

PER ALTRI APPUNTI CONSULTARE IL SITO:  
[https://luigi-v.github.io/Appunti\\_Universita/](https://luigi-v.github.io/Appunti_Universita/)

## 1. DIPENDENZE FUNZIONALI E NORMALIZZAZIONE PER BASI DI DATI RELAZIONALI

Ogni **schema di relazione** è composto da un certo numero di attributi, e lo **schema di base di dati relazionale** è composto da un certo numero di schemi di relazione. Il modello EER fa sì che il progettista individui i tipi di entità e associazione coi loro attributi, il che porta a un raggruppamento naturale e logico degli attributi in relazioni quando sono seguite le procedure di traduzione. Ma si ha il bisogno di misurare formalmente il perché un raggruppamento di attributi in uno schema di relazione possa essere meglio di un altro, misurando la "bontà" o la qualità del progetto.

Esistono due livelli che esaminano la "bontà":

- **Livello logico (o concettuale)**, come gli utenti interpretano gli schemi di relazione e il significato dei loro attributi;
- **Livello d'implementazione (o fisico)**, come le tuple in una relazione di base sono memorizzate e aggiornate.

### 1.1 LINEE GUIDA INFORMALI DI PROGETTAZIONE DI SCHEMI DI RELAZIONE

Ci sono 4 **misure informali** di qualità per la progettazione di uno schema di relazione:

#### 1. Semanticità degli attributi:

Ogni volta che si raggruppano degli attributi per formare uno schema di relazione, si suppone che ad essi sia associato un certo significato. Ogni relazione può essere interpretata come un insieme di fatti o asserzioni. Questa è la **semanticità**, che specifica come interpretare i valori degli attributi memorizzati in una tupla della relazione (o come i valori sono in relazione tra loro).

Il significato dello schema di relazione IMPIEGATO è semplice, ogni tupla rappresenta un impiegato, con valori come nome, indirizzo, ecc...

EMPLOYEE					
ENAME	SSN	BDATE	ADDRESS	DNUMBER	
Smith,John B.	123456789	1965-01-29	731 Fondren,Houston,TX	5	
Wong,Franklin T.	333445555	1965-12-08	638 Voss,Houston,TX	5	
Zelaya,Alicia J.	999887777	1968-07-19	3321 Castle Spring,TX	4	
Wallace,Jennifer S.	987654321	1941-06-20	291 Berry,Bellaire,TX	4	
Narayan,Ramesh K.	666884444	1962-09-15	975 Fire Oak,Humble,TX	5	
English,Joyce A.	453453453	1972-07-31	5631 Rice,Houston,TX	5	
Jabbar,Ahmed V.	987987987	1969-03-29	983 Dallas,Houston,TX	4	
Borg,James E.	888665555	1937-11-10	450 Stone,Houston,TX	1	

DEPARTMENT		DEPT_LOCATIONS	
DNAME	DNUMBER	DNMGRSSN	DLOCATION
Research	5	333445555	1 Houston
Administration	4	887654321	4 Stafford
Headquarters	1	888665555	5 Bellaire
			5 Sugarland
			5 Houston

**LINEA GUIDA 1:** Si progettano ogni schema di relazione in modo tale che sia semplice spiegarne il significato. Non si uniscono attributi provenienti da più tipi di entità e tipi di associazione in un'unica relazione. Se uno schema di relazione corrisponde a un solo tipo di entità o a un solo tipo di associazione, il suo significato tende ad essere chiaro.

#### 2. Riduzione dei valori ridondanti nelle tuple:

Uno scopo della progettazione di schemi è quello di ridurre al minimo lo spazio di memoria occupato dalle relazioni di base. Il raggruppamento di attributi in schemi di relazione ha un effetto significativo sullo spazio di memoria.

Ad esempio, in IMP\_DIP, i valori degli attributi che riguardano uno specifico dipartimento sono ripetuti per ogni impiegato che lavora per quel dipartimento.

Al contrario, l'informazione su ogni dipartimento appare solo una volta nella relazione DIPARTIMENTO. In tal caso lo spazio di memoria cambia radicalmente.

Un altro problema che si presenta è il problema delle **anomalie di aggiornamento**, che possono essere classificate in:

- **Anomalia di inserimento**, per inserire una nuova tupla impiegato in IMP\_DIP occorre inserire i valori degli attributi del dipartimento per cui l'impiegato lavora, o valori nulli. Un altro problema è nell'inserire nella relazione IMP\_DIP un nuovo dipartimento che non ha ancora impiegati, il solo modo per farlo è inserire valori nulli (anche nella chiave primaria).
- **Anomalia di cancellazione**, se si cancella da IMP\_DIP una tupla impiegato che è quella che rappresenta l'ultimo impiegato che lavora per un dipartimento, l'informazione del quel dipartimento non sarà più presente.
- **Anomalia di modifica**, in IMP\_DIP se si cambia un attributo di un dipartimento occorre aggiornare le tuple di tutti gli impiegati che lavorano per quel dipartimento.

Tutte queste anomalie non si presentano nella relazione IMPIEGATO e non bisogna preoccuparsi di questi problemi di consistenza.

**LINEA GUIDA 2:** Si progettino gli schemi di relazione di base in modo che nelle relazioni non siano presenti le anomalie menzionate.

#### 3. Riduzione del numero di valori nulli nelle tuple:

È possibile che nei progetti vengano raggruppati numerosi attributi a formare una grossa relazione, se molti attributi non riguardano tutte le tuple della relazione, quelle tuple saranno nulle. Ciò può dar luogo ad uno spreco di memoria e problemi di comprensione del significato. Inoltre, i valori nulli possono avere più interpretazioni, come:

- L'attributo non è presente per questa tupla;
- Il valore dell'attributo è sconosciuto;
- Il valore è noto ma assente (non ancora memorizzato).

**LINEA GUIDA 3:** Per quanto possibile, si eviti di porre in una relazione attributi i cui valori possono essere frequentemente nulli. Se sono inevitabili, ci si assicuri che essi si presentino solo in casi eccezionali.

#### 4. Impossibilità di generare tuple spurie:

Si supponga di aver usato IMP\_PROG1 e IMP\_SEDI invece di IMP\_PROG come relazioni di base. Ciò dà luogo a un progetto di schema infelice, perché da IMP\_PROG e IMP\_SEDI non si può recuperare l'informazione originariamente presente in IMP\_PROG. Se si tenta un'operazione di JOIN NATURALE su IMP\_PROG1 e IMP\_SEDI, il risultato del join produce molte più tuple rispetto all'originaria popolazione. Tuple aggiuntive sono dette **tuple spurie** perché rappresentano un'informazione spuria o sbagliata che non è valida.

**LINEA GUIDA 4:** Si progettano schemi di relazione in modo tale che essi possano essere riuniti tramite JOIN con condizioni di uguaglianza su attributi che sono o PK o FK in modo da garantire che non vengano generate tuple spurie.

EMP_LOCS		EMP_PROJ1				
ENAME	PLOCATION	SSN	PNUMBER	HOURS	PNAME	PLOCATION
Smith, John B.	Bellaire	123456789	1	32.5	Product X	Bellaire
Smith, John B.	Sugarland	123456789	2	7.5	Product Y	Sugarland
Narayan, Ramesh K.	Houston	666884444	3	40.0	Product Z	Houston
English, Joyce A.	Bellaire	453453453	1	20.0	Product X	Bellaire
English, Joyce A.	Sugarland	333445555	2	20.0	Product Y	Sugarland
Wong, Franklin T.	Sugarland	333445555	3	10.0	Product Z	Houston
Wong, Franklin T.	Houston	333445555	10	10.0	Computerization	Stafford
Wong, Franklin T.	Stafford	333445555	20	10.0	Reorganization	Houston

SSN	PNUMBER	HOURS	PNAME	PLOCATION	ENAME
123456789	1	32.5	Product X	Bellaire	Smith,John B.
123456789	1	32.5	Product X	Bellaire	English, Joyce A.
123456789	2	7.5	Product Y	Sugarland	Smith,John B.
+ 123456789	2	7.5	Product Y	Sugarland	English, Joyce A.
+ 123456789	2	7.5	Product Y	Sugarland	Wong, Franklin T.
666884444	3	40.0	Product Z	Houston	Narayan, Ramesh K.
+ 666884444	3	40.0	Product Z	Houston	Wong, Franklin T.
+ 453453453	1	20.0	Product X	Bellaire	Smith,John B.
+ 453453453	1	20.0	Product X	Bellaire	English, Joyce A.
+ 453453453	2	20.0	Product Y	Sugarland	Smith,John B.

## 1.2 DIPENDENZE FUNZIONALI

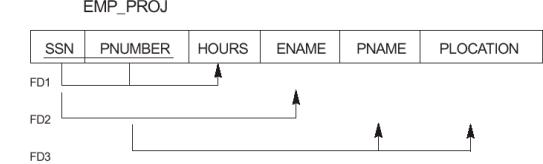
Una **dipendenza funzionale** è un vincolo tra due insiemi di attributi della base di dati. Supponendo che lo schema relazionale abbia  $n$  attributi  $A_1, A_2, \dots, A_n$ ,  $A_n$  e si pensi all'intera base di dati come se fosse descritta da un solo schema di **relazione universale**  $R = \{A_1, A_2, \dots, A_n\}$ .

Una **dipendenza funzionale (DF)**, indicata con  $X \rightarrow Y$ , tra due insiemi di attributi  $X$  e  $Y$  che siano sottoinsiemi di  $R$  specifica un **vincolo** sulle tuple che possono formare uno stato di relazione  $r$  di  $R$ . Il vincolo è che per ogni coppia di tuple  $t_1$  e  $t_2$  in  $r$  per le quali è  $t_1[X] = t_2[X]$ , si deve avere anche  $t_1[Y] = t_2[Y]$ . Ciò significa che i valori della componente  $Y$  di una tupla in  $r$  dipendono da, o sono **determinati da**, i valori della componente  $X$ , o, in alternativa, che i valori della componente  $X$  di una tupla **determinano** univocamente (o **funzionalmente**) i valori della componente  $Y$ . Si dice anche che  $Y$  è **funzionalmente dipendente** da  $X$ . L'insieme degli attributi  $X$  è detto **parte sinistra** della DF, e  $Y$  è detto **parte destra**.

Ogni volta che la semantica di due insiemi di attributi in  $R$  indica che deve sussistere una dipendenza funzionale, la dipendenza viene specificata con un vincolo. Estensioni di relazione  $r(R)$  che soddisfano i vincoli di dipendenza funzionale sono dette **estensioni valide** (o **stati validi di relazione**) di  $R$ , perché soddisfano i vincoli di dipendenza funzionale. L'uso principale delle dipendenze funzionali è perciò quello di descrivere ulteriormente uno schema di relazione  $R$ , tramite una specificazione dei vincoli sui suoi attributi che devono valere sempre.

Dalla semantica degli attributi sappiamo che devono sussistere le seguenti dipendenze funzionali (rappresentate nello schema da linee):

- a.  $SSN \rightarrow NOME\_I$  (il valore di  $SSN$  determina univocamente  $NOME\_I$ )
- b.  $NUMERO\_P \rightarrow \{NOME\_P, SEDE\_P\}$
- c.  $\{SSN, NUMERO\_P\} \rightarrow ORE$



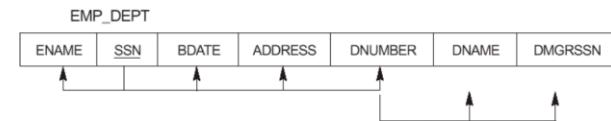
### REGOLE DI INFERENZA PER DIPENDENZE FUNZIONALI:

Si indica con ***F l'insieme di DF*** specificate sullo schema di relazione  $R$ . Tipicamente non possono essere dedotte tutte le DF semanticamente, alcune possono essere **inferite** o **dedotte** dalle DF in  $F$ . L'insieme di tutte queste dipendenze è detto **chiusura di F**, indicato con  $F^+$ . Ad esempio:

- $F = \{SSN \rightarrow \{NOME\_I, DATA\_N, INDIRIZZO, NUMERO\_D\}, NUMERO\_D \rightarrow \{NOME\_D, DMGRSSN\}\}$

È possibile **inferire** da  $F$  le DF aggiuntive:

- $SSN \rightarrow \{NOME\_I, DATA\_N, INDIRIZZO, NUMERO\_D\}$
- $SSN \rightarrow SSN$
- $NUMERO\_D \rightarrow NOME\_D$



Una DF  $X \rightarrow Y$  è **inferita** da un insieme di dipendenze  $F$  specificate su  $R$  se  $X \rightarrow Y$  sussiste in ogni stato di relazione  $r$  che sia un'estensione valida di  $r$ , cioè, ogni volta che  $r$  soddisfa tutte le dipendenze in  $F$ , in  $r$  sussiste anche  $X \rightarrow Y$ .

La **chiusura  $F^+$  di  $F$  è l'insieme di tutte le DF che possono essere dedotte da  $F$** . Verrà usata la notazione  $F \rightarrow X \rightarrow Y$  per indicare che la DF  $X \rightarrow Y$  è inferita dall'insieme di DF  $F$ . Esistono sei **regole di inferenza** per DF:

**RI1 (regola riflessiva):**

$$\{X \rightarrow Y\} \vdash X \supseteq Y, \text{ allora } X \rightarrow Y$$

Afferma che un insieme di attributi determina sempre sé stesso o uno qualsiasi dei suoi sottoinsiemi.

**Formalmente una DF  $X \rightarrow Y$  è banale se  $X \supseteq Y$ , altrimenti è non-banale.**

**PROVA DI RI1:**

Si supponga che  $X \supseteq Y$  e che esistano due tuple  $t_1$  e  $t_2$  in una certa istanza di relazione  $r$  di  $R$  tali che  $t_1[X] = t_2[X]$ . Allora  $t_1[Y] = t_2[Y]$  perché  $X \supseteq Y$ , perciò in  $r$  deve valere  $X \rightarrow Y$ .

**RI2 (regola di arricchimento):**

$$\{X \rightarrow Y\} \vdash XZ \rightarrow YZ$$

Sostiene che aggiungendo lo stesso insieme di attributi alla parte sinistra e destra di una dipendenza si ottiene un'altra dipendenza valida.

**PROVA DI RI2 (per assurdo):**

Si supponga che in un'istanza di relazione  $r$  di  $R$  valga la  $X \rightarrow Y$ , ma che non valga la  $XZ \rightarrow YZ$ . Devono perciò esistere due tuple  $t_1$  e  $t_2$  in  $r$  tali che:

$$(1) \quad t_1[X] = t_2[X], \quad (2) \quad t_1[Y] = t_2[Y], \quad (3) \quad t_1[XZ] = t_2[XZ] \text{ e } (4) \quad t_1[YZ] \neq t_2[YZ].$$

Ciò non è possibile, perché da (1) e (3) si deduce (5)  $t_1[Z] = t_2[Z]$ , e da (2) e (5) si deduce (6)  $t_1[YZ] = t_2[YZ]$ , contraddicendo (4).

**RI3 (regola transitiva):**

$$\{X \rightarrow Y, Y \rightarrow Z\} \vdash X \rightarrow Z$$

Le dipendenze funzionali sono transitive.

**PROVA DI RI3:**

Si supponga che in una relazione  $r$  sussistano sia (1)  $X \rightarrow Y$  che (2)  $Y \rightarrow Z$ . Allora per ogni coppia di tuple  $t_1$  e  $t_2$  in  $r$  tali che  $t_1[X] = t_2[X]$ , si deve avere (3)  $t_1[Y] = t_2[Y]$ , dall'assunzione (1), perciò occorre anche avere (4)  $t_1[Z] = t_2[Z]$ , dalla (3) e dall'assunzione (2), quindi in  $r$  deve valere la  $X \rightarrow Z$ .

**RI4 (regola di decomposizione/proiezione):**

$$\{X \rightarrow YZ\} \vdash X \rightarrow Y$$

Sostiene che si possono rimuovere attributi dalla parte destra di una dipendenza, l'applicazione ripetuta di questa regola può decomporre la DF  $X \rightarrow \{A_1, A_2, \dots, A_n\}$  nell'insieme di dipendenze  $\{X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n\}$ .

**PROVA DI RI4 (usando da RI1 a RI3):**

1.  $X \rightarrow YZ$  (data)
2.  $YZ \rightarrow Y$  (usando RI1 e sapendo che  $YZ \supseteq Y$ )
3.  $X \rightarrow Y$  (usando RI3 su 1 e 2)

**RI5 (regola di unione/additiva):**

$$\{X \rightarrow Y, X \rightarrow Z\} \vdash X \rightarrow YZ$$

Consente di fare l'opposto della RI4, è possibile combinare un insieme di dipendenze  $\{X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n\}$  nelle singole DF  $X \rightarrow \{A_1, A_2, \dots, A_n\}$ .

**PROVA DI RI5 (usando da RI1 a RI3):**

1.  $X \rightarrow Y$  (data)
2.  $X \rightarrow Z$  (data)
3.  $X \rightarrow XY$  (usando RI2 su 1 arricchendo con  $X$ , si noti che  $XX = X$ )
4.  $XY \rightarrow YZ$  (usando RI2 su 2 arricchendo con  $Y$ )
5.  $X \rightarrow YZ$  (usando RI3 su 3 e 4)

**RI6 (regola pseudo transitiva):**

$$\{X \rightarrow Y, WY \rightarrow Z\} \vdash WX \rightarrow Z$$

**PROVA DI RI6 (usando da RI1 a RI3):**

1.  $X \rightarrow Y$  (data)
2.  $WY \rightarrow Z$  (data)
3.  $WX \rightarrow WY$  (usando RI2 su 1 arricchendo con  $W$ )
4.  $WX \rightarrow Z$  (usando RI3 su 3 e 2)

## REGOLE DI INFERNZA DI ARMSTRONG:

È stato dimostrato da Armstrong che le prime 3 regole di inferenza sono **corrette** e **complete**. Per **corrette** si intende che, dato un insieme di dipendenze funzionali F specificate su uno schema di relazione R, tutte le dipendenze che è possibile inferire da F usando le regole da RI1 a RI3 sussistono in ogni stato di relazione r di R che soddisfa le dipendenze in F. Per **complete** si intende che, usando ripetutamente le regole da RI1 a RI3 per inferire dipendenze finché non se ne possono dedurre più, si ottiene come risultato l'insieme completo di tutte le possibili dipendenze che possono essere dedotte da F. In altre parole, l'insieme di dipendenze F, che è stato detto chiusura di F, può essere determinato da F usando solo le regole di inferenza da RI1 a RI3. Le regole di inferenza da RI1 a RI3 sono note come **regole di inferenza di Armstrong**.

Un modo sistematico per determinare queste dipendenze funzionali aggiuntive è quello di determinare prima di tutto ogni insieme X di attributi che appare come parte sinistra di qualche dipendenza funzionale in F, e poi di determinare l'insieme di tutti gli attributi che sono dipendenti da X. Perciò per ogni insieme X di attributi di questo tipo, si calcola l'insieme X<sup>+</sup> di attributi che sono determinati funzionalmente da X sulla base di F. X<sup>+</sup> è detto **chiusura di X rispetto a F**. Per calcolare X<sup>+</sup> può essere usato l'**Algoritmo 1**:

```
X+ := X;
repeat
    oldX+ := X+;
    for each functional dependency Y → Z in F do
        if X+ ⊇ Y
            then X+ := X+ ∪ Z;
    until (oldX+ = X+);
```

Tale algoritmo inizia ponendo X<sup>+</sup> uguale a tutti gli attributi in X. Da RI1 è noto che tutti questi attributi sono funzionalmente dipendenti da X. Servendosi delle regole di inferenza RI3 e RI4 si aggiungono attributi a X<sup>+</sup>, usando tutte le dipendenze funzionali in F. Si continuano a considerare tutte le dipendenze in F (il ciclo repeat) finché non vengono più aggiunti attributi a X<sup>+</sup> durante un ciclo completo (il ciclo for) sulle dipendenze in F.

Ad esempio:

F = {SSN → NOME\_I, NUMERO\_P → {NOME\_P, SEDE\_P},  
{SSN, NUMERO\_P} → ORE }

→ {SSN}<sup>+</sup> = {SSN, NOME\_I}  
{NUMERO\_P}<sup>+</sup> = {NUMERO\_P, NOME\_P, SEDE\_P}  
{SSN, NUMERO\_P}<sup>+</sup> = {SSN, NUMERO\_P, NOME\_I, NOME\_P, SEDE\_P, ORE}

## EQUIVALENZE DI INSIEMI DI DIPENDENZE FUNZIONALI:

Un insieme di dipendenze funzionali E è **coperto da** un insieme di dipendenze funzionali F, alternativamente si dice che F **copre** E, se ogni DF in E è presente anche in F<sup>+</sup>, cioè se ogni dipendenza in E può essere inferita da F. Due insiemi E e F di dipendenze funzionali sono **equivalenti** se E<sup>+</sup> = F<sup>+</sup>. Perciò l'equivalenza implica che ogni DF in E possa essere inferita da F, e ogni DF in F possa essere inferita da E, ossia E è equivalente a F se sussistono entrambe le condizioni E copre F e F copre E. Si può determinare se F copre E calcolando X<sup>+</sup> rispetto a F per ogni DF X → Y in E, e quindi verificando se questo comprende gli attributi presenti in Y. Se è così per ogni DF in E, allora F copre E.

## INSIEMI MINIMALI DI DIPENDENZE FUNZIONALI:

Un insieme F di dipendenze funzionali è **minimale** se soddisfa le condizioni:

1. Ogni dipendenza presente in F ha come parte destra un solo attributo;
2. Non è mai possibile sostituire una dipendenza X → A di F con una dipendenza Y → A, dove Y è un sottoinsieme proprio di X, e avere ancora un insieme di dipendenze equivalenti a F;
3. Non è mai possibile rimuovere una dipendenza da F e avere ancora un insieme di dipendenze equivalenti a F.

Si può pensare a un insieme minimale di dipendenze come a un insieme di dipendenze in una *forma canonica o standard e senza ridondanze*.

Una copertura minimale di un insieme F di DF è un insieme minimale di dipendenze F<sub>min</sub> equivalente a F (possono esistere più coperture minime).

## 1.3 FORME NORMALI

Il processo di **normalizzazione** sottopone uno schema di relazione a una serie di test per "certificare" se soddisfa una certa forma normale. La **normalizzazione dei dati** può essere considerata come un processo di analisi degli schemi di relazione forniti per raggiungere le proprietà desiderate di **minimizzare la ridondanza e anomalie**.

