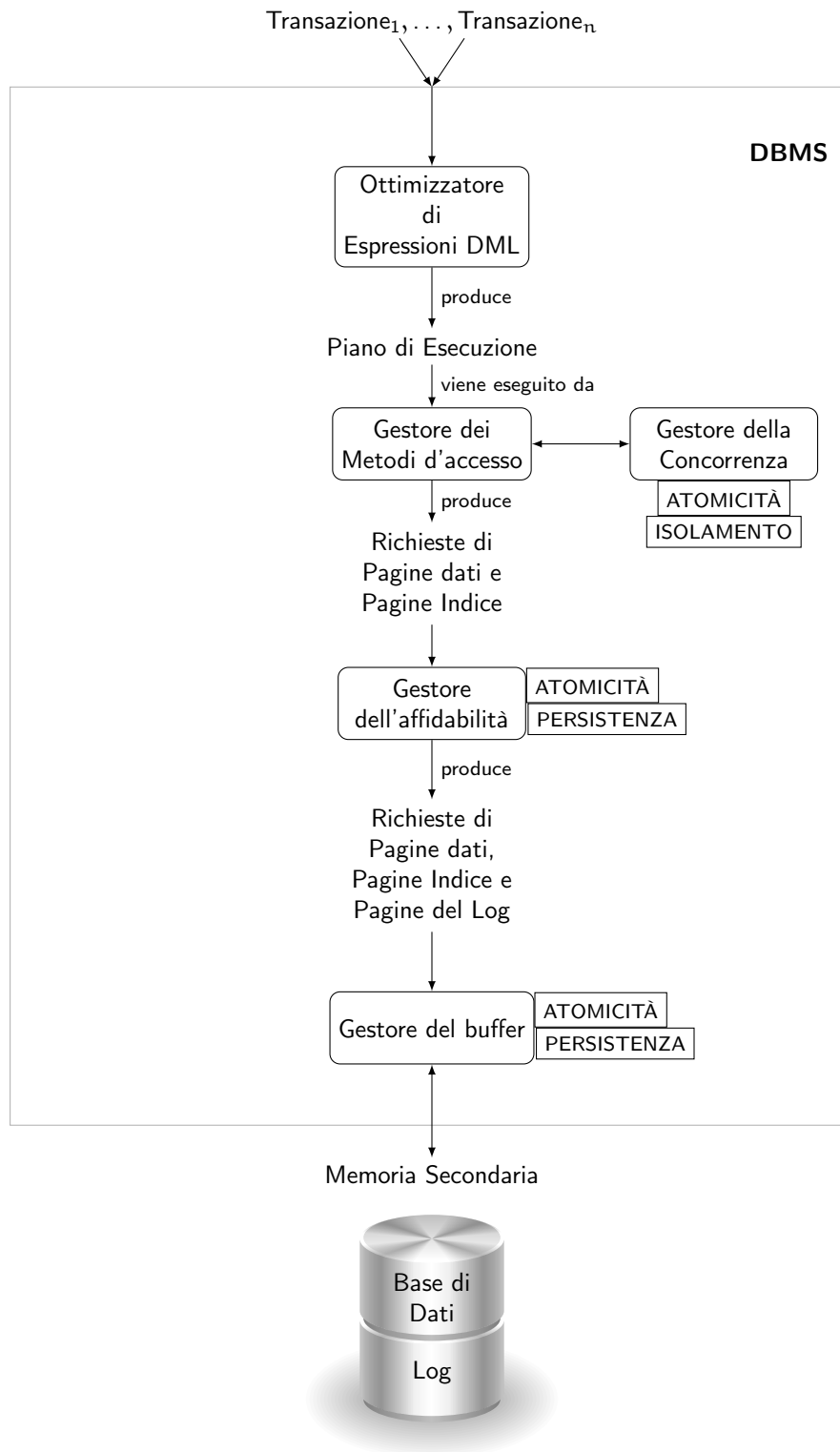
L'effetto di una transazione che ha eseguito il commit non deve andare perso.

Implicazioni:

- Il sistema deve essere in grado, in caso di guasto, di garantire gli effetti delle transazioni che al momento del guasto avevano già eseguito un commit.

## 10 Architettura di un DBMS

L'architettura mostra i moduli principali che possiamo individuare nei DBMS attuali, considerando le diverse funzionalità che il DBMS svolge durante l'esecuzione delle transazioni. Per ogni modulo dell'architettura presentiamo le funzionalità che esso svolge e alcune delle tecniche che applica.



## 11 Strutture fisiche e di accesso ai dati

Le basi di dati gestite da un DBMS risiedono in memoria secondaria, in quanto sono:

- Grandi: non possono essere contenute in memoria centrale
- Persistenti: hanno un tempo di vita che non è limitato all'esecuzione dei programmi che le utilizzano

Caratteristiche della memoria secondaria:

- Non è direttamente utilizzabile dai programmi
- I dati sono organizzati in blocchi (o pagine)
- Le uniche operazioni possibili sono la lettura e la scrittura di un intero blocco (pagina)
- Il costo di tali operazioni è ordini di grandezza maggiore del costo per accedere ai dati in memoria centrale

### 11.1 Gestore del buffer

L'interazione tra memoria secondaria e memoria centrale avviene attraverso il trasferimento di pagine della memoria secondaria in una zona appositamente dedicata della memoria centrale detta **buffer**.

Si noti che:

- Il buffer è una zona di memoria condivisa dalle applicazioni
- Quando uno stesso dato viene utilizzato più volte in tempi ravvicinati il buffer evita l'accesso alla memoria secondaria
- La gestione ottimale del buffer è strategica per ottenere buone prestazioni nell'accesso ai dati

|           |           |           |           |           |           |
|-----------|-----------|-----------|-----------|-----------|-----------|
| $B_{2,0}$ | L         | L         | $B_{1,0}$ | L         | L         |
| $B_{1,1}$ | $B_{1,1}$ | $B_{1,1}$ | L         | L         | L         |
| L         | L         | $B_{1,0}$ | L         | $B_{0,0}$ | $B_{0,0}$ |
| L         | L         | L         | L         | $B_{0,0}$ | L         |
| L         | L         | L         | $B_{0,0}$ | $B_{0,0}$ | L         |
| L         | L         | L         | L         | L         | L         |



Memoria secondaria

$B_{i,j}$  indica che nella pagina del buffer è caricato il blocco  $B$ , inoltre  $i$  indica che il blocco è attualmente utilizzato da  $i$  transazioni mentre  $j$  è 1 se il blocco è stato modificato e 0 altrimenti.

L indica pagina libera.

Il buffer è organizzato in **pagine**. Una pagina ha le dimensioni di un blocco della memoria secondaria. Il gestore del buffer si occupa del caricamento/salvataggio delle pagine in memoria secondaria.

**Politica** del gestore dei buffer:

- In caso di richiesta di lettura di un blocco, se il blocco è presente in una pagina del buffer allora non si esegue una lettura su memoria secondaria e si restituisce un puntatore alla pagina del buffer, altrimenti si cerca una pagina libera e si carica il blocco nella pagina, restituendo il puntatore alla pagina stessa.

- In caso di richiesta di scrittura di un blocco precedentemente caricato in una pagina del buffer, il gestore del buffer può decidere di differire la scrittura su memoria secondaria in un secondo momento.

In entrambi i casi (lettura e scrittura) l'obiettivo è quello di aumentare la velocità di accesso ai dati. Tale comportamento del gestore dei buffer si basa sul **principio di località**:

*"I dati referenziati di recente hanno maggiore probabilità di essere referenziati nuovamente in futuro".*

Inoltre, una nota legge empirica dice che: "il 20% dei dati è acceduto dall'80% delle applicazioni". Tutto ciò rende conveniente dilazionare la scrittura su memoria secondaria delle pagine del buffer.

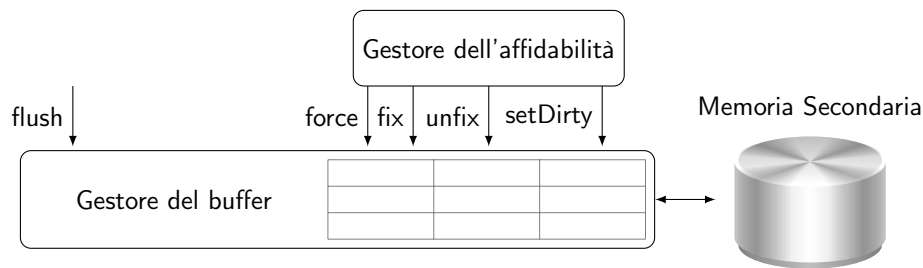
## 11.2 Gestione delle pagine

Dati necessari per la gestione delle pagine:

- Per ogni pagina del buffer si memorizza il blocco contenuto indicando il file e il numero di blocco (o offset)
- Per ogni pagina del buffer si memorizza un insieme di variabili di stato tra cui si trovano sicuramente:
  - Un contatore per indicare il numero di transazioni che utilizzano le pagine
  - Un bit di stato per indicare se la pagina è stata modificata o no.

Primitive per la gestione delle pagine:

- **Fix**: viene usata dalle transazioni per richiedere l'accesso ad un blocco e restituisce al chiamante un puntatore alla pagina contenente il blocco richiesto. Passi della primitiva:
  - Il blocco richiesto viene cercato nelle pagine del buffer, in caso sia presente, si restituisce un puntatore alla pagina contenente il blocco
  - Altrimenti, viene scelta una pagina libera P (contatore = zero). Si sceglie la pagina in base a criteri diversi: la pagina usata meno di recente (LRU) o caricata da più tempo (FIFO). Se il bit di stato di P è a 1 P viene salvata in memoria secondaria (operazione di flush). Si carica il blocco in P aggiornando file e numero di blocco corrispondenti.
  - Se non esistono pagine libere, il gestore del buffer può adottare due politiche (STEAL): "ruba" una pagina ad un'altra transazione (operazione di flush); oppure (NO STEAL) sospende la transazione inserendola in una coda di attesa. Se una pagina si libera (contatore = zero) il gestore si comporterà come al punto precedente.
  - Quando una transazione accede ad una pagina per la prima volta il contatore si incrementa.
- **unfix**: viene usata dalle transazioni per indicare che la transazione ha terminato di usare il blocco. L'effetto è il decremento del contatore di utilizzo della pagina.
- **setDirty**: viene usata dalle transazioni per indicare al gestore del buffer che il blocco della pagina è stato modificato: L'effetto è la modifica del bit di stato a 1
- **force**: viene usata per salvare in memoria secondaria in modo **sincrono** il blocco contenuto nella pagina (primitiva usata dal gestore dell'affidabilità). L'effetto è il salvataggio in memoria secondaria del blocco e il bit di stato posto a zero.
- **flush**: viene usata dal gestore del buffer per salvare blocchi sulla memoria secondaria in modo **asincrono**. Tale operazione libera pagine "dirty".



### 11.3 Gestore dell'affidabilità

È il modulo responsabile di ciò che riguarda:

- L'esecuzione delle istruzioni per la gestione delle transazioni
- La realizzazione delle operazioni necessarie al ripristino della base di dati dopo eventuali malfunzionamenti

Per il suo funzionamento il gestore dell'affidabilità deve disporre di un dispositivo di **memoria stabile**. **memoria stabile** = memoria **resistente ai guasti**

In memoria stabile viene memorizzato il file di **log** che registra in modo sequenziale le operazioni eseguite dalle transazioni sulla base di dati.

#### Record di log di transizione

- Begin della transazione T: record B(T)
- Commit della transazione T: record C(T)
- Abort della transazione T: record A(T)
- Insert, Delete e Update eseguiti dalla transazione T sull'oggetto O:
  - record I(T,O,AS): AS=After State
  - record D(T,O,BS): BS=Before State
  - record U(T,O,BS,AS)

I record di transazione salvati nel LOG consentono di eseguire in caso di ripristino le seguenti operazioni:

- **undo**: per disfare un'azione su un oggetto O è sufficiente ricopiare in O il valore BS; l'insert/delete viene disfatto cancellando/inserendo O;
- **redo**: per rifare un'azione su un oggetto O è sufficiente ricopiare in O il valore AS; l'insert/delete rifatto inserendo/cancellando O;

Gli inserimenti controllano sempre l'esistenza di O (non si inseriscono duplicati).