Una **superchiave** di uno schema di relazione R = {A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>n</sub>} è un insieme di attributi S ⊆ R con la proprietà che nessuna coppia di tuple t<sub>1</sub> e t<sub>2</sub> in un generico stato valido di relazione r di R avrà t<sub>1</sub>[S] = t<sub>2</sub>[S]. Una **chiave** K è una superchiave con la proprietà aggiuntiva che la rimozione di qualsiasi attributo da K fa sì che K cessi di essere una superchiave. La differenza tra una chiave e superchiave sta nel fatto che la chiave deve essere minimale. Se uno schema di relazione ha più chiavi, esse sono tutte **chiavi candidate** e solo una di queste sarà **chiave primaria**.

Un attributo di uno schema relazionale R è detto **attributo primo** di R se è membro di qualche **chiave candidata** di R, altrimenti è detto non-primo.

## PRIMA FORMA NORMALE:

La **prima forma normale (1NF)** è stata definita per non permettere l'uso di attributi multivale, composti o loro combinazioni. Essa richiede che il dominio di un attributo comprenda solo **valori atomici** e che il valore di qualsiasi attributo in una tupla sia un **valore singolo** del suo dominio.

(a) DIPARTIMENTO			
NOME_D	NUMERO_D	SSN_DIR_DIP	SEDI_D

(b) DIPARTIMENTO			
NOME_D	NUMERO_D	SSN_DIR_DIP	SEDI_D
Ricerca	5	333445555	{Bellaire, Sugarland, Houston}
Amministrazione	4	987654321	{Stafford}
Sede centrale	1	888665555	{Houston}

(c) DIPARTIMENTO			
NOME_D	NUMERO_D	SSN_DIR_DIP	SEDE_D
Ricerca	5	333445555	Bellaire
Ricerca	5	333445555	Sugarland
Ricerca	5	333445555	Houston
Amministrazione	4	987654321	Stafford
Sede centrale	1	888665555	Houston

Figura 10.8 Normalizzazione in 1NF. (a) Schema di relazione che non è in 1NF. (b) Istanza di relazione d'esempio. (c) Relazione in 1NF con ridondanza.

La prima soluzione è la migliore, perché non presenta ridondanze ed è generale, non presentando limiti sul numero massimo di valori.

La 1NF non permette neppure attributi multivale che siano essi stessi composti. Questa relazione sono dette **relazioni nidificate** perché ogni tupla può avere al suo interno una relazione.

## SECONDA FORMA NORMALE:

La **seconda forma normale** (**2NF**) si basa sul concetto di *dipendenza funzionale completa*.

Una DF  $X \rightarrow Y$  è una **DF completa** se la rimozione di qualsiasi attributo A da X comporta che la dipendenza non sussista più, cioè, per ogni attributo  $A \in X$ ,  $(X - \{A\}) \nrightarrow Y$ . È parziale, invece, se si possono rimuovere certi attributi e la dipendenza continua a sussistere, cioè, per qualche  $A \in X$ ,  $(X - \{A\}) \rightarrow Y$ .

Ad esempio,  $\{\text{SSN}, \text{NUMERO\_P}\} \rightarrow \text{ORE}$  è una dipendenza completa, mentre  $\{\text{SSN}, \text{NUMERO\_P}\} \rightarrow \text{NOME\_I}$  è invece parziale.

Il test per la 2NF comporta l'esame di DF i cui attributi di parte sinistra fanno parte della PK. Se la PK contiene un solo attributo il test è inutile.

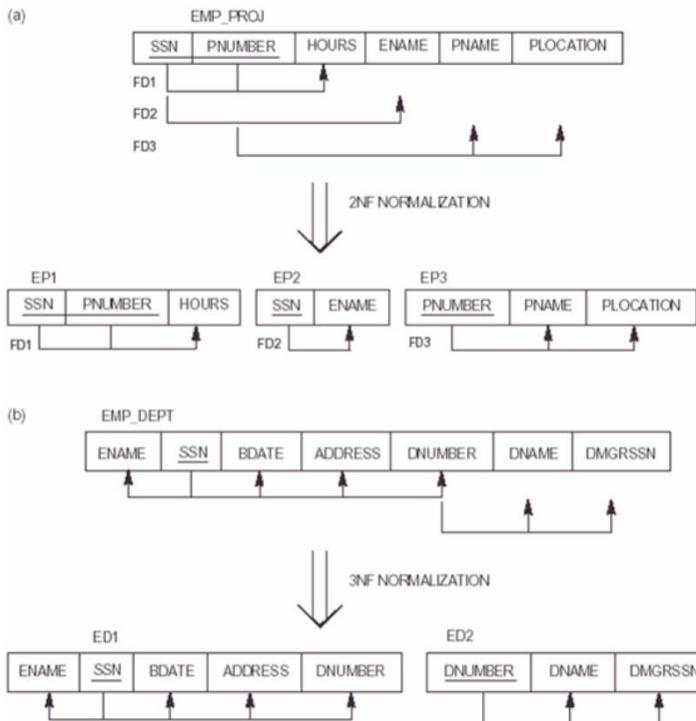
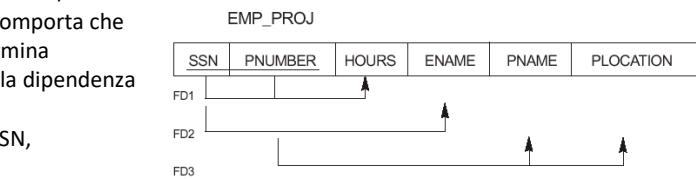
Uno schema di relazione R è in 2NF se ogni attributo non-primo A di R dipende funzionalmente in modo completo dalla PK di R. Nella relazione **IMP\_PROJ**, l'attributo non-primo **NOME\_I** viola la 2NF a causa di DF2, ma anche gli attributi non-primi **NOME\_P** e **SEDE\_P** a causa di DF3. Questi 3 attributi sono parzialmente dipendenti dalla chiave primaria (composta). Per rispettare la 2NF la relazione la si scomponga in un certo numero di relazioni in 2NF, nelle quali gli attributi non-primi sono associati solo alla parte della PK da cui sono funzionalmente dipendenti in modo completo.

## TERZA FORMA NORMALE:

La **terza forma normale** (**3NF**) è basata sul concetto di *dipendenza transitiva*. Una DF  $X \rightarrow Y$  in uno schema di relazione R è una dipendenza transitiva se esiste un insieme di attributi Z, che non è né una chiave candidata né un sottoinsieme di chiavi di R, per cui valgono contemporaneamente  $X \rightarrow Z$  e  $Z \rightarrow Y$ .

Ad esempio,  $\text{SSN} \rightarrow \text{DMGRSSN}$  in **IMP\_DIP** è transitiva attraverso **NUMERO\_D** perché suscitano entrambe le dipendenze  $\text{SSN} \rightarrow \text{NUMERO_D}$  e  $\text{NUMERO_D} \rightarrow \text{DMGRSSN}$ , e **NUMERO\_D** non è né chiave di per sé e né un sottoinsieme delle chiavi di **IMP\_DIP**.

Stando alla definizione uno schema di relazione è in 3NF se è in 2NF e nessun attributo non-primo di R dipende in modo transitivo dalla chiave primaria.

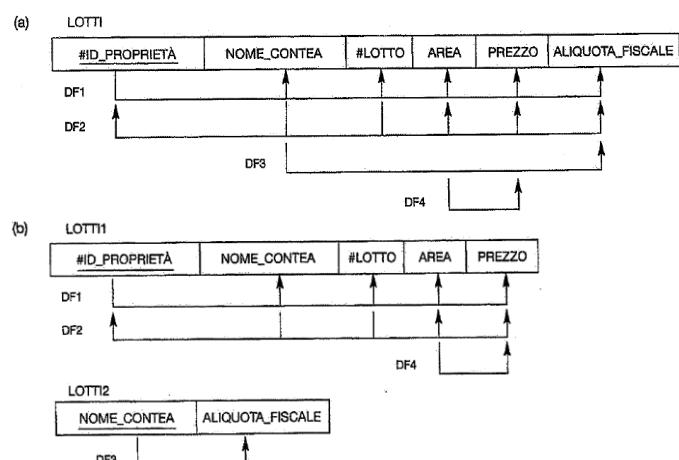


FORMA NORMALE	TEST	RIMEDIO (NORMALIZZAZIONE)
1NF	La relazione non deve avere attributi non-atomici o relazioni nidificate.	Formare nuove relazioni per ogni attributo non-atomico o relazione nidificata.
2NF	Per relazioni in cui la PK contiene più attributi, nessun attributo non-chiave deve essere funzionalmente dipendente da una parte della PK.	Decomporre e preparare una nuova relazione per ogni chiave parziale coi suoi attributi dipendenti. Assicurarsi di mantenere una relazione con la PK originale e tutti gli attributi funzionalmente dipendenti in modo completo da essa.
3NF	La relazione non deve avere un attributo non-chiave determinato funzionalmente da un altro attributo non-chiave (o da un insieme di attributi non-chiave). Cioè non deve esserci nessuna dipendenza transitiva di un attributo non-chiave dalla PK.	Decomporre e preparare una relazione che comprenda gli attributi non-chiave che determinano funzionalmente altri attributi non-chiave.

## DEFINIZIONI GENERALI DI SECONDA FORMA NORMALE:

Considerando lo schema di relazione **LOTTI**, si supponga che ci siano due chiavi candidate **#ID\_PROPRIETÀ** e **{NOME\_CONTEA, #LOTTO}**, ciò significa che i numeri di lotto sono univoci solo all'interno di ogni contea, ma i numeri di **ID\_PROPRIETÀ** sono univoci per tutte le contee dell'intero stato. Sulla base delle due chiavi candidate si sussistono le DF1 e DF2, si sceglie poi **#ID\_PROPRIETÀ** come PK. Tenendo in considerazione le DF3 e DF4, lo schema **LOTTI** viola la 2NF perché **ALIQUOTA\_FISCALE** dipende in modo parziale dalla chiave candidata **{NOME\_CONTEA, #LOTTO}**.

Per normalizzare **LOTTI** in 2NF lo si decompone nelle due relazioni **LOTTI1** e **LOTTI2**. Sia **LOTTI1** che **LOTTI2** sono in 2NF, si noti che DF4 non viola la 2NF ed è riportata in **LOTTI1**.



## DEFINIZIONI GENERALI DI TERZA FORMA NORMALE:

Considerando LOTTI2 è in 3NF, però DF4 in LOTTI1 viola la 3NF perché AREA non è una superchiave e PREZZO non è un attributo primo.

Per normalizzare LOTTI1 lo si decompone negli schemi di relazione LOTTI1A e LOTTI1B. LOTTI1 viola la 3NF perché PREZZO è transitivamente dipendente da ognuna delle chiavi candidate di LOTTI1 tramite l'attributo non-primo AREA.

Non è necessario passare prima attraverso 2NF, se si applica la definizione di 3NF su LOTTI con DF1 e DF4, si trova che DF3 e DF4 violano entrambe la 3NF. Si decompone direttamente LOTTI in LOTTI1A, LOTTI1B e LOTTI2.

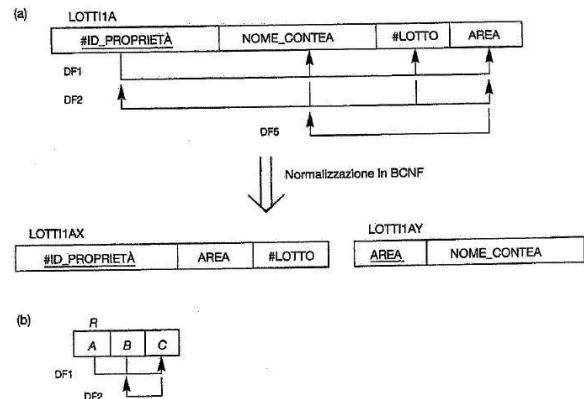
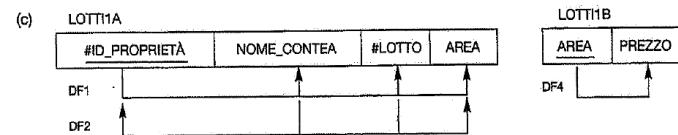
## FORMA NORMALE DI BOYCE E CODD:

La **forma normale di Boyce e Codd (BCNF)** è una forma più restrittiva della 3NF, infatti ogni relazione in BCNF è anche in 3NF, ma non viceversa.

Tornando allo schema di relazione LOTTI di Figura (a), con le sue quattro dipendenze funzionali, da DF1 a DF4. Si supponga di avere migliaia di lotti nella relazione, ma che i lotti vengano solo da due contee: Dekalb e Fulton. Si supponga inoltre che le dimensioni possibili dei lotti nella contea di Dekalb siano ristrette a 0.5, 0.6, 0.7, 0.8, 0.9 e 1.0 acri, mentre le dimensioni possibili dei lotti nella contea di Fulton si limitano a 1.1, 1.2, ..., 1.9, 2.0 acri. In questa situazione si avrà la dipendenza funzionale aggiuntiva FD5: AREA → NOME\_CONTEA. Se la si aggiunge alle altre dipendenze, lo schema di relazione LOTTI1A rimane in 3NF perché NOME\_CONTEA è un attributo primo.

Uno schema di relazione R è in BCNF se, ogni volta, che sussiste in R una dipendenza funzionale non-banale  $X \rightarrow A$ , X è una superchiave di R. La sola differenza tra le definizioni di BCNF e di 3NF è che la condizione (b) della 3NF, che consente ad A di essere primo, è assente dalla BCNF.

Nel nostro esempio DF5 viola la BCNF in LOTTI1A, perché AREA non è una superchiave di LOTTI1A. Si noti che DF5 soddisfa la 3NF in LOTTI1A perché NOME\_CONTEA è un attributo primo (condizione b), ma questa condizione non esiste nella definizione di BCNF. Si può decomporre LOTTI1A nelle due relazioni in BCNF LOTTI1AX e LOTTI1AY di Figura (a). Questa decomposizione perde la dipendenza funzionale DF2 perché i suoi attributi non coesistono più nella stessa relazione. In pratica, la maggior parte degli schemi di relazione in 3NF è anche in BCNF. Solo se in uno schema di relazione R sussiste  $X \rightarrow A$ , con X che non è una superchiave e A attributo primo, R sarà in 3NF ma non in BCNF. Lo schema di relazione R di Figura (b) illustra il caso generale di una relazione di questo tipo. Idealmente la progettazione di una base di dati relazione dovrebbe sforzarsi di raggiungere la BCNF o la 3NF per ogni schema di relazione. Il raggiungimento dello stato di normalizzazione a livello della sola 1NF o 2NF non è considerato sufficiente, dal momento che esse sono state storicamente sviluppate come punti di partenza verso la 3NF e la BCNF.



## 2. ALGORITMI PER LA PROGETTAZIONE DI BASE DI DATI RELAZIONALI

Gli **algoritmi di normalizzazione** cominciano tipicamente sintetizzando uno schema di relazione gigante, detto **relazione universale**, che comprende tutti gli attributi della base di dati. Si eseguono quindi decomposizioni ripetute, basandosi sulle dipendenze funzionali specificate dal progettista di basi di dati, finché non diventino non più possibili o non più desiderabili.

### DECOMPOSIZIONE DELLE RELAZIONI E INSUFFICIENZA DELLE FORME NORMALI:

Gli algoritmi di progettazione di basi di dati relazionali prendono in input un unico schema di **relazione universale**  $R=\{A_1, A_2, \dots, A_n\}$  contenente tutti gli attributi della base di dati. Si farà implicitamente l'**assunzione di relazione universale**, la quale stabilisce che ogni nome di attributo sia unico. L'insieme di dipendenze funzionali  $F$  che deve sussistere sugli attributi di  $R$  è specificato dai progettisti della base di dati ed è fornito agli algoritmi di progettazione. Basandosi sulle dipendenze funzionali, gli algoritmi decompongono lo schema di relazione universale  $R$  in un insieme di schemi di relazione  $D=\{R_1, R_2, \dots, R_m\}$  che diventerà lo schema della base di dati relazionale,  $D$  è detto **decomposizione** di  $R$ .

È necessario assicurarsi che ogni attributo presente in  $R$  sia anche presente in almeno uno schema di relazione  $R_i$  nella decomposizione, così che non ci siano attributi "persi", formalmente si ha:

$$\bigcup_{i=1}^m R_i = R.$$

Questa condizione è detta condizione di **conservazione degli attributi** di una decomposizione.

Un altro obiettivo è quello che ogni singola relazione  $R_i$  nella decomposizione  $D$  sia in BCNF (o in 3NF), ma questa condizione non è sufficiente a garantire una buona progettazione di basi di dati, ed occorre considerare la decomposizione nel complesso, oltre a esaminare le singole relazioni.

### DECOMPOSIZIONE E CONSERVAZIONE DELLE DIPENDENZE:

Sarebbe utile che ogni dipendenza funzionale  $X \rightarrow Y$  specificata in  $F$  apparisse direttamente in uno degli schemi di relazione  $R_i$  della decomposizione  $D$ , oppure potesse essere inferita dalle dipendenze presenti in qualche  $R_i$ . Informalmente, quella enunciata è la **condizione di conservazione delle dipendenze**. Se una delle dipendenze non è rappresentata in una singola relazione  $R_i$  della decomposizione, non è possibile imporre questo vincolo considerando una sola relazione, occorre piuttosto unire tramite join due o più relazioni della decomposizione e quindi verificare che nel risultato dell'operazione di join sussistano le dipendenze funzionali, ma questa è una procedura inefficiente e poco pratica.

Non è necessario che le dipendenze specificate in  $F$  si presentino esattamente nelle singole relazioni della decomposizione  $D$ , ma è sufficiente che l'unione delle dipendenze che sussistono sulle singole relazioni in  $D$  siano equivalenti a  $F$ .

Dato un insieme di dipendenze  $F$  su  $R$ , la **proiezione** di  $F$  su  $R_i$ , denotata con  $\pi_{R_i}(F)$ , dove  $R_i$  è un sottoinsieme di  $R$ , è l'insieme di dipendenze  $X \rightarrow Y$  di  $F^+$  tali che gli attributi di  $XUY$  siano tutti contenuti in  $R_i$ . Perciò la proiezione di  $F$  su ogni schema di relazione  $R_i$  della decomposizione  $D$  è l'insieme delle dipendenze funzionali in  $F^+$ , chiusura di  $F$ , tali che tutti i loro attributi di parte sinistra e di parte destra siano in  $R_i$ . Si dirà che una decomposizione  $D=\{R_1, R_2, \dots, R_m\}$  di  $R$  **conserva le dipendenze** rispetto a  $F$  se l'unione delle proiezioni di  $F$  su ogni  $R_i$  in  $D$  è equivalente a  $F$ , cioè:

$$((\pi_{R_1}(F)) \cup \dots \cup (\pi_{R_m}(F)))^+ = F^+.$$

Se una decomposizione non conserva le dipendenze, qualche dipendenza viene **persa** nella decomposizione. Per verificare se una dipendenza persa sussiste comunque, occorre effettuare il JOIN di due o più relazioni presenti nella decomposizione, fino a ottenere una relazione che presenta tutti gli attributi di parte sinistra e di parte destra nella dipendenza persa, e quindi verificare se nel risultato del JOIN sussiste la dipendenza, ma è un'operazione poco pratica.

Un esempio di decomposizione che non conserva le dipendenze è presentato in Figura, in cui la dipendenza funzionale  $DF2$  è persa quando  $LOTTI1A$  viene decomposta in  $\{LOTT1AX, LOTT1AY\}$ .

**Proposizione 1:** È sempre possibile trovare una decomposizione  $D$  che conserva le dipendenze rispetto a  $F$  e tale che ogni relazione  $R_i$  di  $D$  sia in 3NF.

L'algoritmo di seguito crea, a partire da un insieme di dipendenze funzionali  $F$ , una decomposizione  $D=\{R_1, R_2, \dots, R_m\}$  di una relazione universale  $R$ , che conserva le dipendenze e tale che ogni  $R_i$  di  $D$  sia in 3NF. Esso garantisce solo la proprietà di **conservazione delle dipendenze**, non la proprietà di join senza perdita. Il primo passo dell'algoritmo consiste nel trovare una copertura minimale  $G$  di  $F$ .

#### Algoritmo 1 (**algoritmo di sintesi relazionale**):

**Input:** Una relazione universale  $R$  e un insieme di dipendenze funzionali  $F$  sugli attributi di  $R$ .

1. Trovare una copertura minimale  $G$  di  $F$ .
2. Per ogni parte sinistra  $X$  di una dipendenza funzionale che appare in  $G$ , creare uno schema di relazione  $\{X \cup A_1 \cup A_2 \cup \dots \cup A_m\}$  in  $D$  dove  $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_m$  sono le sole dipendenze in  $G$  aventi  $X$  come parte sinistra.
3. Mettere in uno schema di relazione singolo tutti gli attributi rimanenti, per garantire la proprietà di **conservazione delle dipendenze**.

**Proposizione 1A:** Ogni schema di relazione creato dall'Algoritmo 1 è in 3NF.

L'algoritmo 1 conserva tutte le dipendenze presenti in  $G$ , perché ogni dipendenza compare in una delle relazioni  $R_i$  della decomposizione  $D$ . Dal momento che  $G$  è equivalente a  $F$ , tutte le dipendenze in  $F$  sono direttamente conservative nella decomposizione oppure sono derivabili da quelle presenti nelle relazioni risultanti, assicurando così la proprietà di conservazione delle dipendenze. L'Algoritmo 1 è detto **algoritmo di sintesi relazionale**, perché ogni schema di relazione  $R_i$  nella decomposizione è sintetizzato (costruito) a partire dall'insieme di dipendenze funzionale in  $G$  con la stessa parte sinistra  $X$ .

### COPERTURA MINIMA:

Un insieme  $F$  di dipendenze funzionali è **minimale** se soddisfa le seguenti condizioni:

1. Ogni dipendenza presente in  $F$  ha come parte destra un solo attributo.
2. Non è mai possibile sostituire una dipendenza  $X \rightarrow A$  di  $F$  con una dipendenza  $Y \rightarrow A$ , dove  $Y$  è un sottoinsieme proprio di  $X$ , e avere ancora un insieme di dipendenze equivalente a  $F$ ;
3. Non è mai possibile rimuovere una dipendenza da  $F$  e avere ancora un insieme di dipendenze equivalente a  $F$ .