Proprietà di **idempotenza**:

- $UNDO(UNDO(A))=UNDO(A)$
- $REDO(REDO(A))=REDO(A)$

#### Record di log di sistema

- Operazione di **dump** della base di dati: **record di DUMP**. Contiene un riepilogo della struttura delle tabelle del database medesimo e/o i relativi dati, ed è normalmente nella forma di una lista di dichiarazioni SQL. Tale dump è usato per lo più per fare il backup del database, poiché i suoi contenuti possono essere ripristinati nel caso di perdita di dati. I database "corrotti" (ossia, i cui dati non sono più utilizzabili in seguito ad una modifica "distruttiva" di qualche parametro di

formato) possono spesso essere rigenerati mediante l'analisi del dump. I dumps di database sono spesso pubblicati dai progetti di software libero o di contenuto libero, in modo tale da consentire il riutilizzo o il forking dei database cui si riferiscono. Con l'espressione *core dump* s'intende specificamente lo stato registrato della memoria di un programma in un determinato momento, specie quando tale programma si sia chiuso "inaspettatamente" (crash).

- Operazione di **CheckPoint**: record  $CK(T_1, \dots, T_n)$  indica che all'esecuzione del CheckPoint le transazioni attive erano  $T_1, \dots, T_n$ .

L'operazione di CheckPoint: è un'operazione svolta periodicamente dal gestore dell'affidabilità; prevede i seguenti passi:

- Sospensione delle operazioni di scrittura, commit e abort delle transazioni.
- Esecuzione di operazione di force sulle pagine "dirty" di transazioni che hanno eseguito il commit.
- Scrittura sincrona sul file di LOG del record di CheckPoint con gli identificatori delle transazioni attive.
- Ripresa delle operazioni di scrittura, commit e abort delle transazioni.

Regole per la scrittura sul LOG ed esecuzione delle transazioni:

- Regola **WAL** (Write Ahead Log): i record di log devono essere scritti sul LOG prima dell'esecuzione delle corrispondenti operazioni sulla base di dati (garantisce la possibilità di fare sempre UNDO).
- Regola **Commit-Precedenza**: i record di log devono essere scritti sul LOG prima dell'esecuzione del COMMIT della transazione (garantisce la possibilità di fare sempre REDO)

Tali regole consentono di salvare i blocchi delle pagine "dirty" in modo totalmente asincrono rispetto al commit delle transazioni.

### Esecuzione del commit di una transazione

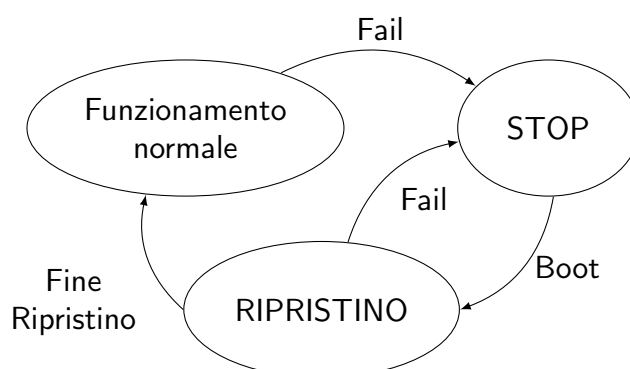
Una transazione sceglie in **modo atomico** l'esito di COMMIT nel momento in cui scrive nel file di LOG in modo **sincrono** (primitiva **force**) il suo record di COMMIT.

### Operazioni di ripristino in caso di guasto

Tipi di guasto:

- **Guasto di sistema**: perdita del contenuto della memoria centrale → **Ripresa a caldo**.
- **Guasto di dispositivo**: perdita di parte o tutto il contenuto della base di dati in memoria secondaria → **Ripresa a freddo**.

Modello di funzionamento:



## Ripresa a caldo

Passi:

- Si accede all'ultimo blocco del LOG e si ripercorre all'indietro il log fino al più recente record CK.
- Si decidono le transazioni da rifare/disfare inizializzando l'insieme UNDO con le transazioni attive al CK e l'insieme REDO con l'insieme vuoto.
- Si ripercorre in avanti il LOG e per ogni record B(T) incontrato si aggiunge T a UNDO e per ogni record C(T) incontrato si sposta T da UNDO a REDO.
- Si ripercorre all'indietro il LOG disfacendo le operazioni eseguite dalle transazioni in UNDO risalendo fino alla prima azione della transazione più vecchia.
- Si rifanno le operazioni delle transazioni dell'insieme REDO.

## Esercizio

B(T1), B(T2), U(T2,01,B1,A1), I(T1,02,A2), B(T3),  
 C(T1), B(T4), U(T3,02,B3,A3), U(T4,03,B4,A4),  
 CK(T2,T3,T4), C(T4), B(T5), U(T3,03,B5,A5),  
 U(T5,04,B6,A6), D(T3,05,B7), A(T3), C(T5),  
 I(T2,06,A8) guasto

Che passi svolge la ripresa a caldo?

| undo                     | redo           |
|--------------------------|----------------|
| T <sub>2</sub>           |                |
| T <sub>3</sub>           |                |
| <del>T<sub>4</sub></del> | T <sub>4</sub> |
| <del>T<sub>5</sub></del> | T <sub>5</sub> |

Undo:

- Delete(06)
- Insert(05)
- 03 := B5;
- 02 := B3;
- 01 := B1;

Redo:

- 03 := A4;
- 04 := A6;



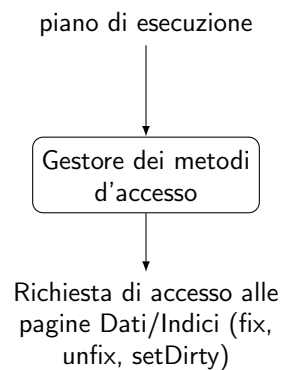
## Ripresa a freddo

Passi:

- Si accede al DUMP della base di dati e si ricopia selettivamente la parte deteriorata della base di dati.
- Si accede al LOG risalendo al record di DUMP.
- Si ripercorre in avanti il LOG rieseguendo tutte le operazioni relative alla parte deteriorata comprese le azioni di commit e abort.
- Si applica una ripresa a caldo.

## 11.4 Gestore dei metodi di accesso

È il modulo del DBMS che esegue il piano di esecuzione prodotto dall'ottimizzatore e produce sequenze di accessi alle pagine della base di dati presenti in memoria secondaria.



## Metodi d'accesso

Sono i moduli software che implementano gli algoritmi di accesso e manipolazione dei dati organizzati in specifiche strutture fisiche.

Esempio:

- Scansione sequenziale
- Accesso via indice
- Ordinamento
- Varie implementazioni del join

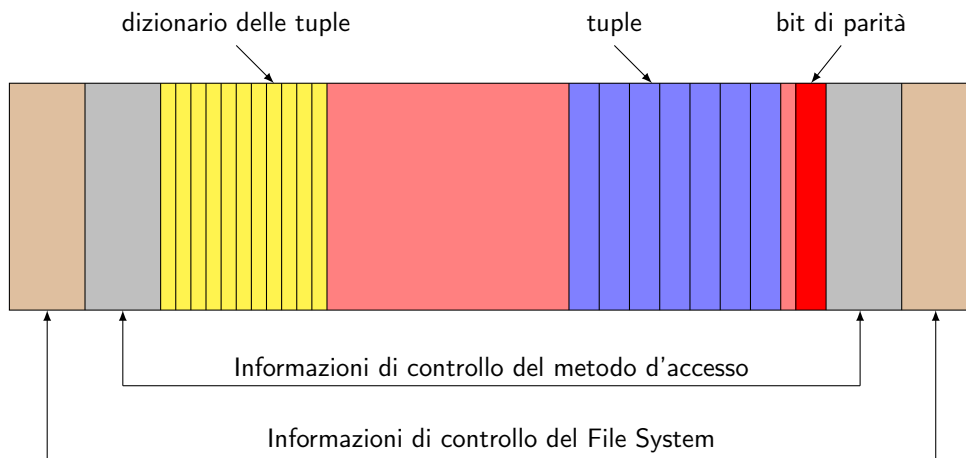
Ogni metodo d'accesso ai dati conosce:

- L'organizzazione delle **tuple** nelle pagine DATI salvate in memoria secondaria (come una tabella viene organizzata in pagine DATI della memoria secondaria)
- L'organizzazione fisica delle **pagine** DATI contenenti tuple e delle pagine che memorizzano le strutture fisiche di accesso o INDICI (come le tuple o i record dell'indice vengono memorizzati all'interno delle pagine).

In una pagina DATI sono presenti informazioni utili e informazioni di controllo:

- Informazioni utili: tuple della tabella
- Informazioni di controllo: dizionario, bit di parità, altre informazioni del file system o del metodo d'accesso specifico.

## Struttura della pagina dati



## Struttura del dizionario

- **Tuple di lunghezza fissa:** il dizionario non è necessario, si deve solo memorizzare la dimensione delle tuple e l'offset del punto iniziale
- **Tuple di lunghezza variabile:** il dizionario memorizza l'offset di ogni tupla presente nella pagina e di ogni attributo di ogni tupla

**Lunghezza massima di una tupla** = dimensione massima dell'area disponibile su una pagina, altrimenti va gestito il caso di tuple memorizzate su più pagine (in postgresSQL si veda la soluzione TOAST - The Oversized-Attribute Storage Technique).

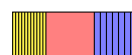
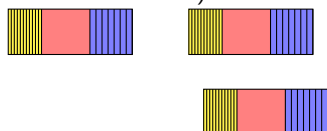
## Operazioni

- Inserimento di una tupla
  - Se esiste spazio contiguo sufficiente: inserimento semplice
  - Se non esiste spazio contiguo ma esiste spazio sufficiente: riorganizzare lo spazio ed eseguire un inserimento semplice
  - Se non esiste spazio sufficiente: operazione rifiutata
- Cancellazione: sempre possibile anche senza riorganizzare
- Accesso ad una tupla
- Accesso ad un attributo di una tupla
- Accesso sequenziale (di solito in ordine di chiave primaria)
- Riorganizzazione

## Rappresentazione di una tabella a livello fisico

### Livello fisico

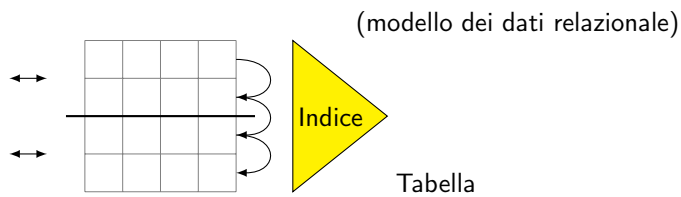
(memoria secondaria)



Struttura fisica:

- Strutture ad accesso calc.
- Seriale
- Array

- Sequenza ordinata o file **Livello logico** sequenziale



|  |  |  |  |
|--|--|--|--|
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

## Strutture fisiche di rappresentazione dei dati

### File sequenziale

Ha una struttura sequenziale ordinata in base alla chiave di ordinamento

Caratteristica fondamentale: è un file sequenziale dove le tuple sono ordinate secondo una chiave di ordinamento.

Esempio:

|          | Pagina | Filiale | Conto | Cliente | Saldo |
|----------|--------|---------|-------|---------|-------|
| Pagina 1 |        | A       | 102   | Rossi   | 1000  |
|          |        | B       | 110   | Rossi   | 3020  |
|          |        | B       | 198   | Bianchi | 500   |
| Pagina 2 |        | E       | 17    | Neri    | 345   |
|          |        | E       | 102   | Verdi   | 1200  |
|          |        | E       | 113   | Bianchi | 200   |
|          |        | H       | 53    | Neri    | 120   |
|          |        | F       | 78    | Verdi   | 3400  |

## Operazioni

- Inserimento di una tupla:
  - Individuare la pagina P che contiene la tupla che precede, nell'ordine della chiave, la tupla da inserire.
  - Inserire la tupla nuova in P; se l'operazione non va a buon fine aggiungere una nuova pagina (overflow page) alla struttura: la pagina contiene la nuova tupla, altrimenti si prosegue.
  - Aggiustare la catena di puntatori.
- Scansione sequenziale ordinata secondo la chiave (seguendo i puntatori)
- Cancellazione di una tupla
  - Individuare la pagina P che contiene la tupla da cancellare.
  - Cancellare la tupla da P.
  - Aggiustare la catena di puntatori.
- Riorganizzazione: si assegnano le tuple alle pagine in base ad opportuni coefficienti di riempimento, riaggiustando i puntatori.

Per aumentare le prestazioni degli accessi alle tuple memorizzate nelle strutture fisiche (FILE SEQUENZIALE), si introducono strutture ausiliarie (dette strutture di accesso ai dati o INDICI).

## Indici

Tali strutture velocizzano l'accesso casuale via chiave di ricerca. La chiave di ricerca è un insieme di attributi utilizzati dall'indice nella ricerca.

### Indici su file sequenziali

- **indice primario:** in questo caso la chiave di ordinamento del file sequenziale coincide con la chiave di ricerca dell'indice.
- **indice secondario:** in questo caso invece la chiave di ordinamento e la chiave di ricerca sono diverse.

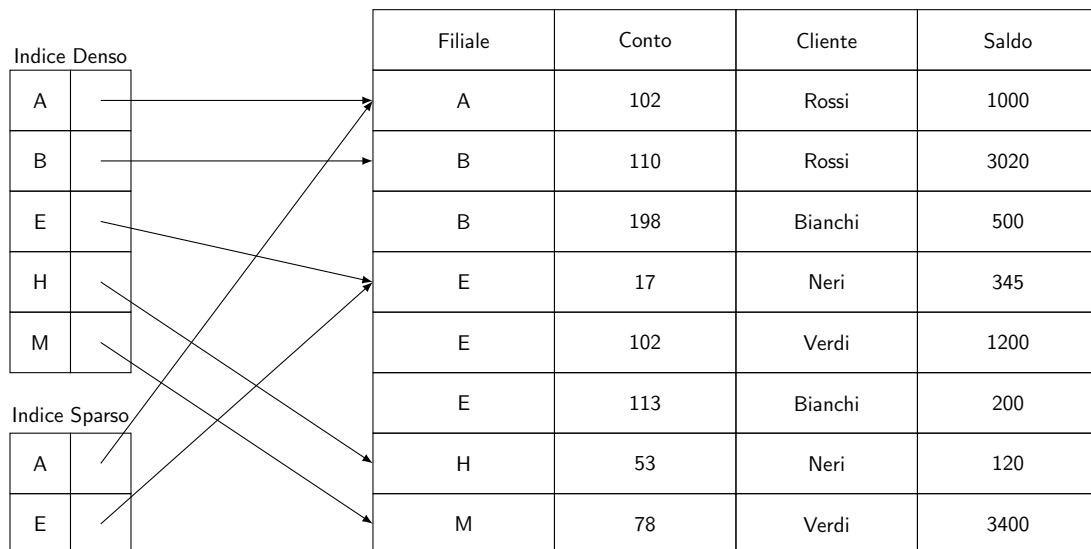
### Indice primario

Usa una chiave di ricerca che coincide con la chiave di ordinamento del file sequenziale. Ogni record dell'indice primario contiene una coppia  $\langle v_i, p_i \rangle$ :

- $v_i$ : valore della chiave di ricerca
- $p_i$ : puntatore al primo record nel file sequenziale con chiave  $v_i$

Esistono due varianti dell'indice primario:

- **indice denso:** per ogni occorrenza della chiave presente nel file esiste un corrispondente record nell'indice.
- **indice sparso:** solo per alcune occorrenze della chiave presenti nel file esiste un corrispondente record nell'indice, tipicamente una per pagina.



### Operazioni

Ricerca di una tupla con chiave di ricerca  $K$ :

- **denso** ( $K$  è presente nell'indice):
  - Scansione sequenziale dell'indice per trovare il record  $(K, p_k)$
  - Accesso al file attraverso il puntatore  $p_k$
  - Costo: 1 accesso indice + 1 accesso pagina dati
- **sparso** ( $K$  potrebbe non essere presente nell'indice)
  - Scansione sequenziale dell'indice fino al record  $(K', p_{k'})$  dove  $K'$  è il valore più grande che sia minore o uguale a  $K$ .
  - Accesso al file attraverso il puntatore  $p'_{k'}$  e scansione del file (pagina corrente) per trovare le tuple con chiave  $K$ .
  - Costo: 1 accesso indice + 1 accesso pagina dati

#### Inserimento di un record nell'indice

Come inserimento nel **file sequenziale** (nella pagina della memoria secondaria invece di tuple ci sono record dell'indice):

- **denso**: L'inserimento nell'indice avviene solo se la tupla inserita nel file ha un valore di chiave K che non è già presente.
- **sparso**: L'inserimento avviene solo quando, per effetto dell'inserimento di una nuova tupla, si aggiunge una pagina dati alla struttura; in tutti gli altri casi l'indice rimane invariato.

#### Cancellazione di un record nell'indice

Come cancellazione nel **file sequenziale**

- **denso**: La cancellazione nell'indice avviene solo se la tupla cancellata nel file è l'ultima tupla con valore di chiave K.
- **sparso**: La cancellazione nell'indice avviene solo quando K è presente nell'indice e la corrispondente pagina viene eliminata; altrimenti, se la pagina sopravvive, va sostituito K nel record dell'indice con il primo valore K' presente nella pagina.

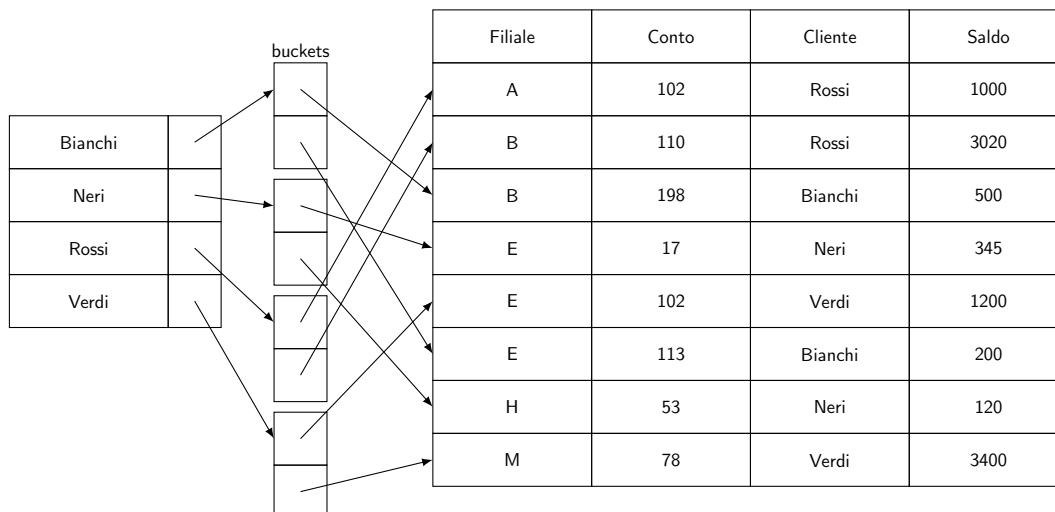
### Indice secondario

Usa una chiave di ricerca che NON coincide con la chiave di ordinamento del file sequenziale. Ogni record dell'indice secondario contiene una coppia  $\langle v_i, p_i \rangle$ :

- $v_i$ : valore della chiave di ricerca.
- $p_i$ : puntatore al **bucket di puntatori** che individuano nel file sequenziale tutte le tuple con valore di chiave  $v_i$ .

*Gli indici secondari sono sempre densi.*

### Esempio



### Operazioni

Ricerca di una tupla con chiave di ricerca K:

- Scansione sequenziale dell'indice per trovare il record  $(K, p_K)$
- Accesso al bucket B di puntatori attraverso il puntatore  $p_K$
- Accesso al file attraverso i puntatori del bucket B.

Costo: 1 accesso indice + 1 accesso al bucket + n accessi pagine dati

Inserimento e cancellazione: come indice primario denso con in più l'aggiornamento dei bucket.

### Osservazione

- Quando l'indice aumenta di dimensioni, non può risiedere sempre in memoria centrale: di conseguenza deve essere gestito in memoria secondaria.
- È possibile utilizzare un FILE SEQUENZIALE ORDINATO per rappresentare l'indice in memoria secondaria.
- Le prestazioni di accesso a tale struttura fisica a fronte di inserimenti/cancellazioni tendono a degradare e richiedono frequenti riorganizzazioni. Inoltre è disponibile solo l'accesso sequenziale.
- Per superare il problema si introducono per gli indici strutture fisiche diverse: le **strutture ad albero** e le **strutture ad accesso calcolato**.

## 11.5 B<sup>+</sup>-tree

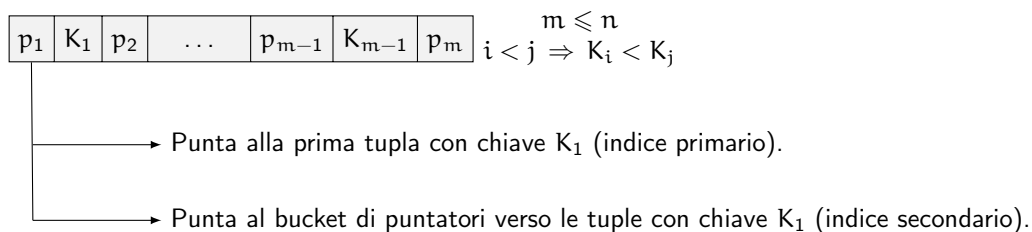
Caratteristiche generali dell'indice B<sup>+</sup>-tree:

- è una struttura ad albero
- ogni **nodo** corrisponde ad una **pagina della memoria secondaria**
- i legami tra nodi diventano puntatori a pagina
- ogni nodo ha un **numero elevato di figli**, quindi l'albero ha tipicamente pochi livelli e molti nodi foglia
- l'albero è **bilanciato**: la lunghezza dei percorsi che collegano la radice ai nodi foglia è costante
- inserimenti e cancellazioni non alterano le prestazioni dell'accesso ai dati: **l'albero si mantiene bilanciato**.

### Struttura di un B<sup>+</sup>-tree

**Nodo foglia:**

- può contenere fino a  $(n - 1)$  valori della chiave di ricerca e fino a  $n$  puntatori.



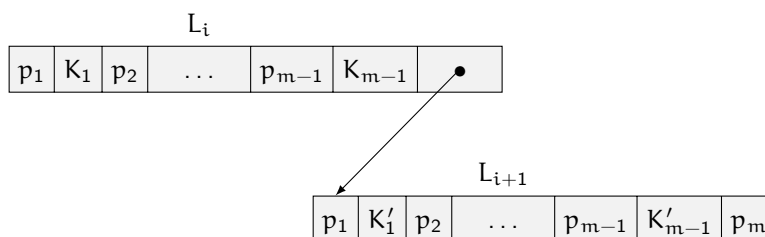
- variante: al posto dei valori chiave il nodo foglia contiene direttamente le tuple (struttura integrata dati/indice)

**Nodo foglia (vincolo di ordinamento):**

- I nodi foglia **sono ordinati**. Inoltre, dati due nodi foglia  $L_i$  e  $L_j$  con  $i < j$  risulta:

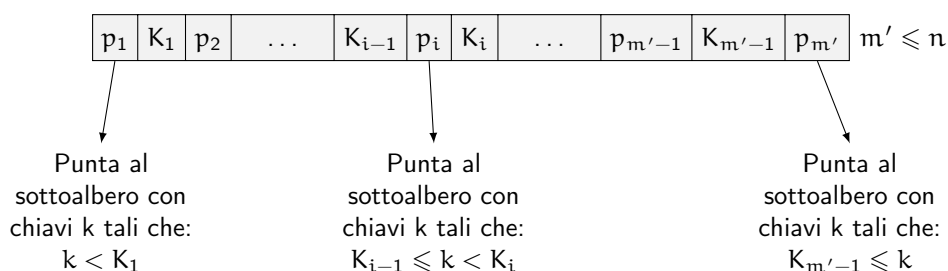
$$\forall K_t \in L_i : \forall K_s \in L_j : K_t < K_s$$

- Il puntatore  $p_m$  del nodo  $L_i$  punta al nodo  $L_{i+1}$  se esiste.



**Nodo intermedio**

Può contenere fino a  $n$  puntatori a nodo.



**Nodo foglia (vincolo di riempimento)**

Ogni nodo foglia contiene un numero di valori chiave  $\#chiavi$  vincolato come segue:

$$\lceil (n-1)/2 \rceil \leq \#chiavi \leq (n-1)$$

**Nodo Intermedio (vincolo di riempimento)**

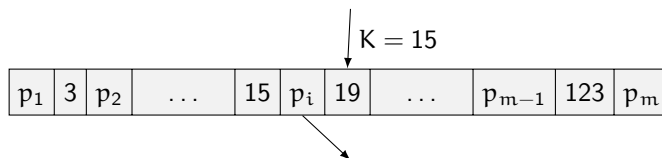
Ogni nodo intermedio contiene un numero di puntatori  $\#puntatori$  vincolato come segue (non vale per la radice):

$$\lceil n/2 \rceil \leq \#puntatori \leq n$$

**Ricerca dei record con chiave K**

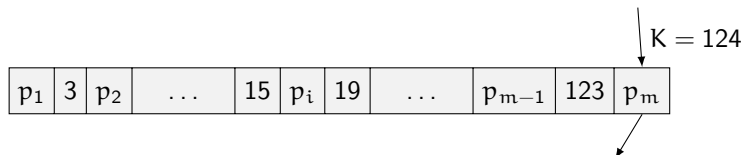
- Passo 1: Cercare nel nodo (radice) il più piccolo valore di chiave maggiore di K.