Una **copertura minima** di un insieme F di dipendenze funzionali è un insieme minima di dipendenze  $F_{\min}$  equivalente a F. Purtroppo, per uno stesso insieme di dipendenze funzionali ci possono essere molte coperture minimali. Usando l'Algoritmo 2 si può sempre trovare almeno una copertura minima G per ogni insieme F di dipendenze.

### Algoritmo 2:

1. Porre  $G := F$ ;
2. Rimpiazzare ogni dipendenza funzionale  $X \rightarrow \{A_1, A_2, \dots, A_n\}$  in G, con n dipendenze funzionali  $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_n$ ;
3. Per ogni dipendenza funzionale  $X \rightarrow A$  in G  
per ogni attributo B che è un elemento di X  
se  $\{G - \{X \rightarrow A\}\} \cup \{(X - \{B\}) \rightarrow A\}$  è equivalente a G  
allora sostituire  $X \rightarrow A$  con  $(X - \{B\}) \rightarrow A$  in G;
4. Per ogni dipendenza funzionale rimanente  $X \rightarrow A$  in G  
se  $\{G - \{X \rightarrow A\}\}$  è equivalente a G  
allora rimuovere  $X \rightarrow A$  da G;

### DECOMPOSIZIONE E JOIN SENZA PERDITA:

Un'altra proprietà che una decomposizione D deve soddisfare è quella di join senza perdita o join non-additivo, che assicura che non vengano generate tuple spurie quando alle relazioni della decomposizione viene applicata un'operazione di JOIN NATURALE.

Formalmente, una decomposizione  $D = \{R_1, R_2, \dots, R_m\}$  di R soddisfa la **proprietà di join senza perdita (non-additivo)** rispetto all'insieme di dipendenze F di R se, per ogni stato di relazione r di R che soddisfa F, sussiste quanto segue, dove \* è il JOIN NATURALE di tutte le relazioni in D:

$$*(\pi_{R_1}(r), \dots, \pi_{R_m}(r)) = r$$

Se una decomposizione non soddisfa la proprietà di join senza perdita, è possibile che si presentino tuple spurie aggiuntive, dopo che sono state eseguire le operazioni di PROIEZIONE ( $\pi$ ) e di JOIN NATURALE (\*), queste tuple aggiuntive rappresentano un'informazione errata.

Per verificare se una data decomposizione D soddisfa la proprietà di join senza perdita, rispetto a un insieme F di dipendenze funzionali, si può usare.

### Algoritmo 1.2:

**Input:** una relazione universale R, una decomposizione  $D = \{R_1, R_2, \dots, R_m\}$  di R, e un insieme F di dipendenze funzionali.

1. Creare una matrice S con una riga i per ogni relazione  $R_i$  nella decomposizione D, e una colonna j per ogni attributo  $A_j$  in R;
2. Porre  $S(i,j) = B_{ij}$  per tutte le entrate della matrice; /\*ogni  $B_{ij}$  è un simbolo distinto associato agli indici (i,j) \*/
3. Per ogni riga i che rappresenta lo schema di relazione  $R_i$   
per ogni colonna j che rappresenta l'attributo  $A_j$   
se  $R_i$  include l'attributo  $A_j$  allora porre  $S(i,j) = a_j$  ;
4. Ripetere quanto segue finché l'esecuzione del ciclo non modifica più S  
per ogni dipendenza funzionale  $X \rightarrow Y$  in F  
per tutte le righe in S, che hanno gli stessi simboli nelle colonne corrispondenti agli attributi in X:  
rendere uguali i simboli in ogni colonna che corrispondono ad un attributo in Y, come segue:  
se una riga ha un simbolo 'a' nella colonna, porre le altre righe allo stesso simbolo 'a' nella colonna. Se non esiste in nessuna riga un simbolo 'a' per l'attributo, scegliere uno dei simboli 'b' che appaiono in una delle righe e porre le altre righe a quel simbolo 'b' nella colonna;
5. Se una riga contiene solo simboli 'a', allora la decomposizione ha la proprietà lossless join, altrimenti no.

Data una relazione R che è decomposta in un certo numero di relazioni  $R_1, R_2, \dots, R_m$ , l'Algoritmo 1.2 comincia col costruire uno stato di relazione r nella matrice S. La riga i in S rappresenta una tupla  $t_i$  (relativa alla relazione  $R_i$ ) che ha simboli "a" nelle colonne corrispondenti agli attributi di  $R_i$  e simboli "b" nelle altre colonne. L'algoritmo quindi trasforma le righe di questa matrice (durante il ciclo del passo 4) in modo tale che esse rappresentino tuple che soddisfano tutte le dipendenze funzionali in F. Alla fine del ciclo di applicazione delle dipendenze funzionali, ogni coppia di righe in S, che rappresentano due tuple di r, che si accordano sui valori degli attributi di parte sinistra X di una dipendenza funzionale  $X \rightarrow Y$  di F, si accorderanno anche sui valori degli attributi di parte destra Y. Si può dimostrare che se, dopo aver applicato il ciclo del passo 4, una qualsiasi riga di S termina con tutti simboli "a", allora la decomposizione D soddisfa la proprietà di join senza perdita rispetto a F. Se, dall'altro lato, nessuna riga ha alla fine tutti i simboli "a", allora D non soddisfa la proprietà di join senza perdita. In quest'ultimo caso lo stato di relazione r rappresentato da S alla fine dell'algoritmo sarà un esempio di stato di relazione r di R che soddisfa le dipendenze in F ma non soddisfa la condizione di join senza perdita, perciò, questa relazione serve come controesempio che prova che D non ha la proprietà di join senza perdita rispetto a F. Si noti che i simboli "a" e "b" alla fine dell'algoritmo non hanno alcun significato particolare. La Figura (a) mostra come si applica l'Algoritmo 1.2 alla decomposizione dello schema di relazione IMP\_PROG nei due schemi di relazione IMP\_PROG1 e IMP\_SEDI. Il ciclo al passo 4 dell'algoritmo non può cambiare nessun simbolo "b" in un simbolo "a", perciò la matrice S risultante non ha una riga di soli simboli "a" e quindi la decomposizione non gode della proprietà di join senza perdita. La Figura (b) mostra un'altra decomposizione IMP\_PROG in IMP, PROGETTO e LAVORA\_SU che gode della proprietà di join senza perdita, e la Figura (c) mostra come si può applicare l'algoritmo a questa decomposizione.

(a)	$R = (\text{SSN}, \text{NOME\_I}, \text{NUMERO\_P}, \text{NOME\_P}, \text{SEDE\_P}, \text{ORE})$ $R_1 = \text{IMP\_SEDI} = (\text{NOME\_I}, \text{SEDE\_P})$ $R_2 = \text{IMP\_PROG1} = (\text{SSN}, \text{NUMERO\_P}, \text{ORE}, \text{NOME\_P}, \text{SEDE\_P})$ $F = (\text{SSN} \rightarrow \text{NOME\_I}, \text{NUMERO\_P} \rightarrow (\text{NOME\_P}, \text{SEDE\_P}), (\text{SSN}, \text{NUMERO\_P}) \rightarrow \text{ORE})$	$D = \{R_1, R_2\}$
(nessun cambiamento alla matrice dopo aver applicato le dipendenze funzionali)		

(b)		IMP	PROGETTO	LAVORA_SU			
SSN	NOME_I	NUMERO_P	NOME_P	SEDE_P	SSN	NUMERO_P	ORE

(c)	$R = (\text{SSN}, \text{NOME\_I}, \text{NUMERO\_P}, \text{NOME\_P}, \text{SEDE\_P}, \text{ORE})$ $R_1 = \text{IMP} = (\text{SSN}, \text{NOME\_I})$ $R_2 = \text{PROG1} = (\text{NUMERO\_P}, \text{NOME\_P}, \text{SEDE\_P})$ $R_3 = \text{LAVORA\_SU} = (\text{SSN}, \text{NUMERO\_P}, \text{ORE})$ $F = (\text{SSN} \rightarrow \text{NOME\_I}, \text{NUMERO\_P} \rightarrow (\text{NOME\_P}, \text{SEDE\_P}), (\text{SSN}, \text{NUMERO\_P}) \rightarrow \text{ORE})$	$D = \{R_1, R_2, R_3\}$
(matrice originale S all'inizio dell'algoritmo)		

(matrice S dopo l'applicazione delle prime due dipendenze funzionali – l'ultima riga è costituita da tutti simboli "a", e pertanto ci fermiamo)							
SSN	NOME_I	NUMERO_P	NOME_P	SEDE_P	ORE		
$R_1$	$a_1$	$a_2$	$b_{13}$	$b_{14}$	$b_{15}$	$b_{16}$	
$R_2$	$b_{21}$	$b_{22}$	$a_3$	$a_4$	$a_5$	$b_{26}$	
$R_3$	$a_1$	$b_{32}$	$a_3$	$b_{34}$	$b_{35}$	$a_6$	

Una volta che una riga consista solo di simboli "a", si sa che la decomposizione gode della proprietà di join senza perdita, ed è possibile smettere di applicare le dipendenze funzionali (passo 4 dell'algoritmo) alla matrice S. L'Algoritmo 1.2 ci consente di controllare se una specifica decomposizione D soddisfa la proprietà di join senza perdita rispetto a un insieme F di dipendenze funzionali.

Esiste un algoritmo per decomporre uno schema di relazione universale  $R = \{A_1, A_2, \dots, A_n\}$  in  $D = \{R_1, R_2, \dots, R_m\}$ , in modo che ogni  $R_i$  sia in BCNF e la decomposizione D gode della proprietà di join senza perdita rispetto a F. Ma prima di descrivere l'algoritmo occorre presentare alcune proprietà generali delle decomposizioni con join senza perdita. La prima proprietà riguarda le **decomposizioni binarie**, decomposizioni di una relazione R in due relazioni. Essa fornisce un test più semplice da eseguire rispetto all'Algoritmo 1.2, ma è limitata alle sole decomposizioni binarie.

#### PROPRIETÀ LJ1 (Lossless Join):

Una decomposizione  $D=\{R_1, R_2\}$  di R soddisfa la proprietà di join senza perdita rispetto a un insieme di dipendenze funzionali F di R se e solo se:

- La DF  $((R_1 \cap R_2)) \rightarrow (R_1 - R_2)$  è in  $F^+$ , oppure
- La DF  $((R_1 \cap R_2)) \rightarrow (R_2 - R_1)$  è in  $F^+$

La seconda proprietà tratta l'esecuzione di decomposizioni successive e può essere applicata solo se si hanno due relazioni,  $R_1$  e  $R_2$ .

#### PROPRIETÀ LJ2:

Una decomposizione  $D=\{R_1, R_2, \dots, R_m\}$  di R gode della proprietà di join senza perdita rispetto a un insieme di dipendenze funzionali F su R, e se una decomposizione  $D_1=\{Q_1, Q_2, \dots, Q_k\}$  di  $R_i$  gode della proprietà di join senza perdita rispetto alla proiezione di F su  $R_i$ , allora la decomposizione

$D_2=\{R_1, R_2, \dots, R_{i-1}, Q_1, Q_2, \dots, Q_k, R_{i+1}, \dots, R_m\}$  di R gode della proprietà di join senza perdita rispetto a F.

La proprietà LJ2 sostiene che, se una decomposizione D soddisfa già la proprietà di join senza perdita e decomponiamo ulteriormente uno degli schemi di relazione  $R_i$  di D in un'altra decomposizione  $D_1$  che gode della proprietà di join senza perdita allora la sostituzione di  $R_i$  in D con  $D_1$  avrà come risultato una decomposizione che gode ancora della proprietà di join senza perdita, rispetto a F.

L'Algoritmo 1.3 utilizza le proprietà LJ1 e LJ2 per creare una decomposizione con join senza perdita di una relazione universale R,  $D = \{R_1, R_2, \dots, R_m\}$ , basata su un insieme di dipendenze funzionali F e tale che ogni  $R_i$  di D sia in BCNF.

#### Algoritmo 1.3:

**Input:** Una relazione universale R e un insieme di dipendenze funzionali F sugli attributi di R.

1. Porre  $D:=R$ ;
2. Finché c'è uno schema di relazione Q in D che non è in BCNF esegui:
  - si scelga uno schema di relazione Q in D che non sia in BCNF;
  - si trovi una dipendenza funzionale  $X \rightarrow Y$  che violi BCNF;
  - si sostituisca Q in D con due schemi di relazione  $(Q-Y)$  e  $(XUY)$ .

Ogni volta che si attraversa il ciclo presente nell'Algoritmo 1.3 si decompone uno schema di relazione Q che non è in BCNF in due schemi di relazione, secondo la LJ<sub>1</sub> e la LJ<sub>2</sub>, la decomposizione D ha la proprietà di join senza perdita. Al termine dell'algoritmo tutti gli schemi di relazione in D saranno in BCNF. Se si vuole che una decomposizione che goda della proprietà di join senza perdita e conservi le dipendenze, occorre accontentarsi di schemi di relazione in 3NF anziché in BCNF. Una semplice variazione dell'Algoritmo 1.1 (si veda l'Algoritmo 1.4), porta a una decomposizione D di R con le seguenti proprietà:

- conserva le dipendenze,
- gode della proprietà di join senza perdita,
- è tale che ogni schema di relazione risultante nella decomposizione sia in 3NF.

#### Algoritmo 1.4:

**Input:** Una relazione universale R e un insieme di dipendenze funzionali F sugli attributi di R.

1. Trovare una copertura minimale G per F;
2. Per ogni parte sinistra X di una FD in G, creare uno schema di relazione in D con attributi  $\{XU_{A_1}U_{A_2}\dots U_{A_m}\}$  dove  $X \rightarrow A_1, X \rightarrow A_2, \dots, X \rightarrow A_m$  sono le sole dipendenze in G aventi X come parte sinistra.
3. Se nessuno degli schemi di relazione in D contiene una chiave di R, creare un altro schema di relazione che contiene attributi che formano una chiave per R.

Si può dimostrare che la decomposizione formata dall'insieme di schemi di relazione costruiti dall'algoritmo precedente conserva le dipendenze e gode della proprietà di join senza perdita. Inoltre, ogni schema di relazione nella decomposizione è in 3NF. Questo algoritmo è un miglioramento dell'Algoritmo 1.1, dato che quest'ultimo garantiva solo la conservazione delle dipendenze.

Il passo 3 dell'Algoritmo 1.4 prevede l'individuazione di una chiave K di R. Per individuare una chiave K di R sulla base dell'insieme dato F di dipendenze funzionali può essere usato l'Algoritmo 1.4a. Si comincia ponendo K uguale a tutti gli attributi di R, quindi si rimuove un attributo alla volta e si verifica se i restanti attributi formano ancora una superchiave. Si noti che l'insieme di dipendenze funzionali usato per determinare una chiave nell'Algoritmo 1.4a può essere F o G, dal momento che essi sono equivalenti. Si noti, inoltre, che l'Algoritmo 1.4a determina solo una chiave fra le possibili chiavi candidate di R, la chiave fornita dipende dall'ordine con cui gli attributi sono rimossi da R al passo 2.

#### Algoritmo 1.4a:

1. Porre  $K:=R$ ;
2. Per ogni attributo A in K:
  - calcolare  $(K-A)^+$  rispetto all'insieme di FD;
  - se  $(K-A)^+$  contiene tutti gli attributi in R,
  - allora  $K=K-\{A\}$ ;

Non è sempre possibile trovare una decomposizione in schemi di relazione che conservi le dipendenze e che consenta a ogni schema di relazione della decomposizione di essere in BCNF (anziché in 3NF come nell'Algoritmo 1.4).

È importante sottolineare che la teoria delle decomposizioni con join senza perdita si basa sull'assunzione che non sono consentiti valori nulli per gli attributi di join.

## PROBLEMI CON VALORI NULLI E TUPLE DANGLING:

Quando si progetta uno schema di base di dati relazionale bisogna valutare attentamente i problemi associati alla presenza di valori nulli. Un problema si verifica quando alcune tuple presentano valori nulli per attributi che saranno usati per effettuare il JOIN di singole relazioni della decomposizione.

Si consideri la Figura (a), in cui sono presenti le due relazioni IMPIEGATO e DIPARTIMENTO. Le ultime due tuple impiegato (Berger e Benitez) rappresentano impiegati appena assunti che non sono stati ancora assegnati a un dipartimento. Si supponga ora di voler recuperare un elenco di valori di (NOME\_I, NOME\_D) per tutti gli impiegati. Se si esegue l'operazione di JOIN NATURALE su IMPIEGATO e DIPARTIMENTO in Figura (b), le due tuple suddette non appariranno nel risultato. Con l'operazione di JOIN ESTERNO si può affrontare questo problema. Si ricordi che, se si considera il JOIN ESTERNO SINISTRO di IMPIEGATO con DIPARTIMENTO, le tuple di IMPIEGATO che presentano un valore nullo per l'attributo di join appariranno comunque nel risultato, congiunte (joined) con una tupla "immaginaria" di DIPARTIMENTO che presenta valori nulli per tutti i suoi attributi, in Figura (c). In generale, ogni volta che viene progettato uno schema di base di dati relazionale in cui due o più relazioni sono collegate tramite chiavi esterne, deve essere dedicata particolare attenzione a ricercare potenziali valori nulli nelle chiavi esterne. Ciò può infatti causare un'inaspettata perdita d'informazione nelle interrogazioni che prevedono join su quella chiave esterna. Inoltre, se si presentano valori nulli in altri attributi, come STIPENDIO, il loro effetto su funzioni built-in(integrate) come SUM e AVERAGE deve essere attentamente valutato.

IMPIEGATO					
NOME_I	SSN	DATA_N	INDIRIZZO	NUM_D	
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	
Zelaya, Alicia J.	999887777	1966-07-19	3321 Castle, Spring, TX	4	
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellair, TX	4	
Nareyan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5	
English, Joyce A.	453453453	1972-07-31	5831 Rice, Houston, TX	5	Ricerca
Jabbar, Ahmad V.	987897987	1969-03-29	980 Dallas, Houston, TX	4	Amministrazione
Borg, James E.	888666555	1937-11-10	450 Stone, Houston, TX	1	Sede centrale
Berger, Andrea C.	999775555	1965-04-26	6530 Braes, Bellair, TX	null	
Benitez, Carlos M.	888664444	1963-01-09	7654 Beach, Houston, TX	null	

DIPARTIMENTO		
NOME_D	NUM_D	SSN_DIR_DIP
Ricerca	5	333445555
Amministrazione	4	987654321
Sede centrale	1	888666555

NOME_I	SSN	DATA_N	INDIRIZZO	NUM_D	NOME_D	SSN_DIR_DIP
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Ricerca	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Ricerca	333445555
Zelaya, Alicia J.	999887777	1966-07-19	3321 Castle, Spring, TX	4	Amministrazione	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellair, TX	4	Amministrazione	987654321
Nareyan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5	Ricerca	333445555
English, Joyce A.	453453453	1972-07-31	5831 Rice, Houston, TX	5	Ricerca	333445555
Jabbar, Ahmad V.	987897987	1969-03-29	980 Dallas, Houston, TX	4	Amministrazione	987654321
Borg, James E.	888666555	1937-11-10	450 Stone, Houston, TX	1	Sede centrale	888666555
Berger, Andrea C.	999775555	1965-04-26	6530 Braes, Bellair, TX	null		
Benitez, Carlos M.	888664444	1963-01-09	7654 Beach, Houston, TX	null		

NOME_I	SSN	DATA_N	INDIRIZZO	NUM_D	NOME_D	SSN_DIR_DIP
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Ricerca	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Ricerca	333445555
Zelaya, Alicia J.	999887777	1966-07-19	3321 Castle, Spring, TX	4	Amministrazione	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellair, TX	4	Amministrazione	987654321
Nareyan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5	Ricerca	333445555
English, Joyce A.	453453453	1972-07-31	5831 Rice, Houston, TX	5	Ricerca	333445555
Jabbar, Ahmad V.	987897987	1969-03-29	980 Dallas, Houston, TX	4	Amministrazione	987654321
Borg, James E.	888666555	1937-11-10	450 Stone, Houston, TX	1	Sede centrale	888666555
Berger, Andrea C.	999775555	1965-04-26	6530 Braes, Bellair, TX	null		
Benitez, Carlos M.	888664444	1963-01-09	7654 Beach, Houston, TX	null		

IMPIEGATO_1					
NOME_I	SSN	DATA_N	INDIRIZZO	NUM_D	NOME_D
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Ricerca
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Ricerca
Zelaya, Alicia J.	999887777	1966-07-19	3321 Castle, Spring, TX	4	Amministrazione
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellair, TX	4	Amministrazione
Nareyan, Ramesh K.	666884444	1962-09-15	975 Fire Oak, Humble, TX	5	Ricerca
English, Joyce A.	453453453	1972-07-31	5831 Rice, Houston, TX	5	Ricerca
Jabbar, Ahmad V.	987897987	1969-03-29	980 Dallas, Houston, TX	4	Amministrazione
Borg, James E.	888666555	1937-11-10	450 Stone, Houston, TX	1	Sede centrale
Berger, Andrea C.	999775555	1965-04-26	6530 Braes, Bellair, TX	null	
Benitez, Carlos M.	888664444	1963-01-09	7654 Beach, Houston, TX	null	

IMPIEGATO_2	
SSN	NUM_D
123456789	5
333445555	5
999887777	4
987654321	4
666884444	5
453453453	5
987897987	4
888666555	1
999775555	null
888664444	null

IMPIEGATO_3	
SSN	NUM_D
123456789	5
333445555	5
999887777	4
987654321	4
666884444	5
453453453	5
987897987	4
888666555	1

ALGORITMO	INPUT	OUTPUT	PROPRIETÀ/SCOPO	NOTE
Algoritmo 1.1	Insieme F di dipendenze funzionali	Un insieme di relazioni in 3NF	Conservazione delle dipendenze	Nessuna garanzia sulla verifica della proprietà di join senza perdita
Algoritmo 1.2	Una decomposizione D di R e un insieme F di dipendenze funzionali	Risultato booleano, sì o no per la proprietà di join non-additivo	Test per la decomposizione non-additiva	Esiste un test più semplice
Algoritmo 1.3	Insieme F di dipendenze funzionali	Un insieme di relazioni in BCNF	Decomposizione non-additiva	Nessuna garanzia della conservazione delle dipendenze
Algoritmo 1.4	Insieme F di dipendenze funzionali	Un insieme di relazione in 3NF	Decomposizione non-additiva e con conservazione delle dipendenze	Può capitare di non avere una BCNF, ma si ottiene la 3NF e tutte le proprietà auspicabili
Algoritmo 1.4a	Schema di relazione R con un insieme F di dipendenze funzionali	Chiave K di R	Per trovare una chiave K (cioè un sottoinsieme di R)	L'intera relazione R è sempre una superchiave di default

## DIPENDENZE MULTIVALORE:

In molti casi le relazioni hanno vincoli che non possono essere specificati sotto forma di dipendenze funzionali. Le dipendenze multivalore sono una conseguenza della prima forma normale (1NF), che impedisce a un attributo in una tupla di conoscere un insieme di valori. Se vi sono due o più attributi multivalore indipendenti nello stesso schema di relazione, si ha il problema di dover ripetere ogni valore di uno degli attributi per ciascun valore dell'altro attributo per mantenere consistente lo stato della relazione e per conservare l'indipendenza tra gli attributi coinvolti. Questo vincolo viene espresso mediante una **dipendenza multivaleure**.