Se tale valore esiste (supponiamo sia  $K_i$ ) allora seguire il puntatore  $p_i$ .



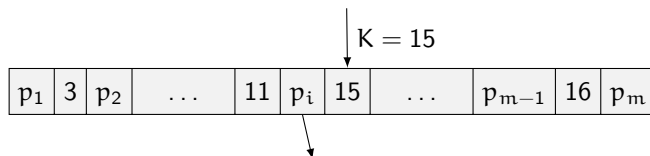
Punta al sottoalbero con chiavi  $k$  tali che:  $15 \leq k < 19$

Se tale valore non esiste seguire il puntatore  $p_m$ .



Punta al sottoalbero con chiavi  $k$  tali che:  $123 \leq k$

- Passo 2: Se il nodo raggiunto è un nodo foglia cercare il valore K nel nodo e seguire il corrispondente puntatore verso le tuple, altrimenti riprendere il passo 1.



Punta alle tuple con valore:  $k = 15$

**Osservazioni**

- Il costo di una ricerca nell'indice, in termini di numero di accessi alla memoria secondaria, risulta pari al numero di nodi acceduti nella ricerca.
- Tale numero in una struttura ad albero è pari alla profondità dell'albero, che nel B<sup>+</sup>-tree è indipendente dal percorso ed è funzione del fan-out  $n$  e del numero di valori chiave presenti nell'albero  $\#valoriChiave$ :

$$\text{prof}_{B^+-tree} \leq 1 + \log_{\lceil n/2 \rceil} \left( \frac{\#valoriChiave}{\lceil (n-1)/2 \rceil} \right)$$

**Dimostrazione:**

- Dato un certo numero di valori chiave da inserire ( $\#valoriChiave$ ) il numero massimo di nodi foglia è pari a:

$$NF_{\max} = \frac{\#valoriChiave}{\lceil (n-1)/2 \rceil}$$



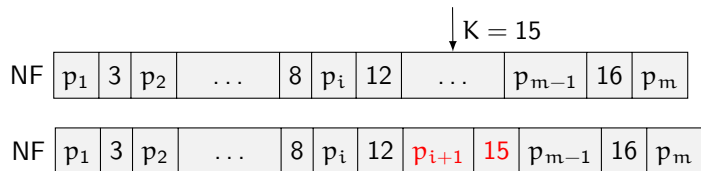
- Quindi partendo dal numero massimo di nodi foglia  $NF_{max}$  il numero massimo di livelli dell'albero, in presenza di nodi intermedi a riempimento minimo, risulta pari a:

$$NI_{max} = \log_{\lceil n/2 \rceil}(NF_{max})$$

- Contando il livello dei nodi foglia si ottiene la profondità massima pari a:  $1 + NI_{max}$

### Inserimento di un nuovo valore della chiave K

- Passo 1: ricerca del nodo foglia NF dove il valore K va inserito
  - Passo 2: se K è presente in NF, allora:
    - Indice primario: nessuna azione
    - Indice secondario: aggiornare il bucket di puntatori
- altrimenti, inserire K in NF rispettando l'ordine e:
- Indice primario: inserire puntatore alla tupla con valore K della chiave
  - Indice secondario: inserire un nuovo bucket di puntatori contenente il puntatore alla tupla con valore K della chiave.



se non è possibile inserire K in NF, allora eseguire uno **split** di NF.

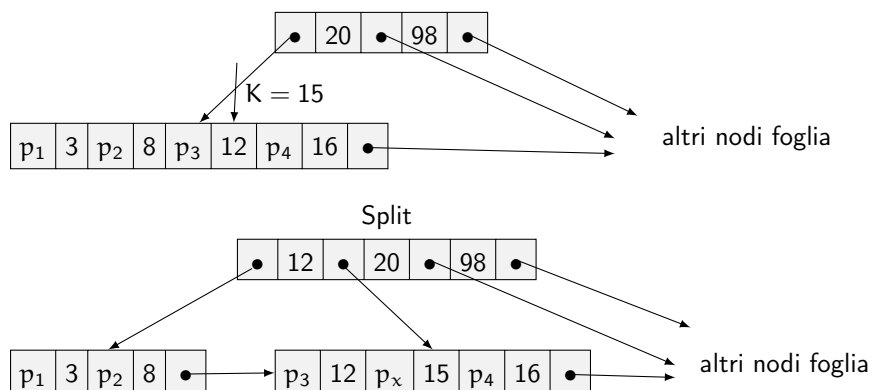
### Inserimento di un nuovo valore della chiave K

#### Split di un nodo foglia

Nel nodo da dividere esistono n valori chiave, si procede come segue:

- Creare due nodi foglia
- Inserire i primi  $\lceil (n-1)/2 \rceil$  valori nel primo
- Inserire i rimanenti nel secondo
- Inserire nel nodo padre un nuovo puntatore per il secondo nodo foglia generato e riaggiustare i valori chiave presenti nel nodo padre.
- Se anche il nodo padre è pieno (n puntatori già presenti) lo split si propaga al padre e così via, se necessario, fino alla radice.

#### Esempio di split di un nodo foglia (fan-out = 5)

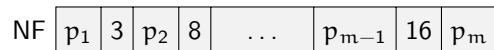
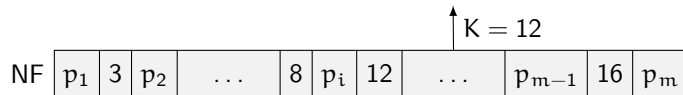


### Cancellazione di un valore della chiave K

- Passo1: ricerca del nodo foglia NF dove il valore K va cancellato
- Passo 2: cancellare K da NF insieme al suo puntatore:

Indice primario: nessuna ulteriore azione

Indice secondario: liberare il bucket di puntatori



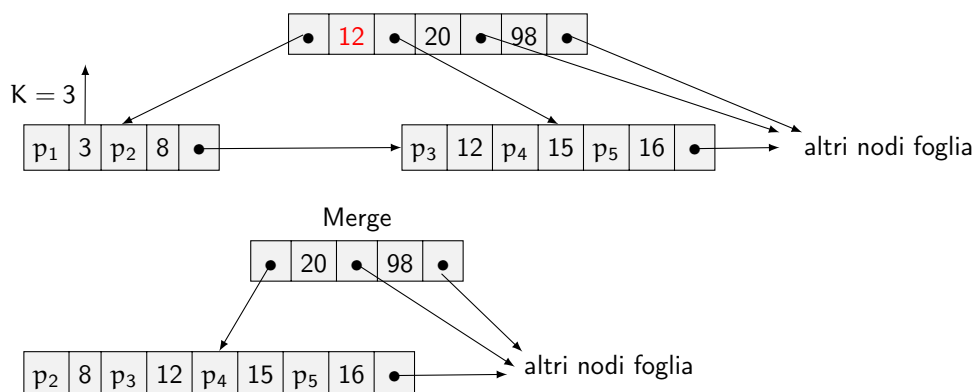
se dopo la cancellazione di K da NF viene violato il vincolo di riempimento minimo di NF, allora eseguire un Merge di NF.

#### Merge di un nodo foglia:

Nel nodo da unire esistono  $\lceil (n-1)/2 \rceil - 1$  valori chiave, si procede come segue:

- Individuare il nodo fratello adiacente da unire al nodo corrente
- Se i due nodi hanno complessivamente al massimo n-1 valori chiave, allora
  - si genera un unico nodo contenente tutti i valori
  - si toglie un puntatore dal nodo padre
  - si aggiustano i valori chiave del nodo padre
- Altrimenti si distribuiscono i valori chiave tra i due nodi e si aggiustano i valori chiave del nodo padre
- Se anche il nodo padre viola il vincolo minimo di riempimento (meno di  $n/2$  puntatori presenti), il Merge si propaga al padre e così via, se necessario, fino alla radice.

#### Esempio di Merge di un nodo foglia ( $f_{in} - out = 5$ ):



## 11.6 Strutture ad accesso calcolato

Caratteristiche:

1. si basano su una funzione di hash che mappa le chiavi sugli indirizzi di memorizzazione delle tuple nelle pagine dati della memoria secondaria

$$h : K \rightarrow B$$

K: dominio delle chiavi, B: dominio degli indirizzi

2. Uso pratico di una funzione di hash

Si stima il numero di valori chiave che saranno contenuti nella tabella

Si alloca un numero di bucket di puntatori (B) uguale al numero stimato

Si definisce una funzione di FOLDING che trasforma i valori chiave in numeri interi positivi:

$$f : K \rightarrow Z^+$$

Si definisce una funzione di HASHING:

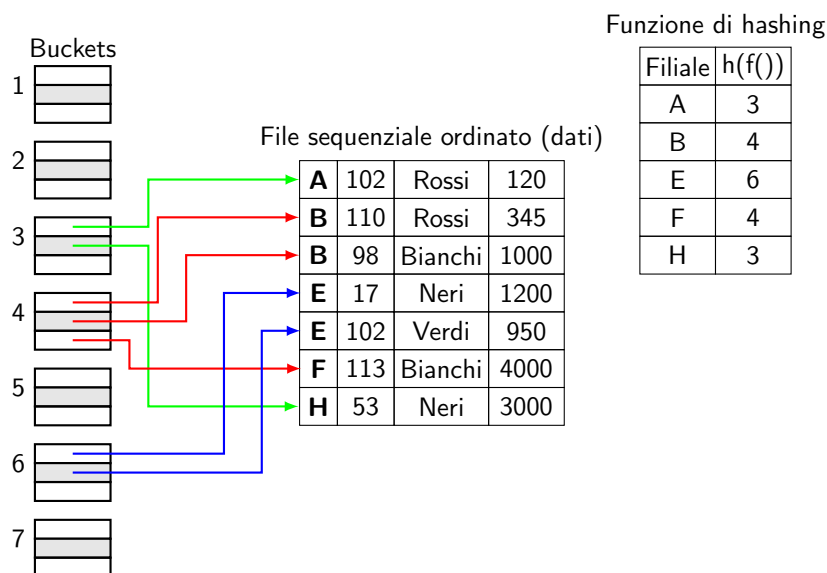
$$h : Z^+ \rightarrow B$$

### Caratteristica di una buona funzione di HASHING:

Distribuire in modo UNIFORME e CASUALE i valori chiave nei bucket.

**Nota:** È pesante cambiare la funzione di hashing dopo che la struttura d'accesso è stata riempita.

### Esempio:



### Operazioni

**Ricerca:** Dato un valore di chiave K trovare la corrispondente tupla

- Calcolare  $b = h(f(K))$  (costo zero)
- Accedere al bucket b (costo: 1 accesso a pagina)
- Accedere alle n tuple attraverso i puntatori del bucket (costo: m accessi a pagina con  $m \leq n$ )

**Inserimento e Cancellazione:** Di complessità simile alla ricerca.

### Osservazione

- La struttura ad accesso calcolato funziona se i buckets conservano un basso coefficiente di riempimento. Infatti il problema delle strutture ad accesso calcolato è la gestione delle **collisioni**.
- Collisione**: si verifica quando, dati due valori di chiave  $K_1$  e  $K_2$  con  $K_1 \neq K_2$ , risulta:

$$h(f(K_1)) = h(f(K_2))$$

Un numero eccessivo di collisioni porta alla saturazione del bucket corrispondente.

Probabilità che un bucket riceva  $t$  chiavi su  $n$  inserimenti:

$$p(t) = \binom{n}{t} \left(\frac{1}{B}\right)^t \left(1 - \frac{1}{B}\right)^{n-t}$$

dove  $B$  è il numero totale di buckets.

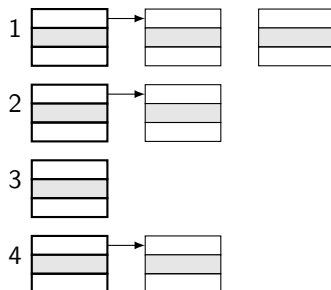
Probabilità di avere più di  $F$  collisioni ( $F$ : numero di puntatori nel bucket):

$$p_K = 1 - \sum_{i=0}^F p(i)$$

### Gestione delle collisioni

Un numero eccessivo di collisioni sullo stesso indirizzo porta alla saturazione del bucket corrispondente. Per gestire tale situazione si prevede la possibilità di allocare Bucket di overflow, collegati al Bucket di base.

**Nota:** Le prestazioni della ricerca peggiorano in quanto individuato il bucket di base, potrebbe essere poi necessario accedere ai buckets di overflow.



### Ricerca

- Selezioni basate su condizioni di uguaglianza  $A = \text{cost}$ 
  - Hashing (senza overflow buckets): tempo costante
  - $B^+$ -tree: tempo logaritmico nel numero di chiavi
- Selezioni basate su intervalli (range)  $A > \text{cost}_1 \text{ AND } A < \text{cost}_2$ 
  - Hashing: numero elevato di selezioni su condizioni di uguaglianza per scandire tutti i valori del range
  - $B^+$ -tree: tempo logaritmico per accedere al primo valore dell'intervallo, scansione sui nodi foglia fino all'ultimo valore compreso nel range.

## 12 Esecuzione concorrente di Transizioni

### Osservazione

- Per gestire con prestazione accettabili il carico di lavoro tipico delle applicazioni gestionali (100 o 1000 tps) un DBMS deve eseguire le transazioni in modo concorrente.
- L'esecuzione concorrente di transazioni senza controllo può generare anomalie (come vedremo in alcuni esempi).
- È quindi necessario introdurre dei meccanismi di controllo nell'esecuzione delle transazioni per evitare tali anomalie.

### 12.1 Anomalie di esecuzione concorrente

Anomalie tipiche:

- **Lettura sporca**
- **Lettura inconsistente**
- **Perdita di update**
- **Aggiornamento fantasma**

Notazione:

- Indichiamo con  $t_i$  una transazione
- $r_i(x)$ : è una operazione di lettura eseguita dalla transazione  $t_i$  sulla risorsa  $x$ .
- $w_i(x)$ : è una operazione di scrittura eseguita dalla transazione  $t_i$  sulla risorsa  $x$ .

### Esempi

#### Anomalia di Aggiornamento

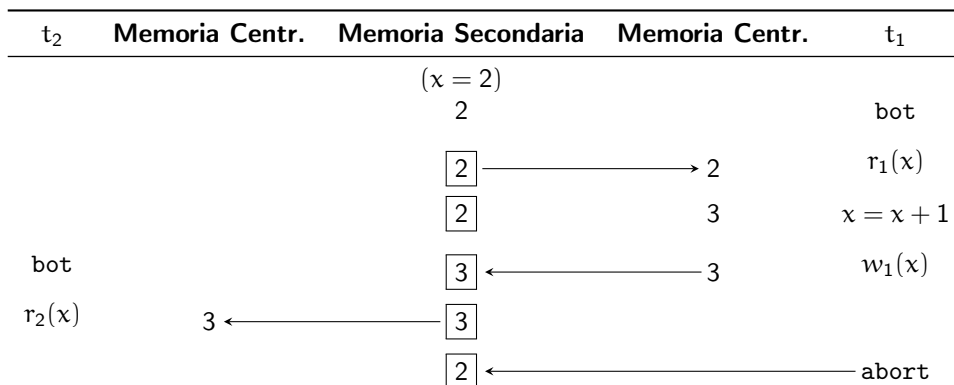
$t_1$  : bot  $r_1(x) \ x = x + 1$  ;  $w_1(x)$  commit eot

$t_2$  : bot  $r_2(x) \ x = x + 1$  ;  $w_2(x)$  commit eot

Osserviamo cosa accade e come le due transazioni eseguono le operazioni

| $t_1$       | Memoria Centr. | Memoria Secondaria  | Memoria Centr. | $t_2$       |
|-------------|----------------|---|----------------|-------------|
| bot         |                | ( $x = 2$ )   |                |             |
| $r_1(x)$    | 2              | <span style="border: 1px solid black; padding: 2px;">2</span>   |                |             |
| $x = x + 1$ | 3              | <span style="border: 1px solid black; padding: 2px;">2</span>   |                | bot         |
|             | 3              | <span style="border: 1px solid black; padding: 2px;">2</span> → | 2              | $r_2(x)$    |
|             | 3              | <span style="border: 1px solid black; padding: 2px;">2</span>   | 3              | $x = x + 1$ |
|             | 3              | <span style="border: 1px solid black; padding: 2px;">3</span> ← | 3              | $w_2(x)$    |
|             | 3              | 3   | 3              | commit      |
| $w_1(x)$    | 3 →            | <span style="border: 1px solid black; padding: 2px;">3</span>   |                | eot         |
| commit      |                | 3   |                |             |
| eot         |                |   |                |             |

$$t_1 : \text{bot } r_1(x) \ x = x + 1; \ w_1(x) \ \dots \text{abort}$$

$$t_2 : \text{bot } r_2(x)$$


## 12.2 Schedule

Sequenza di operazioni di lettura e scrittura eseguite su risorse della base di dati da diverse transazioni concorrenti

Esso rappresenta una possibile esecuzione concorrente di diverse transazioni.

Quali schedule accettare per evitare le anomalie?

Si stabilisce di accettare SOLO gli schedule "equivalenti" ad uno schedule seriale.

## 12.3 Schedule seriale

È uno schedule dove le operazioni di ogni transazione compaiono in sequenza senza essere inframmezzate da operazioni di altre transazioni.

$s_1 : r_1(x) \ r_2(x) \ w_1(x) \ w_2(x)$  NON è SERIALE

$s_2 : r_1(x) \ w_1(x) \ r_2(x) \ w_2(x)$  è SERIALE

## 12.4 Schedule serializzabile

È uno schedule "equivalente" ad uno schedule seriale.

A questo punto occorre definire il concetto di EQUIVALENZA tra schedule.

L'idea è che si vogliono considerare equivalenti due schedule che producono gli stessi effetti sulla base di dati. Quindi accettare schedule serializzabili significa accettare schedule che hanno gli stessi effetti di uno schedule seriale.

## Ipotesi di commit-proiezione

Si suppone che le transazioni abbiano esito noto. Quindi si tolgono dagli schedule tutte le operazioni delle transazioni che non vanno a buon fine.

Richiedono questa ipotesi le seguenti tecniche di gestione della concorrenza:

- Gestione della concorrenza basata sulla view-equivalenza
- Gestione della concorrenza basata sulla conflict-equivalenza

### Relazione "LEGGE\_DA"

Dato uno schedule  $S$  si dice che un'operazione di lettura  $r_i(x)$ , che compare in  $S$ , LEGGE\_DA un'operazione di scrittura  $w_j(x)$ , che compare in  $S$ , se  $w_j(x)$  precede  $r_i(x)$  in  $S$  e non vi è alcuna operazione  $w_k(x)$  tra le due.

Esempio:

$$\begin{aligned} S_1 : r_1(x) r_2(x) w_2(x) w_1(x) & \quad \text{LEGGE\_DA}(S_1) = \emptyset \\ S_2 : r_1(x) w_1(x) r_2(x) w_2(x) & \quad \text{LEGGE\_DA}(S_2) = \{(r_2(x), w_1(x))\} \end{aligned}$$

### Scritture finali

Dato uno schedule  $S$  si dice che un'operazione di scrittura  $w_i(x)$ , che compare in  $S$ , è una SCRITTURA FINALE se è l'ultima operazione di **scrittura della risorsa  $x$**  in  $S$ .

Esempio:

$$\begin{aligned} S_1 : r_1(x) r_2(x) w_2(x) w_3(y) w_1(x) & \quad \text{SCRITTURE\_FINALI}(S_1) = w_3(y), w_1(x) \\ S_2 : r_1(x) w_1(x) r_2(x) w_2(x) w_3(y) & \quad \text{SCRITTURE\_FINALI}(S_2) = w_3(y), w_2(x) \end{aligned}$$

### View-equivalenza

Due schedule  $S_1$  e  $S_2$  sono view-equivalenti ( $S_1 \approx_V S_2$ ) se possiedono le stesse relazioni LEGGE\_DA e le stesse SCRITTURE FINALI.

### View-serializzabilità

Uno schedule  $S$  è view-serializzabile (VSR) se esiste uno schedule seriale  $S'$  tale che  $S' \approx_V S$ .

### Osservazioni

- L'algoritmo per il test di **view-equivalenza** tra due schedule è di **complessità lineare**.
- L'algoritmo per il test di **view-serializzabilità** di uno schedule è di **complessità esponenziale**.

In conclusione:

la VSR richiede algoritmi di complessità troppo elevata, richiede l'ipotesi di commit-proiezione  $\Rightarrow$  non è applicabile nei sistemi reali.

## 12.5 Conflitto

Dato uno schedule  $S$  si dice che una coppia di operazioni  $(a_i, a_j)$ , dove  $a_i$  e  $a_j$  compaiono nello schedule  $S$ , rappresentano un conflitto se:

- $i \neq j$  (transazioni diverse)
- entrambe operano sulla stessa risorsa
- almeno una di esse è una operazione di scrittura
- $a_i$  compare in  $S$  prima di  $a_j$

### Conflict-equivalenza

Due schedule  $S_1$  e  $S_2$  sono conflict-equivalenti ( $S_1 \approx_C S_2$ ) se possiedono le stesse operazioni e gli stessi conflitti.

### Conflict-serializzabilità

Uno schedule  $S$  è conflict-serializzabile (CSR) se esiste uno schedule seriale  $S'$  tale che  $S' \approx_C S$ .

## Osservazioni

- L'algoritmo per il test di conflict-serializzabilità di uno schedule è di complessità lineare.
- Algoritmo (dato uno schedule  $S$ ):
  - Si costruisce il grafo  $G(N,A)$  dove  $N = \{t_1, \dots, t_n\}$  con  $t_1, \dots, t_n$  transazioni di  $S$ ;  $(t_i, t_j) \in A$  se esiste almeno un conflitto  $(a_i, a_j)$  in  $S$ .
  - Se il grafo così costruito è ACICLICO allora  $S$  è conflict- serializzabile.

## Verifica algoritmo sul grafo

### Se $S$ è CSR $\Rightarrow$ il suo grafo è ACICLICO.

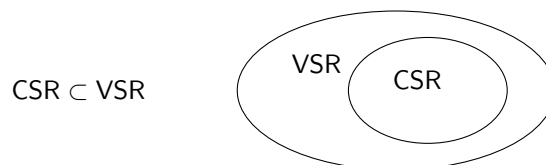
Se uno schedule  $S$  è CSR allora è  $\approx_C$  ad uno schedule seriale  $S_{ser}$ . Supponiamo che le transazioni  $S_{ser}$  in siano ordinate secondo il  $T_{ID} : t_1, t_2, \dots, t_n$ .  $S_{ser}$  ha tutti i conflitti di  $S$  esattamente nello stesso ordine, poiché  $S$  è  $\approx_C$  a  $S_{ser}$ ; ora nel grafo di  $S_{ser}$  poiché è seriale ci possono essere solo archi  $(i, j)$  con  $i < j$  e quindi il grafo non può avere cicli, perché un ciclo richiede almeno un arco  $(i, j)$  con  $i > j$ . Poiché  $S$  è  $\approx_C$  a  $S_{ser}$  ha come già detto gli stessi conflitti di quest'ultimo e quindi lo stesso grafo che risulterà anch'esso ACICLICO.

### Se il grafo di $S$ è ACICLICO $\Rightarrow S$ è CSR

Se il grafo di  $S$  è aciclico, allora esiste fra i nodi un "ordinamento topologico", vale a dire una numerazione dei nodi tale che il grafo contiene solo archi  $(i, j)$  con  $i < j$ . Per dimostrare che  $S$  è CSR occorre generare a partire da questa osservazione uno schedule seriale che abbia gli stessi conflitti di  $S$ . Lo schedule seriale le cui transazioni sono ordinate secondo l'ordinamento topologico è CONFLICT-equivalente a  $S$ , in quanto ha lo stesso grafo di  $S$  e quindi gli stessi conflitti di  $S$ . Quindi l'ACICLICITA' del grafo di  $S$  assicura la possibilità di ottenere uno schedule seriale  $\approx_C$  a  $S$  e quindi assicura che  $S$  sia CSR.

## Osservazioni

La conflict-serializzabilità è condizione sufficiente ma non necessaria per la view-serializzabilità.