Una tupla nella relazione IMP rappresenta il fatto che un impiegato il cui nome è NOME\_I lavora al progetto denominato NOME\_P e ha un dipendente il cui nome è NOME\_D. Un impiegato può lavorare a tanti progetti, può avere molti dipendenti, e i progetti e i dipendenti dell'impiegato sono indipendenti l'uno dall'altro. Per mantenere consistente lo stato della relazione si dovrà avere una tupla distinta per ogni possibile combinazione di dipendente e progetto di ciascun impiegato. Questo vincolo viene specificato tramite una dipendenza multivalore sulla relazione IMP. In modo informale può verificarsi una MVD ogni volta che due associazioni indipendenti A:B e A:C di tipo 1:N sono fuse nella stessa relazione.

IMP		PROGETTI_IMP		DIPENDENTI_IMP	
NOME_I	NOME_P	NOME_I	NOME_P	NOME_I	NOME_D
Smith	X	John		Smith	X
Smith	Y	Anna		Smith	Y
Smith	X	Anna		Brown	W
Smith	Y	John		Brown	X
Brown	W	Jim		Brown	Y
Brown	X	Jim		Brown	Z
Brown	Y	Jim			
Brown	Z	Jim			
Brown	W	Joan			
Brown	X	Joan			
Brown	Y	Joan			
Brown	Z	Joan			
Brown	W	Bob			
Brown	X	Bob			
Brown	Y	Bob			
Brown	Z	Bob			

## DEFINIZIONE FORMALE DI DIPENDENZA MULTIVALORE:

Una dipendenza multivalore  $X \twoheadrightarrow Y$  specificata sullo schema di relazione R, dove X e Y sono sottoinsiemi di R, specifica il seguente vincolo su qualsiasi stato di relazione r di R, se in r esistono due tuple  $t_1$  e  $t_2$  tali che  $t_1[X]=t_2[X]$ , allora in r devono esistere anche due tuple,  $t_3$  e  $t_4$  con le seguenti proprietà, dove viene usato Z per indicare  $(R - (X \cup Y))$ :

- $t_3[X]=t_4[X]=t_1[X]=t_2[X]$
- $t_3[Y]=t_1[Y]$  e  $t_4[Y]=t_2[Y]$
- $t_3[Z]=t_2[Z]$  e  $t_4[Z]=t_1[Z]$

Ogni volta che sussiste  $X \twoheadrightarrow Y$ , diciamo che X **multidetermina** Y. A causa della simmetria nella definizione, quando  $X \twoheadrightarrow Y$  sussiste in R, lo stesso fa  $X \twoheadrightarrow Z$ ; quindi  $X \twoheadrightarrow Y$  implica  $X \twoheadrightarrow Z$  e perciò talvolta viene scritto come  $X \twoheadrightarrow Y \mid Z$  ( $\mid$  è un OR).

La definizione formale specifica che, dato un particolare valore di X, l'insieme di valori di Y determinato da questo valore di X è determinato completamente solo da X e non dipende dai valori dei restanti attributi in Z di R. Ogni volta che esistono due tuple che hanno valori distinti di Y ma lo stesso valore di X, questi valori di Y devono essere ripetuti in tuple separate con ogni valore distinto di Z che si presenta con quello stesso valore di X. Questo corrisponde informalmente al fatto che Y è un attributo multivalore delle entità rappresentate dalle tuple in R.

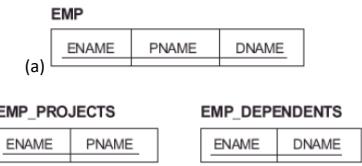
Nella relazione IMP valgono:

$\text{NOME\_I} \twoheadrightarrow \text{NOME\_P}$  e  $\text{NOME\_I} \twoheadrightarrow \text{NOME\_D}$  (oppure  $\text{NOME\_I} \twoheadrightarrow \text{NOME\_P} \mid \text{NOME\_D}$ ).

Una MVD  $X \twoheadrightarrow Y$  in R è detta **MVD banale (trivial)** se:

- Y è un sottoinsieme di X, oppure
- $X \cup Y = R$

In PROGETTI\_IMP,  $\text{NOME\_I} \twoheadrightarrow \text{NOME\_P}$  è una MVD banale.



## REGOLE DI INFERNZA PER FD E MVD:

Come per le dipendenze funzionali, esistono regole di inferenza per le dipendenze multivalore, tuttavia è meglio fornire un contesto unificato che includa le DF e le MVD così che ambedue i tipi di vincolo possano essere considerati insieme. Le regole di inferenza seguenti, da RI1 a RI8, formano un insieme completo e corretto per inferire le dipendenze funzionali e multivalore da un insieme di dipendenze dato. Si supponga che tutti gli attributi siano inclusi in uno schema di relazione "universale"  $R = \{A_1, A_2, \dots, A_n\}$  e che  $X, Y, Z$  e  $W$  siano sottoinsiemi di  $R$ .

- **RI1 (regola riflessiva per le DF):**  $\text{se } X \supseteq Y, \text{ allora } X \rightarrow Y$
- **RI2 (regola di arricchimento per le DF):**  $\{X \rightarrow Y\} \mid= XZ \rightarrow YZ$
- **RI3 (regola transitiva per le DF):**  $\{X \rightarrow Y, Y \rightarrow Z\} \mid= X \rightarrow Z$
- **RI4 (regola di completamento per le MVD):**  $\{X \twoheadrightarrow Y\} \mid= X \twoheadrightarrow (R - (X \cup Y))$
- **RI5 (regola di arricchimento per le MVD):**  $\text{Se } X \twoheadrightarrow Y \text{ e } W \supseteq Z, \text{ allora } WX \twoheadrightarrow YZ$
- **RI6 (regola transitiva per le MVD):**  $\{X \twoheadrightarrow Y, Y \twoheadrightarrow Z\} \mid= X \twoheadrightarrow (Z - Y)$
- **RI7 (regola di replicazione per DF in MVD):**  $\{X \rightarrow Y\} \mid= X \rightarrow Y$
- **RI8 (regola di unione per le DF e le MVD):**  $\text{Se } X \twoheadrightarrow Y \text{ ed esiste } W \text{ con le proprietà che (a) } W \cap Y \text{ è vuota, (b) } W \rightarrow Z \text{ e (c) } Y \supseteq Z, \text{ allora } X \rightarrow Z$

Le regole di inferenza di Armstrong per le DF sono quelle da RI1 a RI3, mentre le regole da RI4 a RI6 sono regole di inferenza che riguardano solo le MVD. Le RI7 e RI8 sono correlate sia alle DF sia alle MVD. In particolare, la RI7 afferma che una dipendenza funzionale è un caso speciale di una dipendenza multivalore, cioè ogni DF è anche una MVD perché soddisfa la definizione formale di una MVD. Questa equivalenza sottende che: una DF  $X \rightarrow Y$  è una MVD  $X \twoheadrightarrow Y$  con la limitazione implicita che al massimo un valore di Y è associato a ogni valore di X. Dato un insieme F di dipendenze funzionali e multivalore specificato su  $R = \{A_1, A_2, \dots, A_n\}$ , si possono usare le regole da RI1 a RI8 per inferire l'insieme (completo) di tutte le dipendenze (funzionali o multivalore)  $F^+$  che sussisterà in ogni stato di relazione r di R che soddisfa F. Viene chiamato  $F^+$  la **chiusura** di F.

## QUARTA FORMA NORMALE:

La **quarta forma normale 4NF** viene violata quando una relazione ha dipendenze multivalore non volute, e quindi può essere usata per identificare e decomporre tali relazioni. Uno schema di relazione R è in 4NF rispetto a un insieme di dipendenze F (che include le dipendenze funzionali e multivalore) se, per ogni dipendenza multivalore non banale  $X \twoheadrightarrow Y$  in  $F^+$ , X è una superchiave di R.

La relazione IMP della Figura (a) non è in 4NF perché nelle MVD non banali  $\text{NOME\_I} \twoheadrightarrow \text{NOME\_P}$  e  $\text{NOME\_I} \twoheadrightarrow \text{NOME\_D}$ ,  $\text{NOME\_I}$  non è una superchiave di IMP. Si decomponе allora IMP in PROGETTI\_IMP e DIPENDENTI\_IMP, come mostrato nella Figura (b). Sia PROGETTI\_IMP sia DIPENDENTI\_IMP sono in 4NF, perché le MVD:  $\text{NOME\_I} \twoheadrightarrow \text{NOME\_P}$  in PROGETTI\_IMP e  $\text{NOME\_I} \twoheadrightarrow \text{NOME\_D}$  in DIPENDENTI\_IMP sono MVD banali. In PROGETTI\_IMP o DIPENDENTI\_IMP non è contenuta alcun'altra MVD non banale. In questi schemi di relazione non sussiste neppure alcuna DF.

**NOTA sulle MVD non-banali:** Relazioni contenenti MVD non-banali tendono ad essere relazioni "tutta chiave" (la chiave è formata da tutti gli attributi).

## DECOMPOSIZIONE DI JOIN NON-ADDITIONAL IN SCHEMI IN 4NF:

Ogni volta che si decomponе uno schema di relazione R in  $R_1 = (X \cup Y)$  e  $R_2 = (R - Y)$  sulla base di una MVD  $X \twoheadrightarrow Y$  che sussiste in R, la decomposizionе gode della proprietà di join non-additivo. Si può dimostrare che questa è una condizione necessaria e sufficiente per decomporre uno schema in due schemi che hanno la proprietà di join non-additivo, come specificato dalla proprietà LJ1' che è un'ulteriore generalizzazione della proprietà LJ1. La proprietà LJ1 è relativa solo alle DF, mentre LJ1' gestisce sia le DF sia le MVD (da ricordare che una DF è anche una MVD).

### PROPRIETÀ LJ1':

Gli schemi di relazione  $R_1$  e  $R_2$  costituiscono una decomposizionе non-additiva di R rispetto a un insieme F di dipendenze funzionali e multivalore se e solo se:

$$(R_1 \cap R_2) \twoheadrightarrow (R_1 - R_2)$$

oppure, per simmetria, se e solo se:

$$(R_1 \cap R_2) \twoheadrightarrow (R_2 - R_1)$$

Con una piccola modifica dell'Algoritmo 1.3 si ottiene l'Algoritmo 1.5, che crea una decomposizionе non-additiva che produce schemi di relazione che sono in 4NF (anziché in BCNF). Come l'Algoritmo 1.3, l'Algoritmo 1.5 non produce necessariamente una decomposizionе che conserva le DF.

### Algoritmo 1.5:

**Input:** Una relazione universale R e un insieme di dipendenze funzionali e multivalore F.

1. Si ponga  $D := \{R\}$ ;
2. Fintantoché vi è uno schema di relazione Q in D che non è in 4NF;
  - si scelga uno schema di relazione Q in D che non è in 4NF;
    - si trovi una MVD non banale  $X \twoheadrightarrow Y$  in Q che viola la 4NF;
    - si sostituisca Q in D con due schemi di relazione  $(Q - Y)$  e  $(X \cup Y)$ ;

### 3. APPLICAZIONI DI DESIGN E TUNING DI DATABASE

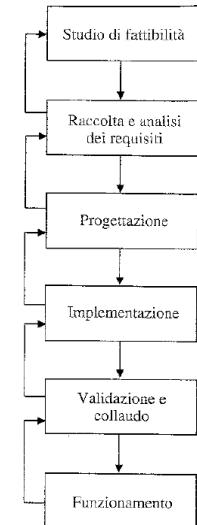
L'attività di design di un database è un processo sistematico che segue una metodologia ben definita ed è spesso legata al tool di design fornito.

Un'accurata gestione dei dati è fondamentale in quanto sono una risorsa dell'azienda e una loro corretta gestione facilita e rende più efficiente il lavoro. Le **caratteristiche chiave** si individuano nell'integrazione dei dati tra diverse applicazioni in un singolo DB, facilità di sviluppo di nuove applicazioni usando linguaggi ad alto livello tipo SQL e la possibilità da parte dei manager di interrogare i dati ed avere risultati aggiornati.

#### MACRO-CICLO DI VITA DI UN SISTEMA INFORMATIVO:

Un database system è solo una parte di un **sistema informativo**. Esso è composto da dati, DBMS, hardware, media di memorizzazione, applicativi che interagiscono con i dati, il personale che gestisce o usa il sistema, gli applicativi che gestiscono l'aggiornamento dei dati, i programmatore che sviluppano tali applicativi. La progettazione di una base di dati costituisce solo una delle componenti del processo di sviluppo di un sistema informativo complesso e va quindi inquadrata in un contesto più ampio. Il ciclo di vita di un sistema informativo è detto **macro-ciclo di vita**, ed ha 6 fasi:

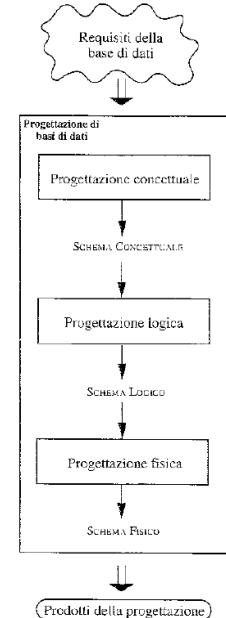
1. **Analisi di fattibilità**: dove si analizzano le potenziali aree di applicazione, si effettuano degli studi di costi/benefici, si determina la complessità di dati e processi, e si impostano le priorità tra le applicazioni.
2. **Raccolta ed analisi dei requisiti**: consiste nell'individuazione e nello studio delle proprietà e delle funzionalità che il sistema informativo dovrà avere. Questa fase richiede un'interazione con gli utenti del sistema e produce una descrizione completa, ma generalmente informale, dei dati coinvolti e delle operazioni su di essi.
3. **Progettazione**: si divide generalmente in progettazione dei dati e delle applicazioni. Nella prima si individua la struttura e l'organizzazione che i dati dovranno avere, nell'altra si definiscono le caratteristiche dei programmi applicativi.
4. **Implementazione**: consiste nella realizzazione del sistema informativo secondo la struttura e le caratteristiche definite nella fase di progettazione. Viene costruito e popolato il database e viene prodotto il codice dei programmi, testando le transazioni.
5. **Validazione e Testing**: serve a certificare il corretto funzionamento e la qualità del sistema informativo. La sperimentazione deve prendere, per quanto possibile, tutte le condizioni operative.
6. **Rilascio e Manutenzione**: in questa fase il sistema informativo diventa operativo ed esegue i compiti per i quali era stato originariamente progettato. Il rilascio può essere preceduto da una fase di addestramento del personale al nuovo sistema. Se emergono nuove funzionalità da implementare, si ripetono i passi precedenti, per includerle nel sistema. Se non si verificano malfunzionamenti o revisioni delle funzionalità del sistema, questa attività richiede solo operazioni di gestione e manutenzione.



#### METODOLOGIA DI PROGETTAZIONE:

Nell'ambito delle basi di dati, si è consolidata negli anni una metodologia di progetto che soddisfa alcune proprietà che una buona base di dati dovrebbe possedere. Tale metodologia è articolata in tre fasi principali da effettuare in cascata, separando in maniera netta le decisioni relativa a "cosa" rappresentare in una base di dati (prima fase), da quelle relative a "come" farlo (seconda e terza fase).

- **Progettazione concettuale**: Il suo scopo è quello di rappresentare le specifiche informali della realtà d'interesse in termini di una descrizione formale e completa, ma indipendente dai criteri di rappresentazione utilizzati nei sistemi di gestione di basi di dati. Il prodotto di questa fase viene chiamato **schema concettuale** e fa riferimento a un **modello concettuale** dei dati. I modelli concettuali consentono di descrivere l'organizzazione dei dati a un alto livello di astrazione, senza tenere conto degli aspetti implementativi. In questa fase il progettista deve cercare di rappresentare il contenuto informativo della base di dati, senza preoccuparsi né della modalità con le quali queste informazioni verranno codificate in un sistema reale, né dell'efficienza dei programmi che faranno uso di queste informazioni.
- **Progettazione logica**: Consiste nella traduzione dello schema concettuale definito nella fase precedente, in termini del modello di rappresentazione dei dati adottato dal sistema di gestione di base di dati a disposizione. Il prodotto in questa fase viene denominato **schema logico** della base di dati e fa riferimento a un modello logico dei dati. Come noto, un **modello logico** permette di descrivere i dati secondo una rappresentazione ancora indipendente da dettagli fisici, ma concreta perché disponibile nei sistemi di gestione di base di dati. In questa fase, le scelte progettuali si basano su criteri di ottimizzazione delle operazioni da effettuare sui dati. Si fa comunque uso anche di tecniche di verifica della qualità dello schema logico ottenuto e nel caso del modello relazionale dei dati, la tecnica comunemente utilizzata è la normalizzazione.
- **Progettazione fisica**: In questa fase lo schema logico viene completato con la specifica dei parametri fisici di memorizzazione dei dati (organizzazione dei file e degli indici). Il prodotto di questa fase viene denominato **schema fisico** e fa riferimento a un **modello fisico** dei dati. Tale modello dipende dallo specifico sistema di gestione di basi di dati scelto e si basa sui criteri di organizzazione fisica dei dati in quel sistema.



#### MICRO-CICLO DI VITA DI UN DATABASE SYSTEM:

Il ciclo di vita di un database system è detto **micro-ciclo di vita** e includono le seguenti 8 fasi:

1. **Definizione del sistema**: dove viene definito l'ambito del sistema di base di dati, i suoi utenti e le funzionalità. Inoltre, si identificano le interfacce per le categorie di utenti, i vincoli sui tempi di risposta ed i requisiti hardware.
2. **Progettazione della base di dati**: si realizza la progettazione logica e fisica per il DBMS scelto.
3. **Implementazione della base di dati**: si specificano le definizioni concettuali, esterne ed interne, si creano i file del database vuoti e si implementa dell'eventuale software applicativo di supporto.
4. **Caricamento / conversione dei dati**: si popola il database, o inserendo direttamente i dati o convertendo file esistenti nel nuovo formato.
5. **Conversione delle applicazioni**: si convertono le vecchie applicazioni software al nuovo sistema.
6. **Test e validazione**: si effettuano test e validazione del nuovo sistema.
7. **Operation**: il sistema di base di dati e le sue applicazioni diventano operativi. In genere, per un certo tempo vengono utilizzati in parallelo il vecchio ed il nuovo sistema.
8. **Controllo e manutenzione**: il sistema è sottoposto a costante monitoraggio. Eventualmente si possono gestire aggiunte nei dati o nelle funzionalità presenti.

## PROCESSO DI PROGETTAZIONE DI UN DATABASE:

La fase più interessante nel **micro-ciclo di vita** è quella di progettazione. Questo processo serve a progettare la struttura logica e fisica di uno o più database, per soddisfare i requisiti degli utenti di un'organizzazione su un determinato insieme di operazioni. Gli scopi della fase di progettazione sono:

1. Soddisfare i requisiti sui dati che interessano gli utenti e a cui accedono le applicazioni.
2. Fornire una strutturazione delle informazioni naturale e di facile comprensione.
3. Soddisfare i requisiti di elaborazione e di prestazioni (tempo di risposta, spazio di memorizzazione, ecc...).

È difficile raggiungere tutti gli scopi, in quanto alcuni sono in contrasto tra loro ed un modello più comprensibile può comportare un costo in termini di prestazioni. Il processo di progettazione è costituito da due attività parallele:

- Progettazione di strutture e contenuti dei dati (*progettisti di basi di dati*).
- Progettazione delle applicazioni che usano la base di dati (*ingegneri del software*).

Le metodologie di progettazione di database si sono focalizzate sulla prima attività (**approccio data-driven** vs **progettazione process-driven**).

## FASI DELLA PORGETTAZIONE DI UN DATABASE:

La progettazione di un database è composta da 6 fasi principali, non sono eseguite in sequenza, in quanto spesso delle modifiche ad un livello devono essere propagate al livello superiore, creando dei **cicli di feedback**.

### FASE 1: Raccolta ed analisi dei requisiti:

Per progettare un database è necessario conoscere ed analizzare le aspettative degli utenti nel modo più dettagliato possibile. Per specificare i requisiti, è necessario individuare tutte le componenti che interagiranno col database. Tipicamente queste sono gli utenti e le applicazioni.

La fase 1 comprende di quattro attività:

1. Identificare le principali aree di applicazione, gli utenti che useranno il database e quelli il cui lavoro sarà influenzato dal database stesso. Individuare in ogni gruppo di persone un rappresentante per poter portare avanti la raccolta delle specifiche.
2. Analizzare la documentazione già esistente riguardante le applicazioni. Esaminare anche altri tipi di documentazione (form, report, grafici aziendali, ecc...) che in qualche modo possono influenzare i requisiti.
3. Esaminare il contesto operativo e l'utilizzo pianificato delle informazioni. Questa attività include l'analisi delle transazioni, dei flussi di informazioni e la specifica dei dati di input e output per ogni transazione.
4. Intervistare gli utenti finali per determinare priorità ed importanza previste per le varie applicazioni.