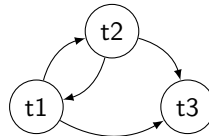
Controesempio per la non necessità:

$S : r_1(x)w_2(x)w_1(x)w_3(x) \quad \text{LEGGE\_DA} = \emptyset \quad \text{SCR\_FIN} = \{w_3(x)\}$

È view-serializzabile: in quanto view-equivalente allo schedule seriale

$S_1 : r_1(x)w_1(x)w_2(x)w_3(x) \quad \text{LEGGE\_DA} = \emptyset \quad \text{SCR\_FIN} = \{w_3(x)\}$

**Non conflict-serializzabile: in quanto il grafo dei conflitti è ciclico.**



CSR  $\Rightarrow$  VSR

Supponiamo  $S_1 \approx_C S_2$  e dimostriamo che  $S_1 \approx_V S_2$ : Osserviamo che poiché  $S_1 \approx_C S_2$  i due schedule hanno:

- **stesse scritture finali**: se così non fosse, ci sarebbero almeno due scritture sulla stessa risorsa in ordine diverso e poiché due scritture sono in conflitto i due schedule non sarebbero  $\approx_C$ .
- **stessa relazione "legge\_da"**: se così non fosse, ci sarebbero scritture in ordine diverso o coppie lettura-scrittura in ordine diverso e quindi, come sopra sarebbe violata la  $\approx_C$ .

Le tecniche per il controllo della concorrenza che non richiedono di conoscere l'esito delle transazioni sono:

- Timestamp con scritture bufferizzate (TS buffer)
- Locking a due fasi stretto (2PL stretto)

## 12.6 Locking a due fasi

È il metodo applicato nei sistemi commerciali per la gestione dell'esecuzione concorrente di transazioni.

Tre sono gli aspetti che caratterizzano il locking a due fasi:

- Il **meccanismo** di base per la gestione dei **lock**.
- La **politica di concessione** dei lock sulle risorse
- La regola che garantisce la **serializzabilità**

### Meccanismo di base

Si basa sull'introduzione di alcune **primitive di lock** che consentono alle transazioni di bloccare (lock) le risorse sulle quali vogliono agire con operazioni di lettura e scrittura.

**Primitive di lock:**

- $r\_lock_K(x)$ : richiesta di un lock *condiviso* da parte della transazione  $t_K$  sulla risorsa  $x$  per eseguire una lettura.
- $w\_lock_K(x)$ : richiesta di un lock *esclusivo* da parte della transazione  $t_K$  sulla risorsa  $x$  per eseguire una scrittura.
- $unlock_K(x)$ : richiesta da parte della transazione  $t_K$  di liberare la risorsa  $x$  da un precedente lock.

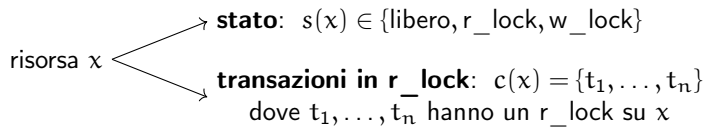
**Regole** per l'uso delle primitive da parte delle transazioni.

- **R1**: ogni lettura deve essere preceduta da un  $r\_lock$  e seguita da un  $unlock$ . Sono ammessi più  $r\_lock$  contemporanei sulla stessa risorsa (lock condiviso).
- **R2**: ogni scrittura deve essere preceduta da un  $w\_lock$  e seguita da un  $unlock$ . Non sono ammessi più  $w\_lock$  (oppure  $w\_lock$  e  $r\_lock$ ) contemporanei sulla stessa risorsa (lock esclusivo).

Se una transazione segue le regole R1 e R2 si dice BEN FORMATA rispetto al locking.

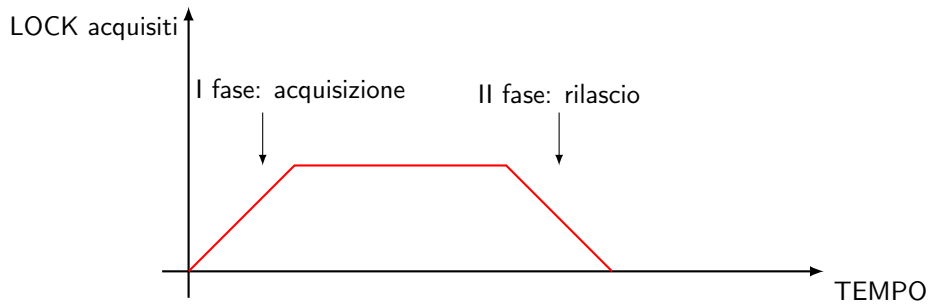
## Politica di concessione dei lock

Il Gestore dei LOCK (o Gestore della concorrenza) mantiene per ogni risorsa le seguenti informazioni:



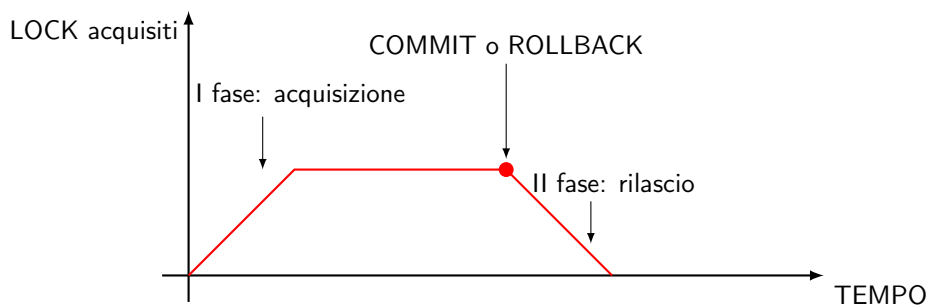
**Regola che garantisce la serializzabilità** (da cui prende il nome il metodo 2PL):

*Una transazione dopo aver rilasciato un LOCK non può acquisirne altri.*



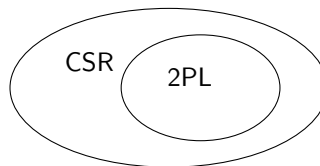
**Condizione aggiuntiva (2PL stretto)**

Una transazione può rilasciare i LOCK solo quando ha eseguito correttamente un COMMIT o un ROLLBACK.



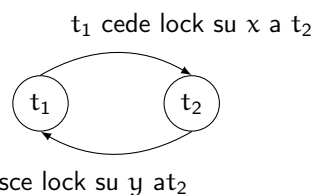
**Relazione tra 2PL e CSR**

$$2PL \subset CSR$$



$$s \in 2PL \Rightarrow s \in CSR$$

Per assurdo:  $s \in 2PL$  e  $s \notin CSR$ . Se  $s \notin CSR$  allora esiste una coppia (catena) di conflitti ciclici.



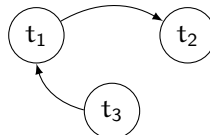
$$\begin{array}{ccccccc} w_1(x) & r_2(x) & w_2(x) & w_2(y) & r_1(y) & & \\ \uparrow & \uparrow & & \uparrow & \uparrow & & \\ t_1 \text{ rilascia lock su } x & & t_1 \text{ acquisisce lock su } y & & & & \end{array} \quad \Bigg| \quad \text{Quindi } s \text{ non è 2PL}$$

$$\exists s \in \text{CSR} \quad \wedge \quad s \notin \text{2PL}$$

$$s : r_1(x) \, w_1(x) \, r_2(x) \, w_2(x) \, r_3(y) \, w_1(y)$$

$\xrightarrow{\quad t_1 \text{ rilascia lock su } x \quad} \quad \xrightarrow{\quad t_1 \text{ acquisisce lock su } y \quad}$

$$\text{Conflitti}(s) = \{ (r_1(x), w_2(x)), (w_1(x), r_2(x)), \\ (w_1(x), w_2(x)), (r_3(y), w_1(y)) \}$$



### Esempio di perdita di update con 2PL

Esempio su risorsa  $x = 2$

| t <sub>1</sub>          | Mem. Centr. | Database |      |     | Mem. Centr. | t <sub>2</sub>          |
|-------------------------|-------------|----------|------|-----|-------------|-------------------------|
|                         |             | c(x)     | s(x) | Val |             |                         |
| bot                     |             | ∅        | L    | 2   |             |                         |
| r_lock <sub>1</sub> (x) |             | {1}      | RL   | 2   |             |                         |
| r <sub>1</sub> (x)      | 2           | {1}      | RL   | 2   |             |                         |
| x = x + 1               | 3           | {1}      | RL   | 2   |             |                         |
|                         | 3           | {1}      | RL   | 2   |             |                         |
|                         | 3           | {1, 2}   | RL   | 2   |             | bot                     |
|                         | 3           | {1, 2}   | RL   | 2   | 2           | r_lock <sub>2</sub> (x) |
|                         | 3           | {1, 2}   | RL   | 2   | 3           | r <sub>2</sub> (x)      |
|                         | 3           | {1, 2}   | RL   | 2   | 3           | x = x + 1               |
|                         | 3           | {1, 2}   | RL   | 2   | 3           | w_lock <sub>2</sub> (x) |
| w_lock <sub>1</sub> (x) | 3           | {1, 2}   | RL   | 2   | 3           | → ATTESA                |
| → ATTESA                |             |          |      |     |             |                         |

#### BLOCCO CRITICO

È una situazione di blocco che si verifica nell'esecuzione di transazioni concorrenti quando:

- Due transazioni hanno bloccato delle risorse: t<sub>1</sub> ha bloccato r<sub>1</sub> e t<sub>2</sub> ha bloccato r<sub>2</sub>
- Inoltre t<sub>1</sub> è in attesa sulla risorsa r<sub>2</sub> e
- t<sub>2</sub> è in attesa sulla risorsa r<sub>1</sub>

Quanto spesso si verifica?

Se il numero medio di tuple per tabella è  $n$  e la granularità del lock è la tupla, la probabilità che si verifichi un lock tra due transazioni è pari a:

$$P(\text{deadlock di lunghezza } 2) = 1/n_2$$

Come risolvere il blocco critico:

- **Timeout:** una transazione in attesa su una risorsa trascorso il timeout viene abortita

- **Prevenzione :**

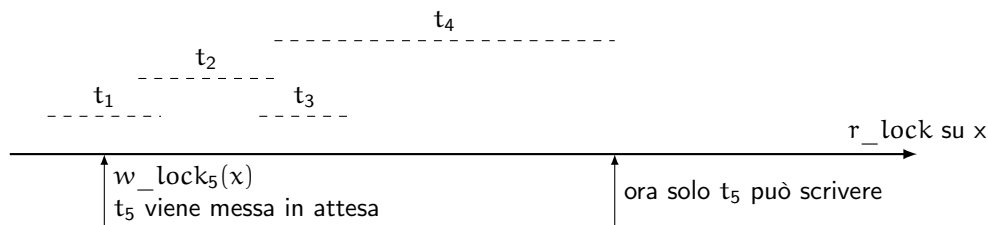
- Una transazione blocca tutte le risorse a cui deve accedere in lettura o scrittura in una sola volta (o blocca tutto o non blocca nulla)
- Ogni transazione  $t_i$  acquisisce un timestamp  $TS_i$  all'inizio della sua esecuzione. La transazione  $t_i$  può attendere una risorsa bloccata da  $t_j$  solo se vale una certa condizione sui TS ( $TS_i < TS_j$ ) altrimenti viene abortita e fatta ripartire con lo stesso TS.

- **Rilevamento** si esegue un'analisi periodica della tabella di LOCK per rilevare la presenza di un blocco critico (corrisponde ad un ciclo nel grafo delle relazioni di attesa). Questo sistema è quello più frequentemente adottato dai sistemi reali.

## 12.7 Blocco di una transazione (starvation)

La scelta di gestire i lock in lettura come lock condivisi produce un ulteriore problema che tuttavia nel contesto delle DBMS risulta poco probabile.

Infatti se una risorsa  $x$  viene costantemente bloccata da una sequenza di transazioni che acquisiscono  $r\_lock$  su  $x$ , un'eventuale transazione in attesa di scrivere su  $x$  viene bloccata per un lungo periodo fino alla fine della sequenza di letture.



Anche se poco probabile tale evenienza può essere scongiurata con tecniche simili a quanto visto per il blocco critico

In particolare è possibile analizzare la tabella delle relazioni di attesa e verificare da quanto tempo le transazioni stanno attendendo la risorsa e di conseguenza sospendere temporaneamente la concessione di lock in lettura condivisi per consentire la scrittura da parte della transazione in attesa.