Successivamente i requisiti andranno sviluppati in quanto, spesso all'inizio i requisiti sono informali, incompleti, inconsistenti e parzialmente incorretti. È quindi necessario molto lavoro per trasformarli in specifiche da fornire a programmatore e tester. Poiché i requisiti si riferiscono ad un sistema non ancora esistente, inevitabilmente vengono spesso modificati. Per trasformare i requisiti in una forma strutturata, si utilizzano delle **tecniche di specifica dei requisiti**. Le principali tecniche sono: Analisi orientata agli oggetti (OOA), Diagramma di flusso dei dati (DFD), Raffinamento degli obiettivi dell'applicazione. Esistono anche altre tecniche che producono una specifica formale dei requisiti che consentono verifiche matematiche di consistenza (analisi simboli "what-if"), sebbene più difficili da usare, queste tecniche sono fondamentali per applicazioni *mission-critical*.

È possibile utilizzare dei tool **CASE (Computer Aided Software Engineering)** per controllare la completezza e la consistenza delle specifiche, detti *Upper Case tool*. La **fase di raccolta ed analisi dei requisiti** richiede un grande sforzo in termini di tempo, ma è cruciale per il successo del sistema informativo. Un errore dovuto a requisiti errati può essere estremamente costoso poiché può comportare la re-implementazione di buona parte del lavoro, il sistema potrebbe non rispondere alle richieste del cliente e non essere usato affatto.

### FASE 2: Progettazione dello schema concettuale:

La seconda fase del progetto di un database consta di 2 attività parallele:

1. **Progettazione dello schema concettuale**, dove si esaminano i requisiti per produrre lo schema concettuale del database.
2. **Progettazione di transazioni ed applicazioni**, dove si esaminano le applicazioni del database per produrre specifiche di alto livello delle applicazioni.

## PROGETTAZIONE DELLO SCHEMA CONCETTUALE:

Lo schema concettuale deve essere indipendente dal DBMS per i seguenti motivi:

- Lo scopo dello schema concettuale è fornire una comprensione completa di struttura, semantica, relazioni e vincoli del database. Legarsi ad un DBMS porterebbe a delle restrizioni che influenzerebbero lo schema.
- Lo schema concettuale è una descrizione stabile dei contenuti del database. La scelta del DBMS e le decisioni di progettazione successive possono cambiare senza che questo debba essere modificato.
- L'utilizzo di una data model di alto livello è più espressivo e generale dei modelli di dati utilizzati dai singoli DBMS, ed è quindi più comprensibile per gli utenti.
- La descrizione diagrammatica dello schema concettuale può essere usata molto efficacemente come mezzo di comunicazione tra gli utenti del database, i progettisti e gli analisti. I modelli di dati dei DBMS, essendo di livello più basso, spesso mancano di un tale livello di espressività.

Un **data model di alto livello** deve godere delle seguenti proprietà:

1. **Espressività**: deve permettere una facile distinzione tra tipi di dati, relazioni e vincoli.
2. **Semplicità e comprensibilità**: deve essere semplice, per consentire a utenti non esperti di comprendere ed utilizzare i suoi concetti.
3. **Minimalità**: dovrebbe avere pochi concetti di base, non sovrapponibili.
4. **Rappresentazione diagrammatica**: dovrebbe avere una notazione diagrammatica per rappresentare schemi concettuali di facile comprensione.
5. **Formalità**: deve fornire dei formalismi per specificare in modo non ambiguo i dati.

## APPROCCI ALLA PROGETTAZIONE DI UNO SCHEMA CONCETTUALE:

Per progettare uno schema concettuale, è necessario individuare le componenti di base di uno schema. Queste sono entità, relazioni, attributi, vincoli di cardinalità e partecipazione, chiavi, gerarchie di specializzazione/generalizzazione, entità deboli.

Esistono due approcci alla progettazione di uno schema concettuale:

1. **Progetto di schema centralizzato** (o **one-shot**): tutti i requisiti di applicazioni differenti e diversi gruppi di utenti sono fusi in un singolo insieme di requisiti prima di iniziare la progettazione. Al Database Administrator spetta il compito di decidere come unire i requisiti e di progettare l'intero schema. Una volta progettato l'intero schema concettuale si progettano gli schemi esterni.
2. **Integrazione di viste**: i requisiti non vengono fusi. Inizialmente si progetta uno schema (o vista) per ogni gruppo di utenti. Successivamente, in una fase di integrazione, le viste sono fuse in un unico schema concettuale globale.

## STRATEGIE PER LA PROGETTAZIONE DI SCHEMI:

Esistono differenti strategie per creare uno schema concettuale partendo dai requisiti:

- **Strategia Top-Down:** si parte da uno schema con astrazioni di alto livello, che vengono successivamente raffinate. Vengono specificate le entità ad alto livello, e quando si vanno a specificare gli attributi, le entità si spezzano in entità di livello inferiore e si introducono relazioni (ad esempio, specializzazione di un tipo di entità in sottoclassi).
- **Strategia Bottom-Up:** si parte da uno schema contenente astrazioni di base, che vengono via via combinate e messe in relazione tra loro (ad esempio, generalizzare tipi di entità in superclassi di alto livello).
- **Strategia Inside-Out:** È una specializzazione del bottom-up, in cui l'attenzione è focalizzata su un nucleo centrale di operazioni. La modellazione, quindi, si allarga verso l'esterno, inglobando nuovi concetti collegati a quelli già considerati.
- **Strategia Mixed:** Non prevede una strategia uniforme per tutto il progetto, ma dà la disponibilità di usare metodi diversi per le varie componenti, che vengono combinate successivamente.

## INTEGRAZIONE DI SCHEMI:

Si analizza una **metodologia per l'integrazione di schemi in uno schema globale di database** (necessaria per grossi database). L'integrazione di schemi può essere divisa in quattro sotto-compiti:

1. **Identificazione di corrispondenze e conflitti tra gli schemi:** in questa fase è possibile individuare quattro possibili tipi di conflitti:
  - a. **Conflitti di nome:** che possono essere di due tipi:
    - I. **Sinonimi:** due schemi utilizzano termini diversi per identificare lo stesso concetto (ad esempio, in due schemi differenti esistono il tipo di entità CUSTOMER e CLIENT che descrivono lo stesso concetto)
    - II. **Omonimi:** due schemi utilizzano lo stesso termine per identificare concetti diversi.
  - b. **Conflitti di tipo:** lo stesso concetto può essere espresso in schemi diversi con costrutti di modellazione diversi (ad esempio, un attributo in uno schema e un tipo di entità in un altro schema).
  - c. **Conflitti di dominio:** un attributo può avere domini differenti in schemi diversi (ad esempio, intero in uno schema e carattere in un altro).  
Conflitti di unità di misura (un attributo è descritto in metri in uno schema e in chilometri in un altro).
  - d. **Conflitti tra vincoli:** due schemi possono imporre vincoli differenti (ad esempio, la chiave di un tipo di entità può risultare diversa in due schemi differenti).
2. **Modifica delle viste per renderle conformi.**
3. **Fusione delle viste:** si crea uno schema globale fondendo tutte le viste. I concetti corrispondenti devono essere rappresentati solo una volta. Non è un compito automatizzabile, e richiede una notevole esperienza.
4. **Ristrutturazione:** lo schema può essere analizzato per eliminare ridondanze.

## PROGETTAZIONE DI TRANSAZIONI:

Lo scopo di questa fase è di progettare le transazioni (o applicazioni) del database in modo indipendente dal DBMS. Specificare le caratteristiche funzionali delle transazioni assicura che lo schema del database includerà le informazioni che esse richiedono. Difficilmente nella fase di progettazione si ha una visione completa di tutte le transazioni da implementare, molte saranno identificate e realizzate al termine dell'implementazione del database. Una tecnica usata per specificare le transazioni prevede l'identificazione di: Input, Output e Comportamento funzionale.

Definendo i parametri di input/output ed il flusso di controllo funzionale interno, è possibile specificare una transazione in un modo concettuale indipendente dal sistema. È possibile raggruppare le transazioni in tre categorie:

1. **Transazioni di retrieval:** usate per recuperare dati da visualizzare o su schermo o su report.
2. **Transazioni di update:** usate per inserire o modificare i dati.
3. **Transazioni miste:** usate per applicazioni più complesse che richiedono sia retrieval che update.

Il design di transazioni è considerato parte dell'Ingegneria del software.

## FASE 3: Scelta del DBMS:

La scelta del DBMS è influenzata da tre fattori:

- **Fattori tecnici:** Data model del DBMS, strutture di memorizzazione offerte, disponibilità di interfacce per utenti e programmatore, disponibilità di tool di sviluppo, linguaggio di query supportato, possibilità di interagire con altri DBMS.
- **Fattori economici:** Costi di acquisto del software, manutenzione, acquisizione nuovo hardware, creazione e conversione database e training.
- **Fattori organizzativi/aziendali:** Adozione di una "filosofia" in tutta l'azienda, accettare un DBMS implica accettarne il data model, il paradigma di programmazione, i tool di sviluppo. Familiarità del personale con il sistema, scegliere un sistema già conosciuto diminuisce i costi di training.

## FASE 4: Mapping del data model (design logico):

La creazione di schemi concettuali ed esterni nel data model specifico del DBMS selezionato avviene in due passi:

1. **Mapping indipendente dal sistema:** il mapping non considera nessuna caratteristica specifica del DBMS (traduzione da ER a Relazionale);
2. **Mapping per lo specifico DBMS:** si modifica lo schema ottenuto al passo 1 per adeguarlo alle caratteristiche ed ai vincoli dello specifico DBMS.

Il risultato sono le istruzioni DDL (Data Definition Language) per specificare gli schemi concettuali ed esterni del database.

## FASE 5: Progettazione dello schema fisico:

Comprende la definizione di strutture di memorizzazione e di access path per i file del database. Esistono 3 parametri guida:

1. **Tempo di risposta:** è il tempo medio trascorso dalla sottomissione di una transazione alla ricezione dei risultati.
2. **Utilizzazione di spazio:** è lo spazio totale utilizzato dal database, compresi gli indici.
3. **Throughput delle transazioni:** è il numero medio di transazioni completate al minuto (parametro critico per sistemi transazionali quali banche).

A causa dell'importanza di tali parametri, spesso nei requisiti si includono i limiti per il caso medio e per il caso pessimo. Le prestazioni dipendono dalla taglia dei record e dal numero di record nei file, essi sono parametri che vanno stimati. Spesso si utilizzano dei prototipi del sistema per valutarne le prestazioni effettive. Devono essere considerati gli attributi usati per accedere ai record e gli indici primari e secondari necessari.

## FASE 6: Implementazione e tuning del database system:

L'implementazione del database è a carico del Database Administrator e dei Database Designer. Le istruzioni DDL sono compilate ed eseguite per creare gli schemi ed i file vuoti. Il database viene quindi popolato coi dati, se essi sono già esistenti in un altro formato, può essere necessario implementare delle routine di conversione. I programmatore implementano le transazioni utilizzando comandi Data Manipulation Language del DBMS.

## TUNING DEL DATABASE:

La maggior parte dei DBMS includono tool di monitoraggio. In base ai dati collezionati, è possibile modificare tabelle, access path, query. Alcune query o transazioni possono essere riscritte per migliorare le prestazioni. Il processo di tuning continua durante tutta l'operatività del database system.

## LA PROGETTAZIONE FISICA NEI DATABASE RELAZIONALI:

La progettazione fisica di un database si propone non solo di fornire delle strutture dati appropriate, ma anche di garantire delle buone performance del database system. Per effettuare una buona progettazione, è necessario conoscere le query, le transazioni e gli applicativi eseguiti sul database, analizzarne la frequenza di esecuzione, e gli eventuali vincoli posti nei requisiti. Fattori che influenzano il design fisico:

- **Analisi di query.** Per ogni query si deve specificare:

1. I file a cui accede la query
2. Gli attributi su cui sono specificate le condizioni di selezione (Attributi candidati per la definizione di access path)
3. Gli attributi su cui sono specificate le condizioni di join (Attributi candidati per la definizione di access path)
4. Gli attributi di cui la query recupera i valori

- **Analisi di transazioni.** Per ogni transazione di update si specificano:

1. I file aggiornati dalla transazione
2. Il tipo di operazioni per ogni file (lettura, modifica, cancellazione)
3. Gli attributi su cui sono specificate condizioni di cancellazione o di modifica (Attributi candidati per la definizione di access path)
4. Gli attributi i cui valori sono modificati dalla transazione (Questi attributi NON dovrebbero essere utilizzati in access path)

- **Analisi sulla frequenza di esecuzione di query e transazioni,** si determina la frequenza con cui le query sono eseguite, in generale, vale la regola "80-20" dove l'80% del processing è effettuato dal 20% delle query.

- **Analisi sui vincoli temporali,** alcune query e transazioni possono avere dei vincoli temporali che impongono priorità nella definizione di access path (ad esempio, una transazione deve terminare entro 5s dalla sua invocazione). Questo vincolo fornisce una priorità maggiore a degli attributi che dovranno essere usati per access path.

- **Analisi sulla frequenza di operazioni di update,** il numero di access path su file aggiornati frequentemente deve essere minimizzato, in quanto produce un overhead per aggiornare anche gli access path.

- **Analisi dei vincoli di univocità degli attributi,** andrebbero specificati degli access path per ogni chiave candidata.

## DECISIONI PROGETTUALI SUGLI INDICI:

Sebbene le performance di query migliorino fortemente in presenza di indici o schemi hash, le operazioni di inserimento, modifica e cancellazione sono rallentate dagli indici. Le decisioni sulle indicizzazioni ricadono in una delle cinque categorie seguenti:

1. **Quando indicizzare un attributo,** un attributo deve essere indicizzato se è chiave o se è utilizzato in una condizione di select (uguaglianza o range di valori) o join da una query.
2. **Quale attributo indicizzare,** un indice può essere definito su uno o più attributi. Se più attributi sono coinvolti in varie query, è necessario definire un indice multi-attributo. L'ordine degli attributi nell'indice deve corrispondere a quello nella query.
3. **Quando creare un indice clustered,** al più un indice per tabella può essere primario o clustering. Le query su range di valori si avvantaggiano di tali indici, mentre le query di ricerca su indici, che non restituiscono dati, non hanno miglioramenti con indici clustering.
4. **Quando usare indici hash invece di indici ad albero,** i database in genere usano i B+-Tree, utilizzabili sia con condizioni di uguaglianza sia con query su range di valori. Gli indici hash, invece, funzionano solo con condizioni di uguaglianza.
5. **Quando utilizzare hashing dinamico,** con i file di dimensioni molto variabili è consigliabile utilizzare tecniche di hashing dinamico (non offerte dai DBMS più commercializzati).

## DENORMALIZZARE UNO SCHEMA:

Lo scopo della normalizzazione è di separare attributi in relazione logica, per minimizzare la ridondanza ed evitare le anomalie di aggiornamento. Tali concetti a volte possono essere sacrificati per ottenere delle performance migliori su alcuni tipi di query che occorrono frequentemente. Questo processo è detto **denormalizzazione**. Il progettista aggiunge degli attributi ad uno schema per rispondere a delle query o a dei report per ridurre gli accessi a disco, evitando operazioni di join. Ad esempio, si potrebbe scegliere di denormalizzare uno schema di relazione da 4NF a 2NF, una forma più debole, ma che ridurrebbe gli accessi a disco in quanto eviterebbe la necessità di effettuare un'operazione di join.

## 4. MEMORIZZAZIONE DI RECORD ED ORGANIZZAZIONE DEI FILE

La collezione di dati che costituisce una base di dati computerizzata deve essere fisicamente memorizzata su un qualche **supporto di memoria** del calcolatore. Il software del DBMS potrà poi recuperare, aggiornare ed elaborare questi dati quando necessario. I supporti di memoria del calcolatore formano una gerarchia di memoria che comprende due categorie fondamentali:

- **Memoria principale:** comprende supporti di memoria su cui può operare direttamente l'unità centrale di elaborazione (CPU) del calcolatore, consente un rapido accesso ai dati ma ha una limitata capacità di memorizzazione. La **memoria cache**, costituita da una RAM statica, è la più costosa e viene tipicamente usata dalla CPU per accelerare l'esecuzione dei programmi. La **DRAM (Dynamic RAM)**, che fornisce l'area di lavoro principale della CPU per memorizzare programmi e dati ed è comunemente della **memoria centrale**. Il vantaggio della DRAM consiste nel suo basso costo, lo svantaggio è la sua volatilità e la minore velocità se confrontata con la RAM statica.
- **Memoria secondaria:** comprende dischi magnetici, dischi ottici e nastri, hanno una maggiore capacità, costano meno ma forniscono un accesso più lento ai dati rispetto ai dispositivi di memorizzazione principale. I dati nella memoria secondaria non possono essere elaborati direttamente dalla CPU, ma devono essere prima copiati nella memoria principale.

La **capacità di memoria** è misurata in kilobyte (Kbyte o  $2^{10}$  byte), megabyte (Mbyte o  $2^{20}$  byte), gigabyte (Gbyte o  $2^{30}$ ) e terabyte ( $2^{40}$  byte).

I programmi risiedono e vengono eseguiti nella DRAM. Basi di dati permanenti di grandi dimensioni risiedono in memoria secondaria, e quando necessario porzioni della base di dati sono poste in, e trascritte da, buffer di memoria centrale. In alcuni casi intere basi di dati possono essere memorizzate in memoria centrale, ma con un copia di backup su memoria secondaria, sono utili in applicazioni che richiedono tempi di risposta brevi.

La **memoria flash** non è volatile, è una memoria ad alta densità e prestazioni che usa la tecnologia EEPROM, il vantaggio risiede nell'elevata velocità di accesso, lo svantaggio nel fatto che deve essere cancellato e riscritto un intero blocco alla volta. I **CD-ROM** memorizzano i dati otticamente e sono letti da un laser, contengono dati preregistrati che non possono essere sovrascritti. I **dischi WORM** (Write Once Read Many) costituiscono una forma di memorizzazione ottica usata per archiviare dati, i dati vengono scritti una volta e letti un numero qualsiasi di volte senza avere la possibilità di cancellarli. I **juke-box di memorie ottiche** usano una schiera di CD-ROM, che vengono caricati nei drive (unità di lettura) su richiesta, hanno capacità nell'ordine di centinaia di gigabyte, ma i tempi di recupero sono nell'ordine di centinaia di millisecondi, e sono quindi più lenti dei dischi magnetici. Il **DVD** è uno standard recente per i dischi ottici che consente da quattro a quindici gigabyte di memoria per disco. I **nastri magnetici** sono usati per archiviate e salvare grandi quantità di dati. I juke-box di nastri, contenenti una banca di nastri che sono catalogati e che possono essere caricati automaticamente nei drive per i nastri, sono popolari come **memoria terziaria** per conservare terabyte di dati.

### MEMORIZZAZIONE DI UN DATABASE:

Le basi di dati memorizzano grandi quantità di dati, i quali devono persistere per lunghi periodi di tempo. Durante questi periodi si accede ai dati e li si elabora ripetutamente. Ciò è in contrasto con la nozione di strutture dati **transitorie** che persistono solo per un tempo limitato durante l'esecuzione del programma. La maggior parte delle basi di dati è memorizzata **permanentemente** (o **persistentemente**) su memoria secondaria perché:

- Di solito le basi di dati sono troppo ampie per poter essere interamente contenute in memoria centrale;
- Le circostanze che causano una perdita permanente di dati memorizzati si verificano meno frequentemente per la memorizzazione secondaria su disco che per la memorizzazione primaria, per questo motivo ci si riferisce al disco, e ad altri dispositivi di memoria secondaria, come a **memoria non volatile**, mentre la memoria centrale è spesso chiamata **memoria volatile**;
- Il costo di memorizzazione per unità di dato è di un ordine di grandezza inferiore per il disco che per la memoria centrale.

I nastri magnetici sono frequentemente usati come supporto di memoria per fare una copia di backup della base di dati, dato che la memorizzazione su nastro costa ancor meno della memorizzazione su disco, ma l'accesso ai dati su nastro è molto lento. I dati memorizzati su nastro sono off-line, cioè è necessario l'intervento di un operatore, o un dispositivo automatico di caricamento, per caricare un nastro prima che questi dati si rendano disponibili. Al contrario, i dischi sono dispositivi on-line a cui si può accedere direttamente in ogni momento.

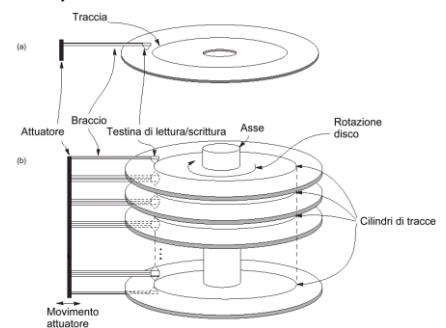
I progettisti della base di dati e il DBA devono conoscere i vantaggi e gli svantaggi di ciascuna tecnica di memorizzazione quando progettano, implementano e gestiscono una base di dati su uno specifico DBMS. Di solito il DBMS fornisce molte opzioni per organizzare i dati, e il processo di progettazione fisica di una base di dati prevede la scelta, fra le opzioni, delle particolari tecniche di organizzazione dei dati che meglio si adattano ai requisiti propri dell'applicazione. Gli implementatori di sistema DBMS devono studiare le tecniche di organizzazione dei dati per poterle implementare efficientemente e perciò fornire a DBA e utenti un numero sufficiente di opzioni. Le applicazioni tipiche di basi di dati hanno bisogno per l'elaborazione solo di una piccola porzione alla volta della basi di dati. Questa porzione deve essere localizzata su disco, copiata in memoria centrale per l'elaborazione, e quindi riscritta su disco se i dati sono cambiati. I dati memorizzati su disco sono organizzati come **file di record**.

### DISPOSITIVI DI MEMORIA SECONDARIA:

La più elementare unità dati su disco è il singolo **bit** di informazione. Magnetizzando un'area su disco, si può far sì che essa rappresenti un valore di bit 0 o 1. Per codificare le informazioni, i bit sono raggruppati in **byte**. Le dimensioni dei byte vanno da 4 a 8 bit, a seconda del computer e del dispositivo. Successivamente, verrà supposto che un carattere sia memorizzato in un singolo byte, e si useranno i termini **byte** e **carattere** indifferentemente.

Per **capacità** di un disco si intende il numero di byte che esso può memorizzare.