## 12.8 Gestione della concorrenza in SQL

Poiché risulta molto oneroso per il sistema gestire l'esecuzione concorrente di transazioni con una conseguente diminuzione delle prestazioni del sistema, SQL prevede la possibilità di rinunciare in tutto o in parte al controllo di concorrenza per aumentare le prestazioni del sistema.

Ciò significa che è possibile a livello di singola transazione decidere di tollerare alcune anomalie di esecuzione concorrente.

| Livello di isolamento | Perdita di update | Lettura sporca | Lettura inconsistente | Update fantasma | Inserimento fantasma |
|-----------------------|-------------------|----------------|-----------------------|-----------------|----------------------|
| serializable          | X                 | X              | X                     | X               | X                    |
| repeatable read       | X                 | X              | X                     | X               |                      |
| read committed        | X                 | X              |                       |                 |                      |
| read uncommitted      | X                 |                |                       |                 |                      |

### Note:

- Tutti i livelli richiedono il 2PL stretto per le scritture.
- **serializable**: lo richiede anche per le letture e applica il lock di predicato per evitare l'inserimento fantasma.
- **repeatable read**: applica il 2PL stretto per tutti i lock in lettura applicati a livello di tupla e non di tabella. Consente inserimenti e quindi non evita l'anomalia di inserimento fantasma.

- `read committed`: richiede lock condivisi per le letture ma non il 2PL stretto.
- `read uncommitted`: non applica lock in lettura.

## 13 Ottimizzazione di interrogazioni

- Ogni interrogazione sottomessa al DBMS viene espressa in un linguaggio dichiarativo (ad esempio SQL)
- È quindi necessario trovare un equivalente espressione in linguaggio procedurale (ad esempio in algebra relazionale) per generare un piano di esecuzione.
- L'espressione algebrica va ottimizzata rispetto alle caratteristiche del DBMS a livello fisico (metodi d'accesso disponibili) e della base di dati corrente (statistiche del dizionario dei dati)

### "Compilazione" di un interrogazione

- Analisi lessicale e sintattica
- Ottimizzazione algebrica (indipendente dal modello di costo)
- Ottimizzazione basata sui costi di esecuzione

### 13.1 Ottimizzazione algebrica

L'ottimizzazione algebrica si basa fundamentalmente sulle regole di ottimizzazione già note dell'algebra relazionale:

- Anticipo delle selezioni (selection push)
- Anticipo delle proiezioni (projection push)

### 13.2 Metodi di accesso disponibili

- Scansione (scan) delle tuple di una relazione
- Ordinamenti
- Accesso diretto attraverso indice
- Diverse implementazioni del Join

### Scansione

Varianti:

- SCAN + PROIEZIONE SENZA ELIMINAZIONE DI DUPLICATI
- SCAN + SELEZIONE IN BASE AD UN PREDICATO
- SCAN + INSERIMENTO/CANCELLAZIONE/MODIFICA

Costo di una scansione sulla relazione R:  $NP(R)$

$NP(R)$  = Numero Pagine dati della relazione R.

## Ordinamento

Tali metodi sono utili per:

- ordinare il risultato di un'interrogazione (clausola order by),
- eliminare duplicati (select distinct),
- raggruppare tuple (group by);

Ordinamento su memoria secondaria: "Z-way Sort-Merge"

FASI:

- **Sort interno:** si leggono una alla volta le pagine della tabella; le tuple di ogni pagina vengono ordinate facendo uso di un algoritmo di sort interno (es. Quicksort); ogni pagina così ordinata, detta anche "run", viene scritta su memoria secondaria in un file temporaneo
- **Merge:** applicando uno o più passi di fusione, le "run" vengono unite, fino a produrre un'unica "run"

### Z-way Sort-Merge

Supponiamo di dover ordinare un input che consiste di una tabella di NP pagine e di avere a disposizione solo NB buffer in memoria centrale, con  $NB < NP$ . Per semplicità consideriamo il caso base a due vie ( $Z = 2$ ), e supponiamo di avere a disposizione solo 3 buffer in memoria centrale ( $NB = 3$ ).

...

Dopo la fase di "sort interno", nel caso base  $Z = 2$  si fondono 2 run alla volta:

- Con  $NB = 3$ , si associa un buffer a ognuna delle *run*, il terzo buffer serve per produrre l'output, 1 pagina alla volta.
- Si legge la prima pagina da ciascuna *run* e si genera la prima pagina dell'output; quando tutti i record di una pagina di *run* sono stati consumati, si legge un'altra pagina della *run*.

...

**Costo:** Consideriamo come costo solo il numero accessi a memoria sec. Nel caso base  $Z = 2$  e con  $NB = 3$  si può osservare che:

- Nella fase di sort interno si leggono e si riscrivono NP pagine
- Ad ogni passo di merge si leggono e si riscrivono NP pagine

Il numero di passi fusione è pari a:

$$\lceil \log_2(NP) \rceil$$

in quanto ad ogni passo il numero di *run* si dimezza ( $Z = 2$ ).

Il costo complessivo è pertanto pari a:

$$2 \times NP \times (\lceil \log_2 NP \rceil + 1)$$

## Accesso diretto via indice

Interrogazioni che fanno uso dell'indice:

- Selezioni con condizione atomica di uguaglianza ( $A = v$ ):  
richiede indice hash o B<sup>+</sup>-tree.
- Selezioni con condizione di range ( $A \geq v1 \text{ AND } A \leq v2$ ):  
richiede indice B<sup>+</sup>-tree.

- Selezioni con condizione costituita da una congiunzione di condizioni di uguaglianza ( $A = v1$  AND  $B = v2$ ):  
in questo caso si sceglie per quale delle condizioni di uguaglianza utilizzare l'indice; la scelta ricade sulla condizione più selettiva. L'altra si verifica direttamente sulle pagine dati.
- Selezioni con condizione costituita da una disgiunzione di condizioni di uguaglianza ( $A = v1$  OR  $B = v2$ ):  
in questo caso è possibile utilizzare più indici in parallelo, facendo un merge dei risultati eliminando i duplicati oppure, se manca anche solo uno degli indici, è necessario eseguire una scansione sequenziale.

## Algoritmi per il join

Le implementazioni più diffuse si riconducono ai seguenti operatori fisici:

- Nested Loop Join
- Merge Scan Join
- Hash-based Join

Si noti che, benché logicamente il join sia commutativo, dal punto di vista fisico vi è una chiara distinzione, che influenza anche le prestazioni, tra operando sinistro (o "esterno", "outer") e operando destro (o "interno", "inner")

Per semplicità nel seguito parliamo di "*relazione esterna*" e "*relazione interna*" per riferirci agli input del join, ma va ricordato che in generale l'input può derivare dall'applicazione di altri operatori.

## Nested-Loop Join

Date 2 relazioni in input R e S tra cui sono definiti i predicati di join **PJ**, e supponendo che R sia la relazione esterna, l'algoritmo opera come segue:

Per ogni tupla  $t_R$  in R:

```
{ Per ogni tupla  $t_S$  in S:
  { se la coppia  $(t_R, t_S)$  soddisfa PJ
    allora aggiungi  $(t_R, t_S)$  al risultato
  }
}
```

## Esempio

| R | A B  | S | A C D  |  | A C D B  |
|---|------|---|--------|--|----------|
|   | 22 a |   | 22 z 8 |  | 22 z 8 a |
|   | 87 s |   | 45 k 4 |  | 22 s 7 a |
|   | 45 h |   | 22 s 7 |  | 87 s 9 s |
|   | 32 b |   | 87 s 9 |  | 45 k 4 h |
|   |      |   | 32 c 3 |  | 45 h 5 h |
|   |      |   | 45 h 5 |  | 32 c 3 b |
|   |      |   | 32 g 6 |  | 32 g 6 b |

PJ:  $R.A = S.A$



## Costo

Il costo di esecuzione dipende dallo spazio a disposizione nei buffer.

Nel caso base in cui vi sia 1 buffer per **R** e 1 buffer per **S**: bisogna leggere 1 volta **R** e  $NR(R)$  volte **S**, ovvero tante volte quante sono le tuple della relazione esterna, per un totale di  $NP(R) + NR(R) * NP(S)$  accessi a memoria secondaria

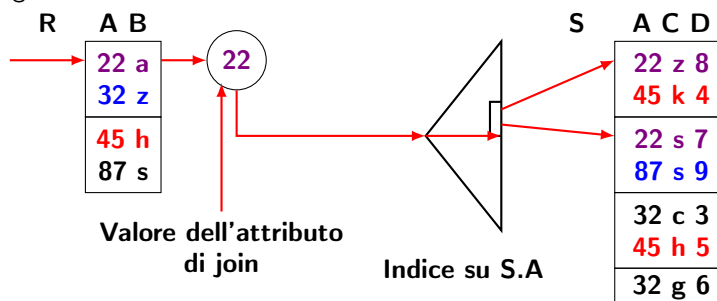
Se è possibile allocare  $NP(S)$  buffer per la relazione interna il costo si riduce a  $NP(R) + NP(S)$  Si ricordi che:

$NR(R)$  = numero tuple di R

$NP(R)$  = numero di pagine di R

## Nested-Loop join con indice

Data una tupla della relazione esterna R, la scansione completa della relazione interna S può essere sostituita da una scansione basata su indice costruito sugli attributi di join di S (nell'esempio A), secondo il seguente schema:



## Nested-Loop JOIN con indice B<sup>+</sup>tree

Nel caso base in cui vi sia 1 buffer per R e 1 buffer per S, bisogna leggere 1 volta R e accedere  $NR(R)$  volte a S, ovvero tante volte quante sono le tuple della relazione esterna, per un totale di:

$$NP(R) + NR(R) \cdot \left( \text{ProfIndice} + \underbrace{NR(S)/VAL(A, S)}_{\text{SELETTIVITÀ di A}} \right)$$

accessi a memoria secondaria

dove:  $VAL(A, S)$  rappresenta il numero di valori distinti di un attributo A della relazione S.

## Merge-Scan JOIN

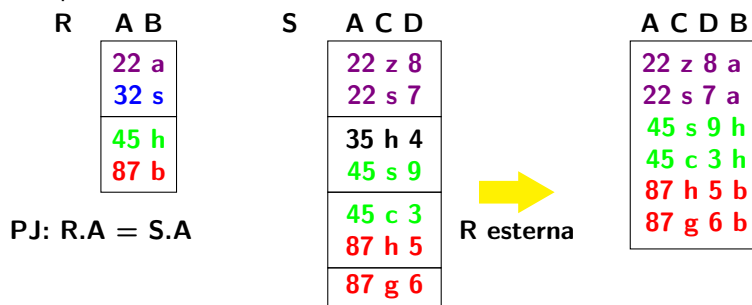
Il Merge-Scan Join è applicabile quando entrambi gli insiemi di tuple in input *sono ordinati sugli attributi di join*.

Ciò accade se per entrambe le relazioni di input R e S vale una almeno delle seguenti condizioni:

- La relazione è fisicamente ordinata sugli attributi di join (*file sequenziale ordinato come struttura fisica*)
- Esiste un indice sugli attributi di join della tabella che consente una scansione ordinata delle tuple.



Esempio:



### Merge-Scan JOIN:Costo

La logica dell'algoritmo sfrutta il fatto che entrambi gli input sono ordinati per evitare di fare inutili confronti, il che fa sì che il numero di letture totali sia dell'ordine di:

$$NP(R) + NP(S)$$

se si accede **sequenzialmente** alle due relazioni.

Con indici il costo può arrivare al massimo a:

$$NR(R) + NR(S)$$

### Hash-based JOIN

L'algoritmo di Hash Join, applicabile solo in caso di equi-join, non richiede né la presenza di indici né input ordinati, e risulta particolarmente vantaggioso in caso di relazioni molto grandi. L'idea su cui si basa l'Hash Join è semplice:

- Si suppone di avere a disposizione una funzione hash  $H$ , che viene applicata agli attributi di join delle due relazioni (ad es.  $R.J$  e  $S.J$ ).
- Se  $t_R$  e  $t_S$  sono 2 tuple di  $R$  e  $S$ , allora è possibile che sia  $t_R.J = t_S.J$  solo se  $H(t_R.J) = H(t_S.J)$
- Se, viceversa,  $H(t_R.J) \neq H(t_S.J)$ , allora sicuramente è  $t_R.J \neq t_S.J$ .

A partire da questa idea si hanno diverse implementazioni, che hanno in comune il fatto che  $R$  e  $S$  vengono partizionate sulla base dei valori della funzione di hash  $H$ , e che la ricerca di "matching tuples" avviene solo tra partizioni relative allo stesso valore di  $H$ .