Un disco è **a singola faccia (single-sided)** se memorizza informazioni su uno solo dei suoi lati e **a doppia faccia (double-sided)** se sono usati entrambi i lati. Per aumentare la capacità di memorizzazione i dischi sono assemblati in una **pila di dischi**. L'informazione è memorizzata sulla faccia di un disco su circonferenze concentriche di **piccola larghezza**, ciascuna delle quali ha un diverso diametro. Ogni circonferenza è detta **traccia (track)**. Per le pile di dischi le tracce con lo stesso diametro sulle varie facce sono dette **cilindro** per la struttura che formerebbero se connesse nello spazio. I dati memorizzati in un cilindro possono essere recuperati molto più velocemente se fossero distribuiti tra cilindri diversi. Dato che una traccia contiene una grande quantità di informazione, essa viene suddivisa in blocchi o settori. La suddivisione di una traccia in **settori** è codificata nell'hardware sulla faccia del disco e non può essere cambiata.



Un tipo di organizzazione a settori chiama settore una porzione di traccia che sottende un angolo al centro. Un'altra organizzazione a settori, consiste nell'avere settori che sottendono angoli al centro più piccoli man mano che ci si allontana dal centro stesso, mantenendo così una densità di memorizzazione costante. Non tutti i dischi hanno le tracce suddivise in settori. La divisione di una traccia in **blocchi di disco (o pagine)** di uguale dimensione è fissata dal sistema operativo durante la **formattazione (o inizializzazione)** del disco, e non può essere cambiata dinamicamente. Tipiche dimensioni di un blocco di disco vanno da 512 a 4096 byte. Un disco con settori codificati nell'hardware ha spesso i settori che vengono suddivisi in

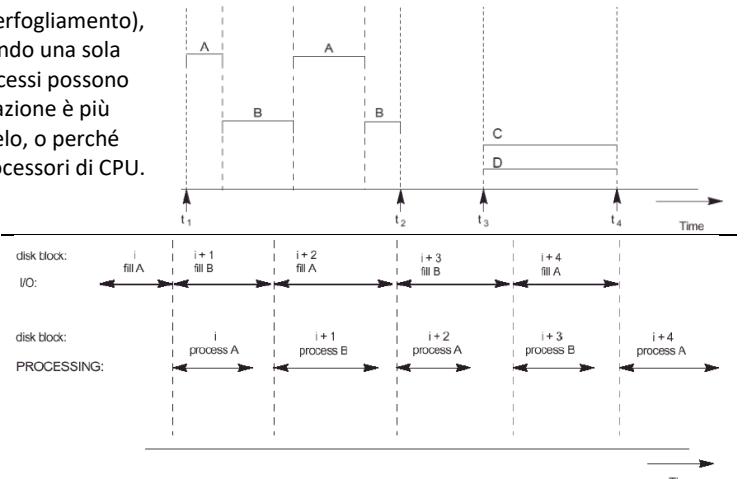
blocchi durante l'inizializzazione del disco. I blocchi sono separati da **spazi tra blocchi** di dimensioni fisse, che comprendono informazioni di controllo codificate appositamente, scritte durante l'inizializzazione del disco. Queste informazioni sono usate per determinare quale blocco della traccia segue ciascuno spazio tra blocchi. Un disco è un dispositivo indirizzabile ad *accesso casuale*. Il trasferimento dati tra la memoria centrale e il disco avviene in unità di blocchi di disco. L'**indirizzo hardware** di un blocco, una combinazione di numero di faccia, numero di traccia (nella faccia) e numero di blocco (nella traccia), è fornito all'hardware dell'input/output (I/O) del disco. Viene anche fornito l'indirizzo di un **buffer**, un'area riservata di locazioni contigue in memoria centrale che può contenere un blocco. Con un comando **read** il blocco dal disco è copiato nel buffer, mentre con un comando **write** il contenuto del buffer è copiato nel blocco di disco. Talora possono essere trasferiti come una cosa sola molti blocchi contigui, detti **cluster**. In questo caso la dimensione del buffer è adattata per corrispondere al numero di byte presenti nel cluster. L'effettivo meccanismo hardware che legge o scrive un blocco è la **testina di lettura/scrittura**, che fa parte di un sistema detto **unità disco (disk drive)**. Un disco o pila di dischi sono inseriti nell'unità disco, che ha un motore che fa ruotare i dischi. Una testina di lettura/scrittura è costituita da un componente elettronico fissato a un **braccio meccanico**. Le pile di dischi con più facce sono controllate da molte testine di lettura/scrittura, una per ogni faccia. Tutti i bracci sono collegati a un **attuatore** unito a un altro motore elettrico, che muove le testine di lettura/scrittura all'unisono e le posiziona esattamente sopra il cilindro di tracce specificato nell'indirizzo di un blocco. Le unità disco per i dischi rigidi fanno ruotare la pila di dischi ininterrottamente a una velocità costante. Una volta che la testina di lettura/scrittura è stata posizionata sulla traccia corretta e il blocco specificato nell'indirizzo di blocco si muove sotto di essa, il componente elettronico della testina di lettura/scrittura viene attivato per trasferire i dati. Alcune unità a disco hanno testine di lettura/scrittura fisse, con tante testine quanto sono le tracce. Queste unità sono dette **dischi a testina fissa**, mentre le unità a disco con un attuatore sono dette **dischi a testina mobile**. Per i dischi a testina fissa, una traccia o cilindro è selezionata tramite la commutazione elettronica all'appropriata testina di lettura/scrittura piuttosto che tramite un effettivo movimento meccanico, di conseguenza questa unità è molto più veloce. Tuttavia, il costo delle testine di lettura/scrittura aggiuntive è piuttosto alto, e pertanto i dischi a testina fissa non sono usati comunemente. Un **disk controller** (controllore di disco), inserito nell'unità disco, controlla l'unità disco e la interfaccia al sistema di elaborazione. Una delle interfacce standard usate oggi per le unità disco nei PC e nelle workstation è detta **SCSI** (Small Computer Storage Interface). Il controller accetta comandi di I/O di alto livello e intraprende azioni appropriate per posizionare il braccio, facendo sì che abbia luogo l'azione di lettura/scrittura. Per trasferire un blocco di disco, dato il suo indirizzo, il controller deve prima posizionare meccanicamente la testina di lettura/scrittura sulla traccia corretta. Il tempo necessario per fare ciò è detto **tempo di posizionamento (seek time)**, tempi tipici sono di 12-14 msec per i personal computer e di 8-9 msec per i server. Segue un ritardo, detto **latenza**, per attendere che l'inizio del blocco desiderato ruoti fino alla posizione sotto la testina di lettura/scrittura. Infine, c'è bisogno di un tempo aggiuntivo per trasferire i dati, detto **tempo di trasferimento di blocco**. Perciò il tempo totale necessario per localizzare e trasferire un blocco arbitrario, dato il suo indirizzo, è costituito dalla somma del tempo di posizionamento, ritardo di rotazione e tempo di trasferimento di blocco. Il tempo di posizionamento e il ritardo di rotazione sono solitamente molto più grandi del tempo di trasferimento di blocco. Per rendere più efficiente il trasferimento di più blocchi, è comune trasferire molti blocchi consecutivi sulla stessa traccia o cilindro. Ciò elimina il tempo di posizionamento e il ritardo di rotazione per tutti i blocchi tranne il primo e può avere come risultato un risparmio di tempo quando vengono trasferiti numerosi blocchi contigui. Il tempo necessario per localizzare e trasferire un blocco di disco è dell'ordine dei millisecondi, andando di solito dai 12 ai 60 msec, ma il trasferimento dei blocchi successivi può richiedere solo da 1 a 2 msec ciascuno. Perciò la localizzazione di dati su disco è un **collo di bottiglia importante** nelle applicazioni di basi di dati. Le strutture di file esaminate successivamente tentano di *ridurre al minimo il numero di trasferimenti di blocchi* necessari per localizzare e trasferire i dati.

#### BAFFERIZZAZIONE DI BLOCCHI:

Quando devono essere trasferiti da disco a memoria centrale numerosi blocchi, e sono noti tutti i loro indirizzi, per accelerare il trasferimento possono essere riservati in memoria centrale molti **buffer** (aree di memoria). Mentre viene letto o scritto un buffer, la CPU può elaborare i dati nell'altro buffer. Ciò è possibile perché esiste un processore indipendente per l'I/O di disco (controller) che, una volta avviato, può procedere nel trasferire un blocco di dati tra la memoria e il disco indipendentemente dall'operazione della CPU, e in parallelo con essa.

I processi A e B sono eseguiti **concorrentemente** e in modo **interleaved** (interfogliamento), mentre i processi C e D sono eseguiti **concorrentemente** e in **parallelo**. Quando una sola CPU controlla più processi, l'esecuzione parallela non è possibile. Però i processi possono ancora essere eseguiti concorrentemente in modo interleaved. La bufferizzazione è più utile quando i processi possono essere eseguiti concorrentemente in parallelo, o perché disponibile un processore di I/O di disco separato, o perché esistono più processori di CPU.

In Figura è illustrato come la lettura e l'elaborazione possano procedere in parallelo quando il tempo richiesto per elaborare un blocco di disco in memoria è minore del tempo richiesto per leggere il blocco successivo e riempire un buffer. La CPU può cominciare a elaborare un blocco una volta che il suo trasferimento in memoria centrale è completato, allo stesso tempo il processore dell'I/O di disco può essere impegnato nella lettura e nel trasferimento del blocco successivo in un buffer diverso.



Questa tecnica è detta **doppia bufferizzazione** e può essere usata anche per scrivere un flusso continuo di blocchi da memoria a disco. Essa consente una lettura o scrittura continua di dati su blocchi di disco consecutivi, che elimina il tempo di posizionamento e il ritardo di rotazione per i trasferimenti di tutti i blocchi escluso il primo. Inoltre, i dati sono tenuti pronti per essere elaborati, riducendo così il tempo di attesa nei programmi.

#### COLLOCAMENTO SU DISCO DEI RECORD DI UN FILE:

Ogni **record** è costituito da una collezione di **valori** collegati, dove ogni valore è formato da uno o più byte e corrisponde a un particolare **campo** del record, che possono essere interpretati come fatti relativi alle entità, ai loro attributi e alle loro associazioni e dovrebbero essere memorizzati su disco in modo da rendere possibile individuarne la collocazione ogni volta che se ne ha bisogno. Per esempio, un record IMPIEGATO rappresenta un'entità impiegato, e il valore di ciascun campo del record specifica alcuni attributi di quell'impiegato, come NOME, DATA\_NASCITA e STIPENDIO. Una collezione di **nomi di campi** e dei **tipi di dati** corrispondenti costituisce una definizione di **tipo di record** o **formato di record**. Un tipo di dati, associato a ciascun campo, specifica il tipo di valori che possono essere assunti da quel campo, che è di solito uno dei tipi di dati standard, come dati numerici (integer o floating), stringhe di caratteri (a lunghezza fissa o variabile), booleani (solo valori 0 e 1), e tipi di dati codificati in modo speciale date e time. Per ciascun sistema di elaborazione il numero di byte richiesti per ogni tipo di dati è fissato (integer 4 byte, booleano 1 byte e stringa a lunghezza fissa).

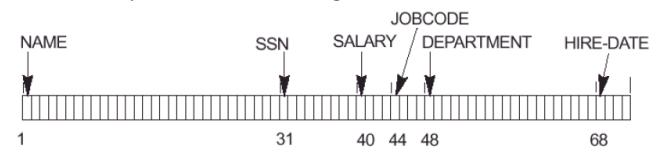
di  $k$  caratteri  $k$  byte). In applicazioni di basi di dati recenti può sorgere la necessità di memorizzare voci di dati che consistono di grandi oggetti non strutturati, rappresentanti immagini, audio o video, o testo libero. Queste sono indicate come **BLOB** (Binary Large OBject). Una voce di dati BLOB è tipicamente memorizzata separatamente dal suo record in un pool di blocchi di disco, e ci si limita a inserire nel record un puntatore al BLOB.

#### FILE, RECORD A LUNGHEZZA FISSA E A LUNGHEZZA VARIABILE:

Un **file** è una sequenza di record. In molti casi tutti i record presenti in un file fanno parte dello stesso tipo di record. Se ogni record nel file ha esattamente la stessa dimensione (in byte), si dice che il file è costituito da **record a lunghezza fissa**. Se record diversi nel file hanno dimensioni diverse, si dice che il file è costituito da **record a lunghezza variabile**. Un file può avere record a lunghezza variabile per molte ragioni:

- Uno o più campi del record ha dimensione variabile (**campi di lunghezza variabile**), ad esempio, il campo NOME di IMPIEGATO può essere un campo di lunghezza variabile;
- Uno o più campi del record può avere valori multipli per singoli record, un tale campo è detto **repeating field** (campo che si ripete) e un gruppo di valori per quel campo è spesso detto **repeating group** (gruppo che si ripete);
- Uno o più campi sono **opzionali**, cioè possono assumere valori per alcuni ma non per tutti i record del file (**campi opzionali**);
- Il file contiene record di *diversi tipi di record* e dimensione variabile (**file misto**). Ciò potrebbe verificarsi se record collegati di tipi diversi fossero *raggruppati (clustered)* su blocchi di disco, ad esempio, i record VOTAZIONE di uno studente possono essere collegati al record dello STUDENTE.

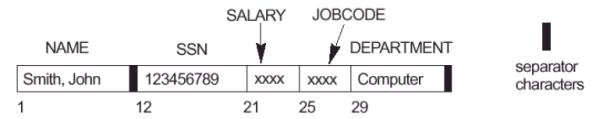
I **record a lunghezza fissa** IMPIEGATO in Figura presentano una dimensione di record di 71 byte. Ogni record ha gli stessi campi, e le lunghezze sono fisse, cosicché il sistema può individuare la posizione del byte di inizio di ciascun campo relativamente alla posizione iniziale del record. Ciò facilita la localizzazione dei valori del campo da parte dei programmi che accedono a questi file.



Si noti che è possibile rappresentare un file che da un punto di vista logico dovrebbe avere record a lunghezza variabile come un file di record a lunghezza fissa. Ad esempio, nel caso di campi opzionali sarebbe possibile aver inserito *tutti i campi* in ogni *record del file*, ma poi memorizzare uno speciale valore nullo se non esiste alcun valore per quel campo. Per un campo che si ripete, si potrebbero allocare in ogni record tanti spazi quant'è il *massimo numero di valori* che il campo può assumere. In entrambi i casi viene sprecato spazio quando certi record non presentano valori per tutti gli spazi fisici forniti in ciascun record.

Per **campi di lunghezza variabile** ogni record presenta un valore legato a ogni campo, ma non è nota la lunghezza esatta dei valori di alcuni campi.

Per determinare i byte che all'interno di un record rappresentano ogni campo, si possono usare speciali caratteri **separatori** (come ? o % o \$), che non si presentano in nessun valore del campo, per terminare campi di lunghezza variabile, o memorizzare la lunghezza in byte del campo in quel record, prima del suo valore.



Un file di record con **campi opzionali** può essere formattato in diversi modi. Se il numero totale di campi per il tipo di record è grande, ma il numero di campi che si presentano in un record è piccolo, è possibile inserire in ciascun record una sequenza di coppie <nome-campo, valore-campo> piuttosto che solo i valori del campo. Un'opzione più pratica è quella di assegnare un breve codice **tipo di campo**, ad esempio, un numero intero, a ciascun campo e inserire in ciascun record una sequenza di coppie <tipo-campo, valore-campo> piuttosto che coppie <nome-campo, valore-campo>.

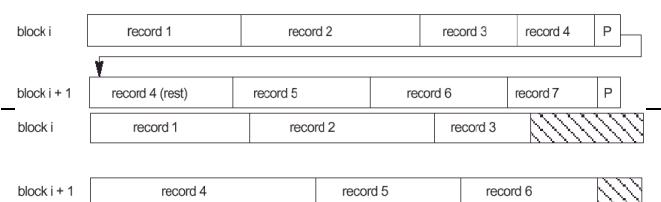
Un **campo che si ripete** ha bisogno di un carattere separatore per separare i valori del campo che si ripetono e di un altro carattere separatore per indicare la fine del campo. Infine, per un file che comprende **record di tipi diversi**, ogni record è preceduto da un indicatore di **tipo di record**. I programmi che elaborano file di record a lunghezza variabile, che sono di solito parte del file system e perciò nascosti ai programatori comuni, devono essere più complessi di quelli per record a lunghezza fissa, dove la posizione di partenza e la dimensione di ogni campo sono note e fissate.

#### RIPARTIZIONE DEI RECORD IN BLOCCHI E CONFRONTO TRA RECORD CON SPANNING E RECORD SENZA SPANNING:

I record di un file devono essere ripartiti su blocchi di disco perché un blocco è l'*unità di trasferimento dati* tra disco e memoria. Quando la dimensione del blocco è maggiore di quella del record, ogni blocco conterrà numerosi record, anche se alcuni file possono avere record inusualmente grandi che non possono trovar posto in un solo blocco. Si supponga che la dimensione del blocco sia di  $B$  byte. Per un file di record a lunghezza fissa della dimensione di  $R$  byte, con  $B \geq R$ , è possibile inserire  $bfr = [B/R]$  record per blocco, dove la  $[x]$  (funzione di floor) arrotonda per difetto il numero  $x$  ad un intero. Il valore  $bfr$  è detto **fattore di blocco (blocking factor)** per il file. In generale  $R$  può non dividere  $B$  esattamente, cosicché in ogni blocco si ha un certo spazio inutilizzato, pari a:  $B - (bfr * R)$  byte.

Per utilizzare questo spazio si può memorizzare parte di un record in un blocco e il resto in un altro. Un **puntatore** alla fine del primo blocco punta al blocco che contiene il resto del record nel caso in cui non ci sia il blocco consecutivo su disco.

Questa organizzazione è detta **spanned** (estesa), perché i record possono estendersi su più di un blocco. Ogni volta che un record è più grande di un blocco si deve usare un'organizzazione spanned.



Se ai record non è concesso di attraversare i confini di un blocco, l'organizzazione è detta **unspanned**. Essa è usata con record a lunghezza fissa che hanno  $B > R$ , perché consente a ciascun record di cominciare in una locazione nota del blocco, semplificando l'organizzazione dei record.

Se il record tipico è grande, è vantaggioso usare l'organizzazione con spanning per ridurre lo spazio perso. Per record a lunghezza variabile che usano l'organizzazione spanned, ogni blocco può memorizzare un diverso numero di record. In questo caso il fattore di blocco  $bfr$  rappresenta il numero medio di record per blocco per il file. Si può usare  $bfr$  per calcolare il numero di blocchi  $b$  necessari per un file di  $r$  record:  $b = [(r/bfr)]$  blocchi, dove la  $[x]$  (funzione ceiling) arrotonda per eccesso il valore di  $x$  al primo intero.

#### ALLOCAZIONE DEI BLOCCHI DI UN FILE SU DISCO:

Ci sono molte tecniche standard per allocare i blocchi di un file su disco. Nell'**allocazione continua** i blocchi del file sono allocati su blocchi di disco consecutivi, ciò rende molto veloce la lettura dell'intero file con l'uso della doppia bufferizzazione, ma allo stesso tempo rende difficile l'aumento delle dimensioni del file. Nell'**allocazione collegata (linked)** ogni blocco contiene un puntatore al successivo blocco del file, ciò facilita l'aumento delle dimensioni del file ma rallenta la lettura dell'intero file. Una combinazione delle due alloca **cluster** di blocchi di disco consecutivi, con i cluster collegati tra loro. I cluster sono talora detti **segmenti del file** o **estensioni del file**. Un'altra possibilità è quella di usare un'**allocazione indicizzata**, dove uno o più **blocchi di indici** contengono puntatori agli effettivi blocchi dei file. È anche comune usare combinazioni di queste tecniche.

## HEADER DEI FILE:

Un **header di file** (*intestazione*) o **descrittore di file** contiene informazioni su un file, necessarie ai programmi di sistema che accedono ai record del file. L'header comprende informazioni per determinare gli indirizzi di disco dei blocchi del file, nonché per le descrizioni dei formati dei record, che possono comprendere lunghezze di campo e ordine dei campi all'interno di un record per record a lunghezza fissa senza spanning, e codici di tipo di campo, caratteri separatori e codici di tipo di record per record a lunghezza variabile.

Per la ricerca di un record su disco, uno o più blocchi sono copiati nel buffer di memoria centrale. Quindi i programmi ricercano i record desiderati dentro i buffer, usando le informazioni presenti nell'header del file. Se l'indirizzo del blocco che contiene il record desiderato non è noto, i programmi devono effettuare una **ricerca lineare** attraverso i blocchi del file. Ogni blocco del file è copiato in un buffer ed esaminato fino a che il record è localizzato o tutti i blocchi del file sono stati esaminati senza successo. Ciò può essere molto dispendioso in termini di tempo per file grandi. Lo scopo di una buona riorganizzazione di file è quello di localizzare il blocco che contiene un record desiderato con il minimo numero di trasferimenti di blocco.

## OPERAZIONI SUI FILE:

Le operazioni su file sono raggruppate in **operazioni di recupero** (*retrieval operations*), non cambiano alcun dato nel file ma si limitano a localizzare certi record in modo tale che i valori dei loro campi possano essere esaminati ed elaborati, e **operazioni di aggiornamento** (*update operations*), cambiano il file con l'inserimento o la cancellazione di record o con la modifica dei valori di alcuni campi. In entrambi i casi è possibile che si debba **selezionare** uno o più record basandosi su una **condizione di selezione** (o **condizione di filtraggio**), che specifica i criteri che il record o i record desiderati devono soddisfare. Ad esempio, un file IMPIEGATO con campi NOME, SSN, STIPENDIO, CODICE\_LAVORO e DIPARTIMENTO, una **condizione di selezione semplice** può comportare un confronto di egualanza su un certo valore di campo (DIPARTIMENTO = 'Ricerca'). Condizioni più complesse possono coinvolgere altri tipi di operatori di confronto, come > o  $\geq$ . Il caso generale consiste nell'avere come condizione di selezione un'espressione booleana arbitraria sui campi del file. Le operazioni di ricerca su file sono basate su condizioni di selezione semplici. Una condizione complessa deve essere decomposta dal DBMS (o dal programmatore) per estrarre una condizione semplice che possa essere usata per localizzare i record su disco. Ogni record localizzato viene poi esaminato per decidere se soddisfa l'intera condizione di selezione. Ad esempio, si può estrarre la condizione semplice (DIPARTIMENTO='Ricerca') dalla condizione complessa ((STIPENDIO  $\geq$  30000) AND (DIPARTIMENTO = 'Ricerca')), ogni record che soddisfa (DIPARTIMENTO = 'Ricerca') viene localizzato e poi esaminato per vedere se soddisfa anche (STIPENDIO  $\geq$  30000).

Quando molti record di un file soddisfano una condizione di ricerca, viene inizialmente localizzato il **primo record**, relativamente alla sequenza fisica dei record del file, nominato **record corrente**, e localizzano il **successivo record** del file che soddisfa la condizione.

Le operazioni effettive per localizzare e accedere ai record di un file variano da sistema a sistema. Tipicamente programmi di alto livello, come i programmi software del DBMS, accedono ai record usando questi comandi, e talvolta nelle seguenti descrizioni ci si riferirà a **variabili di programma**.

- **Open**: prepara il file per la lettura o la scrittura. Alloca buffer appropriati (tipicamente almeno due) per contenere blocchi del file prelevati da disco, e recupera l'header del file. Imposta il file pointer (puntatore al file) all'inizio del file.
- **Reset**: imposta il file pointer di un file aperto all'inizio del file.
- **Find** (o **Locate**): cerca il primo record che soddisfa una condizione di ricerca. Trasferisce il blocco che contiene quel record in un buffer in memoria centrale (se non è già là). Il file pointer punta al record nel buffer ed esso diventa il *record corrente*.
- **Read** (o **Get**): copia il record corrente dal buffer in una variabile di programma del programma utente. Questo comando può anche far avanzare il puntatore dal record corrente al record successivo del file, il che può richiedere la lettura da disco del successivo blocco del file.
- **FindNext**: ricerca nel file il successivo record che soddisfa la condizione di ricerca. Trasferisce il blocco che contiene quel record in un buffer di memoria centrale (se non si trova già là). Viene localizzato il record nel buffer ed esso diventa il *record corrente*.
- **Delete**: cancella il record corrente e (alla fine) aggiorna il file su disco per rispecchiare la cancellazione.
- **Modify**: modifica i valori di alcuni campi del record corrente e alla fine, aggiorna il file su disco per rispecchiare la modifica.
- **Insert**: inserisce un nuovo record nel file localizzando il blocco in cui deve essere inserito il record, trasferendo quel blocco in un buffer in memoria centrale (se non è già là), scrivendo il record nel buffer e (alla fine) scrivendo il contenuto del buffer su disco per rispecchiare l'inserimento.
- **Close**: completa l'accesso al file rilasciando i buffer ed eseguendo tutte le altre operazioni di pulizia necessarie.

Le precedenti operazioni (tranne Open e Close) sono dette operazioni **un-record-alla-volta**, perché ogni operazione si applica a un solo record. È possibile snellire le operazioni Find, FindNext e Read in una sola operazione, Scan.

- **Scan**: se sul file sono appena state applicate le operazioni Open o Reset, restituisce il primo record, altrimenti fornisce il record successivo, se con l'operazione è specificata una condizione, il record restituito è il primo o il successivo record che soddisfa la condizione.

Nei sistemi di basi di dati possono essere applicate a un file operazioni aggiuntive di più alto livello **un-insieme-alla-volta**. Ad esempio:

- **FindAll**: localizza tutti i record nel file che soddisfano una condizione di ricerca.
- **FindOrdered**: recupera tutti i record nel file in un certo ordine specificato.
- **Reorganize**: comincia il processo di riorganizzazione. Alcune organizzazioni di file richiedono una riorganizzazione periodica. Un esempio consiste nel riordinare i record del file classificandoli sulla base di un campo specificato.

Un' **organizzazione di file** si riferisce all'organizzazione dei dati di un file in record, blocchi e strutture di accesso, ciò comprende il modo in cui i record e i blocchi sono posti nel supporto di memorizzazione il modo in cui sono collegati. Un **metodo di accesso** fornisce un gruppo di operazioni che possono essere applicate a un file. È possibile applicare molti metodi di accesso a una stessa organizzazione di file, alcuni, possono essere applicati solo a file organizzati in certi modi. Ad esempio, non è possibile applicare un metodo di accesso a indici a una file che non ha un indice.

Di solito ci si attende di usare alcune condizioni di ricerca piuttosto che altro. Alcuni file possono essere **statici**, nel senso che le operazioni di aggiornamento sono raramente eseguite su di essi, altri file, più **dinamici**, possono cambiare frequentemente, cosicché le operazioni di aggiornamento vengono costantemente applicate ad essi. Un'organizzazione di file di successo dovrebbe consentire di eseguire il più efficientemente possibile le operazioni che ci si attende di eseguire *frequentemente* sul file. Ad esempio, si consideri il file IMPIEGATO che contiene i record relativi agli attuali impiegati di un'azienda. Ciò che ci si attende è di dover inserire record (quando vengono assunti impiegati), cancellare record (quando impiegati lasciano l'azienda) e modificare record (ad esempio, quando cambia lo stipendio o il lavoro di un impiegato). La cancellazione o la modifica di un record richiede una condizione di selezione per identificare un particolare record o insieme di record.

## FILE DI RECORD NON ORDINATI (FILE HEAP):

In questo tipo di organizzazione i record sono collocati nel file nell'ordine in cui sono inseriti, quindi i nuovi record sono inseriti alla fine del file.

Un'organizzazione di questo tipo è detta **file heap** (*file sequenziali*). Essa viene spesso usata con percorsi di accesso opzionali, come gli indici secondari o per raccogliere e memorizzare record di dati in vista di un uso futuro.

L'inserimento di un nuovo record è *molto efficiente*, l'ultimo blocco di disco del file viene copiato in un buffer, viene aggiunto il nuovo record e poi il blocco viene **riscritto** sul disco. L'indirizzo dell'ultimo blocco del file viene tenuto nell'header del file. La ricerca di un record attraverso una qualsiasi condizione comporta una **ricerca lineare** sul file blocco per blocco, procedura dispendiosa. Se un solo record soddisfa la condizione di ricerca, allora un programma leggerà (portandoli in memoria) e ispezionerà metà dei blocchi del file prima di trovare il record. Per un file costituito da  $b$  blocchi, ciò può richiedere in media l'ispezione di  $(b/2)$  blocchi. Se nessun record o viceversa molti record soddisfano la condizione di ricerca, il programma deve leggere e ispezionare tutti i  $b$  blocchi del file.

Per cancellare un record, un programma deve dapprima trovare il suo blocco, copiare il blocco in un buffer, quindi cancellare il record dal buffer e infine **riscrivere il blocco** su disco. Ciò lascia spazio inutilizzato nel blocco di disco. La cancellazione di un gran numero di record ha come risultato uno spreco di spazio di memoria. Un'altra tecnica usata per la cancellazione di record consiste nel tenere memorizzato con ogni record un bit o byte supplementare, detto **indicatore di cancellazione**. Un record viene cancellato semplicemente ponendo l'indicatore di cancellazione a un certo valore. Un valore diverso dell'indicatore indica un record valido (cioè non cancellato). I programmi di ricerca considerano solo i record validi di ogni blocco, quando conducono la loro ricerca. Entrambe queste tecniche di cancellazione richiedono una **riorganizzazione** periodica del file per recuperare lo spazio inutilizzato dei record cancellati. Durante la riorganizzazione si accede consecutivamente ai blocchi del file, e i record vengono compattati rimuovendo quelli cancellati. Dopo una riorganizzazione di questo tipo i blocchi sono di nuovo riempiti fino al limite delle loro capacità. Un'altra possibilità è quella di usare lo spazio lasciato libero dai record cancellati quando si inseriscono nuovi record, anche se ciò richiede una contabilità aggiuntiva per tener traccia delle locazioni vuote.

Per un file non ordinato è possibile usare sia un'organizzazione spanned sia un'organizzazione unspanned, inoltre, esso può essere usato con record a lunghezza fissa o variabile. La modifica di un *record a lunghezza variabile* può richiedere la cancellazione del record vecchio e l'inserimento di un record modificato, perché il record modificato può non trovare posto nel suo vecchio spazio su disco. Per leggere tutti i record ordinati secondo i valori di un certo campo si può creare una copia ordinata del file. L'ordinamento è un'operazione dispendiosa per un grande file su disco, vengono per queste usate tecniche speciali per l'**ordinamento esterno**. Per un file di *record a lunghezza fissa* non ordinati, che usa *blocchi senza spanning e allocazione contigua*, viene naturale accedere a ogni record a partire dalla sua **posizione** nel file. Se i record del file sono numerati 0, 1, 2, ...,  $r - 1$  e i record in ogni blocco sono numerati 0, 1, ...,  $bfr - 1$ , dove  $bfr$  è il fattore di blocco, allora l'  $i$ -esimo record del file è posto nel blocco  $[ (i/bfr) ]$  ed è l' $(i \bmod bfr)$ -esimo record di quel blocco. Un file di questo tipo viene spesso chiamato **file relativo** o **diretto** perché si può agevolmente accedere direttamente ai record a partire dalle loro posizioni relative. L'accesso a un record a partire dalla sua posizione non aiuta a localizzare un record basandosi su una condizione di ricerca, però facilita la costituzione di percorsi di accesso al file.

#### FILE DI RECORD ORDINATI (FILE SORTED):

È possibile ordinare fisicamente i record di un file su disco basandosi sui valori di uno dei loro campi, detto **campo di ordinamento**, ciò porta a un **file ordinato**. Se il campo di ordinamento è anche un **campo chiave** del file, allora esso è detto **chiave di ordinamento** del file.

In Figura è mostrato un file ordinato con NOME come campo chiave di ordinamento.

Nei file di record ordinati, la lettura dei record secondo l'ordine dei valori della chiave di ordinamento diventa estremamente efficiente, perché non è richiesta alcuna azione di riordinamento. In secondo luogo, trovare il record successivo a quello corrente, secondo l'ordine della chiave di ordinamento, di solito non richiede accessi aggiuntivi ai blocchi, perché il record successivo è nello stesso blocco di quello corrente (a meno che il record corrente non sia l'ultimo del blocco). Infine, l'uso di una condizione di ricerca basata sul valore di un campo chiave di ordinamento ha come risultato un accesso più veloce quando viene usata la tecnica di ricerca binaria anche se non è usata spesso per i file su disco.

Una ricerca binaria per file su disco può essere fatta sui blocchi anziché sui record. Si supponga che il file abbia  $b$  blocchi numerati 1, 2, ...,  $b$ , e che i record siano ordinati per valore crescente del loro campo chiave di ordinamento e che si stia cercando un record il cui valore di campo chiave di ordinamento sia  $K$ . Supponendo che gli indirizzi su disco dei blocchi del file siano disponibili nell'header del file.

Una ricerca binaria di solito accede a  $\log_2(b)$  blocchi, che il record sia trovato o meno, rispetto alle ricerche lineari, dove, in media, si accede a  $(b/2)$  blocchi quando il record viene trovato e a  $b$  blocchi quando non viene trovato. Un criterio di ricerca che coinvolge le condizioni  $>$ ,  $<$ ,  $\geq$  e  $\leq$  sul campo di ordinamento è abbastanza efficiente, dal momento che l'ordinamento fisico dei record implica che tutti i record che soddisfano la condizione siano contigui nel file. Per esempio, relativamente alla Figura, se il criterio di ricerca è  $(\text{NOME} < 'G')$  i record che soddisfano il criterio di ricerca sono quelli che vanno dall'inizio del file fino al primo record che ha un valore NOME che comincia con la lettera G.

Per l'accesso casuale ai record si utilizza una ricerca lineare, in quanto quella binaria non fornisca alcun vantaggio. Per accedere ai record in un ordine basato su un campo non di ordinamento è necessario creare un'altra copia ordinata, in ordine diverso, del file.

Per un file ordinato l'inserimento e la cancellazione di record sono operazioni dispendiose perché i record devono rimanere fisicamente ordinati. Per inserire un record si deve trovare la sua posizione corretta nel file, basandosi sul valore del suo campo di ordinamento, e quindi far spazio nel file per inserire il record in quella posizione. Per un file di grosse dimensioni ciò può essere molto dispendioso in termini di tempo perché, in media, metà dei record del file devono essere spostati per fare spazio ai nuovi record. Ciò significa che metà dei blocchi del file devono essere letti e riscritti dopo che i record vengono spostati fra di loro. Per la cancellazione di un record il problema è meno grave se vengono usati gli indicatori di cancellazione e una riorganizzazione periodica. Un'opzione per rendere più efficiente l'inserimento consiste nel tenere in ogni blocco un certo spazio inutilizzato per nuovi record. Ma una volta che questo spazio è stato utilizzato, il problema si ripresenta. Un altro metodo usato consiste nel creare un file *non ordinato* temporaneo detto file di **overflow** o di **transazione**. Con questa tecnica il file ordinato vero e proprio è detto file **principale** o file **master**. I nuovi record vengono inseriti alla fine del file di overflow anziché nella loro posizione corretta nel file principale. Periodicamente, durante la riorganizzazione del file, il file di overflow viene ordinato e quindi fuso con il file master. L'inserimento diviene molto efficiente, ma al costo di una complessità aggiuntiva nell'algoritmo di ricerca. Il file di overflow deve essere ispezionato usando una ricerca lineare se, dopo la ricerca binaria, il record non è stato trovato nel file principale. Per applicazioni che non richiedono le informazioni più recenti, nel corso di una ricerca i record di overflow possono essere ignorati.

La modifica del valore del campo di un record dipende da due fattori: la condizione di ricerca per localizzare il record e il campo che deve essere modificato. Se la condizione di ricerca coinvolge il campo chiave di ordinamento, allora si può localizzare il record usando una ricerca binaria, altrimenti occorre effettuare una ricerca lineare. Un campo non di ordinamento può essere modificato cambiando il record e riscrivendolo nella stessa locazione fisica su disco, supponendo di avere record a lunghezza fissa. La modifica del campo di ordinamento implica che il record possa cambiare la sua posizione nel file, il che richiederebbe la cancellazione del record vecchio seguita dall'inserimento del record modificato.

	NAME	SSN	BIRTHDATE	JOB	SALARY	SEX
block 1	Aaron, Ed					
	Abbott, Diane					
	⋮					
	Acosta, Marc					
block 2	Adams, John					
	Adams, Robin					
	⋮					
	Akers, Jan					
block 3	Alexander, Ed					
	Alfred, Bob					
	⋮					
	Allen, Sam					
⋮						
block n - 1	Wong, James					
	Wood, Donald					
	⋮					
	Woods, Marry					
block n	Wright, Pam					
	Wyatt, Charles					
	⋮					
	Zimmer, Byron					

Leggere i record del file secondo l'ordine del campo di ordinamento è abbastanza efficiente se si trascurano i record in overflow, dal momento che i blocchi possono essere letti consecutivamente usando la doppia bufferizzazione. Per includere i record in overflow, bisogna inserirli nelle loro posizioni corrette, in questo caso si può dapprima riorganizzare il file, e poi leggere i suoi blocchi sequenzialmente. Per riorganizzare il file, prima di tutto si ordinano i record nel file di overflow, e poi li si fonde con il file master. Durante la riorganizzazione i record segnati per la cancellazione vengono rimossi. I file ordinati sono usati raramente nelle applicazioni di basi di dati se non è usato un cammino di accesso aggiuntivo, detto **indice primario**, ciò ha come risultato un **file indicizzato sequenziale**. Questo migliora ulteriormente il tempo di accesso causale sul campo chiave di ordinamento.

## TECNICHE HASH:

Un altro tipo di organizzazione primaria di file è basato sull'hash, che fornisce un accesso molto rapido ai record sotto certe condizioni di ricerca. Questa organizzazione è detta di solito **file hash**. La condizione di ricerca deve essere una condizione di uguaglianza su un campo singolo, detto **campo hash** del file. Di solito il campo hash è anche un campo chiave del file, in questo caso è detto **chiave hash**. L'idea è quella di fornire una funzione  $h$ , detta **funzione hash** o **funzione di randomizzazione**, che è applicata al valore del campo hash di un record e fornisce l'*indirizzo* del blocco di disco in cui è memorizzato il record. Una ricerca del record all'interno del blocco può essere effettuata in un buffer in memoria centrale. Per la maggior parte dei record si ha bisogno solo di un accesso a un singolo blocco per recuperare quel record.

## HASH INTERNO:

Per file interni, l'hash è implementato con una **tavella hash** costruita usando un vettore di record. Si supponga che il campo di valori possibili per l'indice del vettore vada da 0 a  $M-1$ , si ha pertanto  $M$  slot i cui indirizzi corrispondono agli indici del vettore. Si sceglierà allora una funzione hash che trasforma il valore del campo hash in un intero compreso tra 0 e  $M-1$ . Una funzione hash comune è la funzione  $h(K)=K \bmod M$ , che fornisce il resto di un valore interno di campo hash  $K$  dopo la divisione per  $M$ , questo valore viene usato per l'indirizzo del record. Valori non interi del campo hash possono essere trasformati in valori interi prima che venga applicata la funzione. Per stringhe di caratteri possono essere usati nella trasformazione i codici numerici (ASCII) associati ai caratteri.

NAME	SSN	JOB	SALARY
0			
1			
2			
3			
⋮			
$M-2$			
$M-1$			

Una tecnica, detta **folding**, prevede di applicare una funzione aritmetica come l'*addizione* o una funzione logica come l'*or esclusivo* a parti diverse del valore del campo hash per calcolare l'indirizzo hash. Un'altra tecnica prevede la raccolta di alcune cifre del valore del campo hash, ad esempio la terza, la quinta, l'ottava cifra, per formare l'indirizzo hash. Il problema con la maggior parte delle funzioni hash è che esse non garantiscono che valori distinti saranno trasformati in indirizzi distinti, perché lo **spazio del campo hash**, ovvero il numero di valori possibili che un campo hash può assumere, è di solito molto più grande dello **spazio di indirizzi**, ovvero il numero di indirizzi disponibili per i record.

Una **collusione** si verifica quando il valore del campo hash di un record che sta per essere inserito è trasformato da una funzione hash in un indirizzo che contiene già un record diverso. In questa situazione occorre inserire il nuovo record in un'altra posizione, visto che l'indirizzo hash è occupato. Il processo di ricerca di un'altra posizione è detto **risoluzione delle collisioni**:

- **Indirizzamento aperto**: procedendo dalla posizione occupata specificata dall'indirizzo hash, il programma verifica in ordine le posizioni successive fino a che non si trova una posizione inutilizzata (vuota).
- **Concatenamento**: per questo metodo si estende il vettore con un certo numero di posizioni di overflow (locazioni vuote). Inoltre, a ogni locazione di record viene aggiunto un campo puntatore. Una collisione viene risolta ponendo il nuovo record in una locazione di overflow inutilizzata e imponendo il valore dell'indirizzo di tale locazione al puntatore presente nella locazione di indirizzo hash occupata. Si mantiene perciò una lista concatenata di record di overflow per ogni indirizzo hash.
- **Hash multiplo**: il programma applica una seconda funzione hash se la prima ha come risultato una collisione. Se risulta un'altra collisione, il programma usa l'indirizzamento aperto o applica una terza funzione hash e poi, se necessario, usa l'indirizzamento aperto.

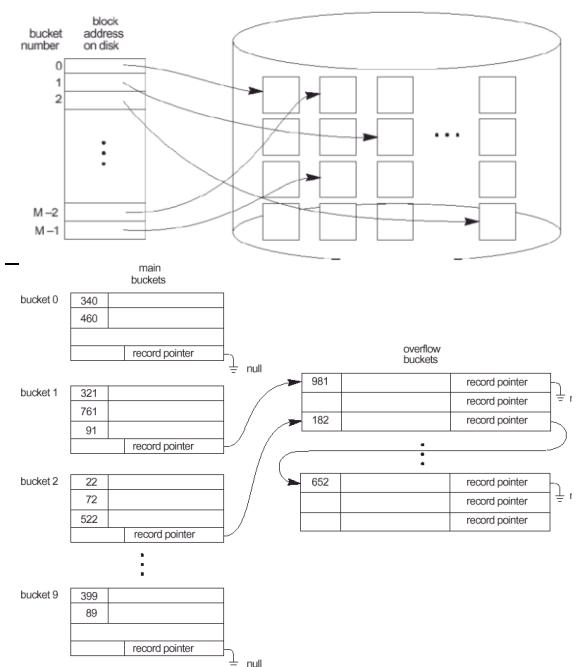
Ogni metodo di risoluzione delle collisioni richiede il proprio algoritmo per l'inserimento, il recupero e la cancellazione di record. Gli algoritmi per il concatenamento sono i più semplici, gli algoritmi di cancellazione per l'indirizzamento aperto sono piuttosto complicati. Lo scopo di una buona funzione hash è quello di distribuire i record uniformemente sullo spazio di indirizzi, in modo da minimizzare le collisioni senza lasciare molte locazioni inutilizzate. È meglio tenere una tabella hash piena tra il 70% e 90%, così che il numero di collisioni rimane basso e non si spreca troppo spazio. Perciò se si prevede di avere  $r$  record da memorizzare nella tabella, si dovranno scegliere  $M$  locazioni per lo spazio di indirizzi in modo tale che  $(r/M)$  sia compreso tra 0,7 e 0,9. Può essere utile scegliere un numero primo per  $M$ , dal momento che è stato dimostrato che ciò distribuisce meglio gli indirizzi hash sullo spazio di indirizzi quando viene usata la funzione mod hash. Altre funzioni hash richiedere che  $M$  sia una potenza di 2.

## HASH ESTERNO PER FILE SU DISCO:

L'hash per file su disco è detto **hash esterno**. Per adattarsi alle caratteristiche della memorizzazione su disco, lo spazio di indirizzi obiettivo è fatto di **bucket**, ciascuno dei quali contiene più record. Un bucket è un blocco di disco o un cluster di blocchi contigui. La funzione hash mappa una chiave in un numero relativo per il bucket, piuttosto di assegnare al bucket un indirizzo di blocco assoluto. Una tabella contenuta nell'header del file converte il numero del bucket nel corrispondente indirizzo di blocco di disco.

Con i bucket il problema delle collisioni è meno grave perché possono essere inviati nello stesso bucket senza causare problemi. Però occorre anche provvedere al caso in cui un bucket sia riempito fino al limite e un nuovo record che sta per essere inserito venga inviato dalla funzione hash in quel bucket.