### Hash-based JOIN: costo

Il costo risulta essere dell'ordine di:

$$NP(R) + NP(S)$$

ma dipende anche fortemente dal numero di buffer a disposizione e dalla distribuzione dei valori degli attributi di join (il caso uniforme è quello migliore)

## 13.3 Scelta finale del piano di esecuzione

Data una espressione ottimizzata in algebra:

- Si generano "tutti i possibili" piani di esecuzione (alberi) alternativi (o un sottoinsieme di questi) ottenuti considerando le seguenti dimensioni:
  - Operatori alternativi applicabili per l'accesso ai dati: ad esempio, scan sequenziale o via indice,
  - Operatori alternativi applicabili nei nodi: ad esempio, nested-loop join or hash-based join
  - L'ordine delle operazioni da compiere (associatività)

- Si valuta con formule approssimate il costo di ogni alternativa in termini di accessi a memoria secondaria richiesti (stima)
- Si sceglie l'albero con costo approssimato minimo.

Nella valutazione delle stime di costo si tiene conto del profilo delle relazioni, solitamente memorizzato nel Data Dictionary e contenente per ogni relazione T:

- Stima della cardinalità ( $\#tuple$ ):  $CARD(T)$
- Stima della dimensione di una tupla:  $SIZE(T)$
- Stima del numero di valori distinti per ogni attributo A:  $VAL(A,T)$
- Stima del valore massimo e minimo per ogni attributo A:  $MAX(A,T)$  e  $MIN(A,T)$

## 14 Interazione tra basi di dati e applicazioni

### 14.1 Tipi di interazione

- **Interazione via cursore** (metodo classico): un API dedicata messa a disposizione dal DBMS consente di sottomettere comandi SQL al sistema e ottenere risultati di interrogazioni rappresentati come cursori (iteratori su liste di tuple).
- **Interazione via cursore con API standardizzata**: come sopra ma con una certa indipendenza dal DBMS (libreria JDBC di Java, ODBC di Microsoft)
- **Object Relational Mapping (ORM)**: è una tecnica che consente di gestire nell'applicazione oggetti "persistenti" e di astrarre dalla base di dati relazionale. Consente un livello di interazione con il DB che maschera l'SQL. Esempi sono: Java Persistence API o Hibernate.

### 14.2 Interazione con DB in Java attraverso JDBC

In Java è possibile interagire con un DBMS attraverso l'uso della libreria JDBC (Java Database Connectivity). <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html>  
JDBC API: fornisce un insieme di classi JAVA che consentono un accesso standardizzato a qualsiasi DBMS purché questo fornisca un driver JDBC.

#### Driver JDBC

È un modulo software in grado di interagire con un DBMS. Traduce ogni invocazione dei metodi delle classi JDBC in comandi SQL accettati dal DBMS a cui è dedicato.

### 14.3 7 Passi per interagire con un DBMS

#### Primo passo

Caricare il driver JDBC per il DBMS che si utilizza.

Per caricare il driver, basta caricare la classe corrispondente:

```
import java.sql.*  
...  
Class.forName("NomeDriver");
```

Per postgresql il nome del driver è: `org.postgresql.Driver`

## Secondo passo

Definire una connessione identificando l'URL della base di dati a cui ci si vuole connettere.

```
String URL = "jdbc:postgresql://dbserver/did2013"
```

dove:

- jdbc:postgresql: = tipo DBMS
- dbserver = DBMS server
- did2013 = database

## Terzo passo

Stabilire una connessione istanziando un oggetto della classe `Connection`:

```
String user = "Utente-postgres";  
String passwd = "Password-utente";  
Connection con = DriverManager.getConnection(  
    URL, user, passwd);
```

## Quarto passo

Utilizzando l'oggetto della classe `Connection` creato al passo precedente, creare un oggetto della classe `Statement` per poter sottomettere comandi al DBMS:

```
Statement stat = con.createStatement();
```

## Quinto passo

Eseguire un'interrogazione SQL o un comando SQL di aggiornamento di una tabella:

- Interrogazione

```
String query = "SELECT * FROM PERSONA";  
ResultSet res = stat.executeQuery(query);
```

- Aggiornamento

```
String update = "UPDATE PERSONA"+  
    "SET NOME = 'Rosa' WHERE id = 1";  
stat.executeUpdate(update);
```

## Sesto passo

Processare il risultato dell'interrogazione attraverso l'applicazione dei metodi della classe `resultSet`. Il `resultSet` è un *corsore* che consente di accedere alle tuple risultato dell'interrogazione:

È possibile scandire in modo sequenziale il contenuto di un `resultSet`:

```
String query = "SELECT * FROM PERSONA";  
resultSet res = stat.executeQuery(query);  
while(res.next()) {  
    .....  
}
```

Il metodo `next()` sposta il puntatore sulla tupla successiva.

È possibile accedere alle proprietà della tupla corrente di un `resultSet` con i seguenti metodi:

- `getXxx(par)`: dove `Xxx` è un tipo base di Java e `par` può essere un indice di posizione o il nome di un attributo della relazione risultato dell'interrogazione; questo metodo restituisce il valore in posizione `par` oppure il valore dell'attributo di nome `par` della tupla corrente.
- `wasNull`: si riferisce all'ultima invocazione di `getXxx` e restituisce `true` se il valore letto era uguale al valore nullo.

... molti altri metodi sono disponibili, si veda:

<http://docs.oracle.com/javase/1.3/docs/api/java/sql/ResultSet.html>

## Settimo passo

Chiudere la connessione usando l'oggetto della classe

```
Connection:
    con.close();
```

## 14.4 Transazioni in JDBC

È possibile eseguire transazioni in JDBC come segue:

```
...
con.setAutoCommit(false);
PreparedStatement ps1 = con.prepareStatement(
    "UPDATE CONTO SET SALDO=SALDO+?" +
    " WHERE NUMERO=? AND FILIALE = 'X'");
ps1.setInt(1, 1000); ps1.setString(2, "358");
ps1.execute();
PreparedStatement ps2 = con.prepareStatement(
    "UPDATE CONTO SET SALDO=SALDO-?" +
    " WHERE NUMERO=? AND FILIALE = 'X'");
ps2.setInt(1, 1000); ps2.setString(2, "876");
ps2.execute();
con.commit();
con.setAutoCommit(true);
```

- Durante la transazione è possibile attivare il trasferimento per blocchi di tuple in un `ResultSet`.
- L'impostazione di default è che l'intero insieme di tuple risultato dell'interrogazione venga trasferito nel `ResultSet`
- Per alterare tale situazione si usa il metodo `setFetchRow(int rows)` di `Statement`
- Usando tale metodo è possibile fare in modo che il trasferimento delle tuple risultato di un'interrogazione dal DBMS all'applicazione avvenga in lotti di al massimo `rows` tuple (quindi eventualmente prevedendo più interazioni per scaricare tutto il risultato)

## Esempio (dal manuale di `psql`)

```
// make sure autocommit is off
conn.setAutoCommit(false);
Statement st = conn.createStatement();
// Turn use of the cursor on.
st.setFetchSize(50);
ResultSet rs = st.executeQuery("SELECT * FROM mytable");
while (rs.next()) {
    System.out.print("a row was returned."); }
rs.close();
// Turn the cursor off.
st.setFetchSize(0);
```

## 14.5 Object Relational Mapping

Architettura:

- Uno strato software (gestore della persistenza) si interpone tra l'applicazione e il DB. L'applicazione vede il DB come insieme di oggetti persistenti.
- Viene specificato in un file di mapping (o direttamente sulle classi Java con annotazioni) la corrispondenza tra oggetti persistenti e tabelle del DB.

Vantaggi:

- Il gestore della persistenza genera parte delle interrogazioni SQL necessarie per generare gli oggetti estraendo dati dalla base di dati in modo trasparente al programmatore.
- La navigazione nei dati segue puntatori tra oggetti

Svantaggi:

- Le interrogazioni più complesse (che non possono sfruttare i puntatori tra oggetti) vanno gestite ad hoc con SQL nativo.
- Le interrogazioni ad elevata cardinalità possono caricare l'intero DB in memoria (rischio di saturazione della memoria dedicata all'applicazione)
- I meccanismi di generazione automatica degli oggetti riferiti da altri possono caricare l'intero DB in memoria!

## Altri approcci per l'interazione tra DB e applicazioni

Evoluzioni in diverse direzioni per superare i problemi legati al cosiddetto "impedance mismatch"

Lato DBMS (dal relazionale verso nuovi modelli)

- **Object-relational model (SQL3):** tuple (oggetti) con struttura complessa, tabelle (classi) con ereditarietà, navigazione tra le tuple attraverso riferimenti diretti (ref), SQL esteso per interrogare gerarchie di tuple.
- **Dati semistrutturati:** dati a struttura complessa e a schema variabile, rappresentazione con linguaggi a marca (XML), interrogazioni con X-Path, X-Query.
- **Document-based models (NoSQL database):** collezioni di documenti con struttura complessa, dati ridondanti e voluminosi
- **Sistemi basati su Cluster per Big Data:** collezione di dati a struttura complessa e variabile, gestiti da sistemi diversi (dato distribuito), dati voluminosi, interrogazioni distribuite ed eseguite in parallelo.

## 15 Tecnologie NoSQL

- **Sistemi Key-Value:** tutti i dati sono rappresentati per mezzo di coppie (chiave, valore), dove la chiave identifica univocamente le istanze e valore può avere una qualsiasi struttura anche non omogenea nella collezione.
- **Sistemi Document-Store:** i dati sono rappresentati come collezioni di documenti (oggetti con struttura complessa). Ogni oggetto della collezione ha una struttura complessa (con dati incapsulati) senza schema fisso. Consentono la definizione di indici secondari su proprietà degli oggetti (attributi comuni). Esempi di questo approccio sono MongoDB e CouchDB.
- **Sistemi Extensible-Record-Store:** i dati sono rappresentati da collezioni di record, dette tabelle, che possono essere nidificate e a struttura variabile (BigTable di Google, HBase, HyperTable)

## 15.1 Sistemi document-store

### Modelli dei dati dei sistemi Document-store

Le caratteristiche principali di tali modelli sono:

- I dati sono rappresentati da collezioni di oggetti (detti, documenti).
- Gli oggetti hanno struttura complessa (non sono tuple piatte ma contengono dati nidificati).
- La nidificazione (encapsulation) dei dati è un aspetto chiave di questi modelli.
- Gli oggetti di una collezione non devono avere necessariamente tutti la stessa struttura; tuttavia dovrebbero condividere un nucleo di proprietà comuni.

### Progettazione dei dati come documenti

La decisione chiave nel processo di progettazione dei dati usando questo nuovo approccio riguarda: la struttura dei documenti e in particolare la rappresentazione delle relazioni (legami) tra i documenti. Esistono in particolare due approcci per rappresentare le relazioni tra documenti:

- Attraverso riferimenti
- Attraverso documenti nidificati

L'uso di riferimenti per rappresentare i legami tra documenti porta a schemi normalizzati. Ad esempio, in **MongoDB**:

Listing 1: Student Document

```
{
  _id: <ObjID1>
  name: "Mario"
  surname: "Rossi"
}
```

Listing 2: Exam Document

```
{
  _id: <ObjID2>
  student_id: <ObjID1>
  course: "Basi di dati"
  grade: 25
}
```

Listing 3: Contact Document

```
{
  _id: <ObjID19>
  student_id: <ObjID1>
  email: "mr@aaa.bb"
}
```

L'uso di documenti nidificati porta a schemi non normalizzati e parzialmente ridondanti, ma molto efficienti in lettura.

Ad esempio, in MongoDB:

Listing 4: Student Document

```
{
  _id: <ObjID1>
  name: "Mario"
  surname: "Rossi"
  exams: {course: "Basi di dati"
          grade: 2 }
  contacts: {email: "mr@aaa.bb"
            tel: 1234567 }
}
```

L'**approccio normalizzato** consente di rappresentare il dato in modo simile a quanto avviene nel modello relazionale. Non c'è ridondanza e gli aggiornamenti sono più semplici (riguardano sempre una sola istanza).

L'**approccio embedded** consente di rappresentare in un'unica istanza quanto viene rappresentato di solito nel modello ER con una entità insieme alle sue entità deboli.

Vale a dire posso rappresentare nello stesso documento un oggetto con altre istanze di informazione in esso logicamente contenute. In questo caso il documento può essere aggiornato come oggetto complesso in modo atomico.