È possibile usare una variazione del concatenamento in cui in ogni bucket si mantiene un puntatore a una lista concatenata di record di overflow per il bucket, come mostrato in Figura. I puntatori nella lista concatenata dovrebbero essere **puntatori a record**, che comprendono sia un indirizzo di blocco sia una posizione relativa del record nel blocco. L'hash fornisce il più rapido accesso possibile per il recupero di un record arbitrario dato il valore del suo campo hash. Anche se la maggior parte delle buone funzioni hash non mantengono i record nell'ordine fissato dai valori del campo chiave, ci sono alcune funzioni, dette **order preserving** (preservano l'ordine), che lo fanno, ad esempio, la funzione hash che prende come indirizzo hash le tre cifre più a sinistra di un campo numero di fattura, e che tiene in ogni bucket i record ordinati secondo il numero di fattura.



Un altro esempio consiste nell'usare una chiave hash intera direttamente come indice per un file relativo, se i valori della chiave hash riempiono un particolare intervallo, ad esempio, se i numeri di impiegato in un'azienda vengono assegnati come 1, 2, 3, ... fino al numero totale di impiegati, si può usare la funzione hash identità che mantiene l'ordine. Purtroppo, ciò funziona solo se le chiavi sono generate in ordine da una certa applicazione.

Lo schema hash descritto è detto **hash statico** perché viene allocato un numero fisso  $M$  di bucket. Ciò può costituire un inconveniente per file dinamici. Si supponga di allocare  $M$  bucket per lo spazio di indirizzi, e sia  $m$  il numero massimo di record che possono trovar posto in un bucket, perciò nello spazio allocato troveranno posto al più ( $m \cdot M$ ) record. Se il numero di record è minore di ( $m \cdot M$ ), rimane spazio inutilizzato. Se il numero di record aumenta fino a molto più di ( $m \cdot M$ ), si verificheranno collisioni e il recupero sarà rallentato a causa delle lunghe liste di record di overflow. In entrambi i casi ci si potrebbe trovare nella situazione di dover cambiare il numero di blocchi allocati  $M$  e quindi usare una nuova funzione hash (basata sul nuovo valore di  $M$ ) per ridistribuire i record. Queste riorganizzazioni sono dispendiose in termini di tempo per file di grandi dimensioni. Organizzazioni più recenti per file dinamici basate sull'hash consentono al numero di bucket di variare dinamicamente, limitandosi a una riorganizzazione localizzata.

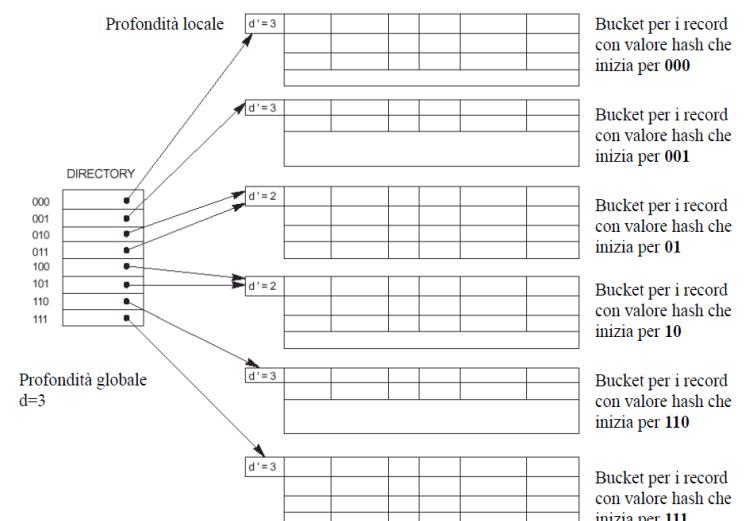
Quando si usa l'**hash esterno**, la ricerca di un record, dato il valore di un certo campo diverso dal campo hash, è tanto dispendiosa quanto nel caso di un file non ordinato. La cancellazione di record può essere implementata rimuovendo il record dal suo bucket. Se il bucket ha una catena di overflow, si può spostare uno dei record di overflow nel bucket per sostituire il record cancellato. Se il record che deve essere cancellato è già in overflow, lo si rimuove dalla lista concatenata. La rimozione di un record in overflow implica di dover tenere traccia delle posizioni di overflow vuote. Ciò può essere fatto mantenendo una lista concatenata di locazioni di overflow inutilizzate. La modifica del valore di campo di un record dipende da due fattori: la condizione di ricerca per localizzare il record e il campo che deve essere modificato. Se la condizione di ricerca è basata sull'uguaglianza del campo hash, è possibile localizzare il record efficientemente usando la funzione hash, altrimenti occorre fare una ricerca lineare. Un campo non hash può essere modificato cambiando il record e riscrivendolo nello stesso bucket. La modifica del campo hash implica che il record possa spostarsi in un altro bucket, il che richiede la cancellazione del record vecchio seguita dall'inserimento del record modificato.

#### TECNICHE DI HASHING PER L'ESPAZIONE DINAMICA DEI FILE:

Un grave inconveniente dello schema **hash statico** è che lo spazio di indirizzo hash è fisso, e quindi è difficile espandere o ridurre il file dinamicamente.

L'**hash estendibile**, memorizza una struttura d'accessi insieme al file, simile all'indicizzazione. La differenza è che la struttura d'accesso è basata sui valori che risultano dopo l'applicazione della funzione hash al campo di ricerca. Nell'indicizzazione, invece, la struttura d'accesso è basata sui valori del campo di ricerca stesso. Più precisamente, nell'**hash estendibile**, si mantiene una specie di **directory**, un vettore di  $2^d$  indirizzi di bucket, dove  $d$  è detta **profondità globale (global depth)** della directory. Il valore intero corrispondente ai primi  $d$  bit (i più significativi) di un valore hash è usato come un indice per il vettore per determinare un elemento della directory, e l'indirizzo in corrispondenza a quell'elemento determina il bucket in cui sono memorizzati i record corrispondenti. Però non ci deve necessariamente essere un bucket distinto per ciascuna delle  $2^d$  locazioni della directory. Molte locazioni della directory con gli stessi  $d'$  primi bit per i loro valori hash possono contenere lo stesso indirizzo di bucket, se tutti i record che vengono inviati dalla funzione hash in queste locazioni trovano posto in un bucket singolo. Una **profondità locale (local depth)  $d'$** , memorizzata con ogni bucket, specifica il numero di bit su cui è basato il contenuto del bucket.

In Figura viene mostrata una directory con profondità globale  $d=3$ . Il valore di  $d$  può essere incrementato o decrementato di un'unità alla volta, raddoppiando o dimezzando così come il numero di elementi nel vettore directory. Il raddoppio è necessario se un bucket, la cui profondità locale  $d'$  è uguale alla profondità globale  $d$ , va in overflow. Il dimezzamento si verifica se  $d > d'$  per tutti i bucket dopo che si verifica qualche cancellazione. La maggior parte dei recuperi di record richiede due accessi ai blocchi, uno alla directory e l'altro al bucket. Per illustrare lo sdoppiamento dei bucket, si supponga che l'inserimento di un nuovo record causi un overflow nel bucket i cui valori hash cominciano con 01, il terzo bucket in Figura. I record saranno distribuiti tra due bucket, il primo contiene tutti i record i cui valori hash cominciano con 010, il secondo tutti quelli i cui valore hash cominciano con 011. Ora le due locazioni della directory per 010 e 011 puntano ai due nuovi bucket distinti. Prima dello sdoppiamento esse puntavano allo stesso bucket. La profondità locale  $d'$  dei due nuovi bucket è 3, che è di un'unità superiore alla profondità locale del bucket vecchio.



Se un bucket va in overflow e deve essere sdoppiato, aveva originariamente una profondità locale  $d'$  uguale alla profondità globale  $d$  della directory, allora la dimensione della directory deve essere raddoppiata, così si può usare un bit aggiuntivo per distinguere fra loro i due nuovi bucket. Ad esempio, se, in Figura, il bucket per i record i cui valori hash cominciano con 111 va in overflow, allora i due nuovi bucket hanno bisogno di una directory con profondità globale  $d=4$ , perché i due nuovi bucket sono ora etichettati 1110 e 1111, e perciò le loro profondità locali sono entrambe pari a 4. La dimensione della directory è perciò raddoppiata, e anche ognuna delle altre locazioni originali nella directory è suddivisa in due locazioni, ciascuna delle quali ha lo stesso valore di puntatore della locazione originale.

Il principale vantaggio dell'**hash estendibile** è che le prestazioni del file non degradano man mano che il file cresce di dimensione, al contrario di quanto avviene con l'hash esterno statico, in cui le collisioni aumentano e il corrispondente concatenamento causa accessi aggiuntivi. Inoltre, nell'hash estendibile non viene allocato nessuno spazio per una crescita futura, ma bucket aggiuntivi possono essere allocati dinamicamente quando necessario. Lo spazio in più per la tabella di directory è trascurabile. La dimensione massima della directory è  $2^k$ , dove  $k$  è il numero di bit nel valore hash. Un altro vantaggio è che lo sdoppiamento causa in molti casi una minore riorganizzazione, dal momento che solo i record in un bucket vengono ridistribuiti sui due nuovi bucket. L'unica occasione in cui una riorganizzazione è più dispendiosa si presenta quando la directory deve essere raddoppiata (o dimezzata). Uno svantaggio è che la directory deve essere ispezionata prima di accedere ai bucket veri e propri, il che ha come risultati due accessi ai blocchi anziché uno solo, com'è invece nell'hash statico. Questa penalizzazione nelle prestazioni è considerata di minore importanza.

La seconda tecnica, detta **hash lineare**, non richiede strutture d'accesso addizionali. In questi schemi hash il risultato dell'applicazione di una funzione hash è un intero non negativo, e perciò può essere rappresentato come un numero binario. La struttura d'accesso è costruita sulla **rappresentazione binaria** del risultato della funzione hash, che è una stringa di **bit**. Tale rappresentazione sarà detta **valore hash** di un record. I record sono distribuiti fra i bucket sulla base dei **bit iniziali (leading bits)** presenti nei loro valori hash.

Più precisamente, nell'**hash lineare**, l'idea è quella di consentire a un file hash di espandere e ridurre dinamicamente il suo numero di bucket senza aver bisogno di una directory. Si supponga che il file abbia inizialmente  $M$  bucket numerati 0, 1, ...,  $M-1$  e usi la funzione mod hash  $h(K)=K \bmod M$ , questa funzione hash è detta funzione hash iniziale  $h_i$ . L'overflow a seguito di collisioni è ancora necessario e può essere gestito mantenendo singole

catene di overflow per ogni bucket. Però, quando una collisione porta a un record di overflow in *ogni* bucket del file, il *primo* bucket nel file (bucket 0) viene sdoppiato in due bucket: il bucket originario 0 e un nuovo bucket  $M$  alla fine del file. I record originariamente nel bucket 0 sono distribuiti tra i due bucket sulla base di una diversa funzione hash  $h_{i+1}(K)=K \bmod 2M$ . Una proprietà fondamentale delle due funzioni hash  $h_i$  e  $h_{i+1}$  è che ogni record che era inviato al bucket 0 sulla base di  $h_i$  sarà inviato o al bucket 0 o al bucket  $M$  sulla base di  $h_{i+1}$ , ciò è necessario affinché l'hash lineare funzioni. Quando ulteriori collisioni portano a record di overflow, vengono sdoppiati altri bucket nell'ordine *lineare* 1, 2, 3, .... Se si verificano abbastanza overflow, saranno stati sdoppiati tutti i bucket originari del file, 0, 1, 2, ...,  $M-1$ , cosicché ora il file avrà  $2M$  bucket anziché  $M$ , e tutti i bucket useranno la stessa funzione hash  $h_{i+1}$ . Perciò i record in overflow sono ridistribuiti in bucket regolari, usando la funzione  $h_{i+1}$  attraverso uno *sdoppiamento differito* dei loro bucket. Non c'è alcuna directory, è necessario solo un valore  $n$ , inizialmente posto a 0 e incrementato di 1 ogni volta che si verifica uno sdoppiamento, per determinare quali bucket sono stati sdoppiati. Per recuperare un record con valore di chiave hash  $K$ , dapprima si applica la funzione  $h_i$  a  $K$ , se  $h_i(K) < n$ , allora si applica la funzione  $h_{i+1}$  a  $K$  perché il bucket è già stato sdoppiato. Inizialmente è  $n=0$ , a indicare che la funzione  $h_i$  si applica a tutti i bucket,  $n$  cresce linearmente quando i bucket vengono sdoppiati. Quando  $n=M$  dopo essere stato incrementato, ciò significa che tutti i bucket originari sono stati sdoppiati e la funzione hash  $h_{i+1}$  si applica a tutti i record nel file. A questo punto,  $n$  è riportato a 0, e ogni nuova collisione che causa overflow porta all'uso di una nuova funzione hash  $h_{i+j}(K)=K \bmod (2/M)$ . In generale si usa una sequenza di funzioni hash  $h_{i+j}(K)=K \bmod (2/M)$ , dove  $j = 0, 1, \dots$ , è necessaria una nuova funzione hash  $h_{i+j+1}$  ogni volta che tutti i bucket 0, ...,  $(2/M)-1$  sono stati sdoppiati e  $n$  è riportato a 0.

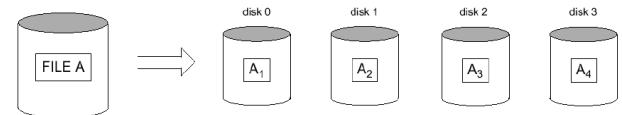
## TECNOLOGIA RAID:

Un miglioramento nella tecnologia della memoria secondaria è rappresentato dallo sviluppo della tecnologia **RAID**, acronimo di **Redundant Array of Inexpensive** (o **Independent**) **Disks** (vettori ridondanti di dischi economici o indipendenti). Lo scopo principale della tecnologia RAID è quello di uguagliare i tassi molto diversi di miglioramento delle prestazioni dei dischi rispetto a quelli della memoria principale e dei microprocessori.

Una soluzione naturale consiste in un grande vettore di piccoli dischi indipendenti che si comporta come un singolo disco logico di maggiori prestazioni. Viene usato un concetto detto **data striping** (suddivisione dei dati), che utilizza il *parallelismo* per incrementare le prestazioni del disco. Il data striping distribuisce i dati in modo trasparente fra più dischi, così che essi si comportino come un unico disco, grande e veloce.

In Figura è rappresentato un file distribuito o *suddiviso (striped)* fra quattro dischi.

La suddivisione migliora le prestazioni di I/O complessive consentendo che I/O multipli vengano serviti in parallelo, fornendo così alti tassi di trasferimento. La suddivisione dei dati realizza anche un bilanciamento del carico fra i dischi.



## MIGLIORAMENTO DELL'AFFIDABILITÀ:

Per un vettore di  $n$  dischi la frequenza di un guasto è  $n$  volte quella che si ha per un solo disco. Perciò, se l'MTTF (Mean Time To Failure: tempo medio perché si verifichi un guasto) di un'unità disco è considerato di 200.000 ore (tempi tipici vanno fino a circa 1 milione di ore), quello di un banco di 100 unità disco diventa solo di 2000 ore. Mantenere una sola copia di dati in un tale vettore di dischi causerà una significativa perdita di affidabilità.

Un'ovvia soluzione è quella di servirsi di una certa ridondanza dei dati in modo tale che i guasti ai dischi possano essere tollerati. Gli svantaggi sono le operazioni di I/O aggiuntive per la scrittura, calcoli supplementari per mantenere la ridondanza e per effettuare il ripristino dagli errori, e capacità di disco aggiuntiva per memorizzare informazioni ridondanti.

Una tecnica per introdurre ridondanza è detta **mirroring o shadowing**, dove i dati sono scritti con ridondanza in due dischi fisici identici, che sono trattati come un solo disco logico. Quando i dati vengono letti, essi possono essere recuperati dal disco con minori ritardi di accodamento, di posizionamento e di rotazione. Se un disco si guasta, l'altro disco viene usato finché il primo non è stato riparato. Si supponga che il tempo medio per la riparazione sia di 24 ore, allora il tempo medio per la perdita di dati in un sistema a disco riflesso (*mirrored disk system*) che usa 100 dischi con MTTF di 200.000 ore ciascuno è di  $(200.000)3/(2 * 24) = 8,33 * 108$  ore, cioè di 95.028 anni. La riflessione di disco raddoppia anche il tasso con il quale sono gestite le richieste di lettura, dal momento che una lettura può essere indirizzata all'uno o all'altro disco. Il tasso di trasferimento di ciascuna lettura, però, rimane uguale a quello che si ha per un singolo disco.

Un'altra soluzione al problema dell'affidabilità è quella di memorizzare informazioni supplementari che non sono normalmente necessarie ma che possono essere usate per ricostruire le informazioni perse in caso di guasto del disco. L'introduzione di ridondanze deve considerare due problemi: la scelta di una tecnica per calcolare l'informazione ridondante e la scelta di un metodo per distribuire l'informazione ridondante sul vettore di dischi. Si affronta il primo problema tramite l'uso di **codici a correzione d'errore** che utilizzano bit di parità, o codici specializzati come i codici Hamming. Con lo schema di parità si può considerare che un disco ridondante contenga la somma di tutti i dati presenti negli altri dischi. Quando un disco si guasta le informazioni mancanti possono essere ricostruite tramite un processo simile a una sottrazione. Per il secondo problema, i due approcci più importanti consistono nel memorizzare le informazioni ridondanti su un piccolo numero di dischi o nel distribuirle uniformemente su tutti i dischi. Quest'ultimo ha come risultato un miglior bilanciamento del carico. I diversi livelli di RAID scelgono una combinazione di queste opzioni per implementare la ridondanza, e perciò per migliorare l'affidabilità.

## MIGLIORAMENTO DELLE PRESTAZIONI:

I vettori di dischi si servono della tecnica del data striping per ottenere tassi di trasferimento migliori. Si noti che i dati possono essere letti o scritti solo un blocco alla volta. La suddivisione su disco può essere eseguita a un livello di granularità più fine, scomponendo un byte di dati in bit e distribuendo i bit su dischi diversi. Perciò il **data striping a livello di bit (bit-level data striping)** consiste nello spezzare un byte di dati e nello scrivere il bit  $j$  nel  $j$ -esimo disco. Con byte di 8 bit, otto dischi fisici possono essere considerati come un solo disco logico con un aumento di otto volte del tasso di trasferimento dati. Ogni disco partecipa a ciascuna richiesta di I/O e l'ammontare totale di dati letti per ogni richiesta è di otto volte i dati letti dal singolo disco. Lo striping a livello di bit può essere generalizzato a un numero di dischi che sia o un multiplo o un fattore di otto. Perciò in un vettore di quattro dischi il bit  $n$  va nel disco ( $n \bmod 4$ ). La granularità dell'interleaving dei dati può essere maggiore di un singolo bit, ad esempio, i blocchi di un file possono essere suddivisi su dischi diversi, dando origine allo **striping a livello di blocco (block-level striping)**. In Figura precedente è mostrato il data striping a livello di blocco supponendo che il file di dati contenga quattro blocchi. Con lo striping a livello di blocco, richieste indipendenti multiple che accedono a blocchi singoli (piccole richieste) possono essere servite in parallelo da dischi separati, diminuendo così il tempo di accodamento delle richieste di I/O. Le richieste che accedono a più blocchi (grandi richieste) possono essere parallele, riducendo così il loro tempo di risposta. In generale, più elevato è il numero di dischi in un vettore, più ne beneficiano le prestazioni potenziali. Però, supponendo che i guasti siano indipendenti, il vettore di dischi di 100 dischi ha nell'insieme un'affidabilità che è 1/100 di quella di un disco singolo. Perciò la ridondanza tramite codici a correzione d'errore e il mirroring di disco è necessaria per fornire affidabilità insieme con alte prestazioni.

## ORGANIZZAZIONI E LIVELLI DI RAID:

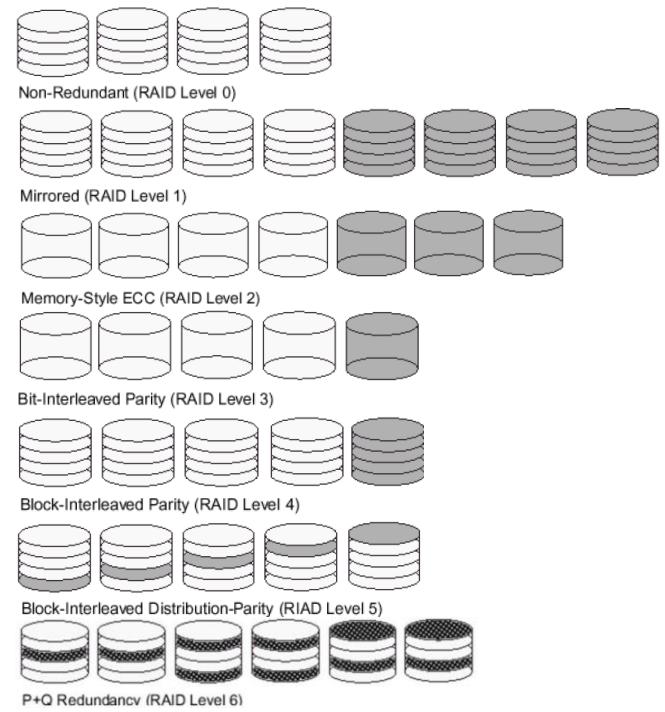
Sono state definite diverse organizzazioni RAID, basate su differenti combinazioni dei due fattori granularità dell'interleaving dei dati (*striping*) e modello usato per calcolare informazioni ridondanti. Nella proposta iniziale erano i RAID da 1 a 5, mentre 0 e 6 sono stati aggiunti in seguito.

Il livello RAID 0 non presenta dati ridondanti, e perciò ha le migliori prestazioni in scrittura, dal momento che gli aggiornamenti non devono essere duplicati. Però le sue prestazioni in lettura non sono così buone come per il livello RAID 1, che usa dischi riflessi. In quest'ultimo è possibile un incremento di prestazioni, pianificando una richiesta di lettura al disco che presenta il minore ritardo previsto di posizionamento e di rotazione. Il livello RAID 2 usa una ridondanza analoga a quella usata in memoria di lavoro tramite l'uso di codici Hamming, che contengono bit di parità per diversi sottoinsieme sovrapposti di componenti. Perciò, in una versione particolare di questo livello, tre dischi ridondanti sono sufficienti per quattro dischi originali, mentre con il mirroring, come nel livello 1, ne sarebbero stati richiesti quattro. Il livello 2 prevede sia la rivelazione sia la correzione d'errore, anche se la rivelazione non è in generale richiesta dato che i dischi rotti sono facilmente identificabili.

Il livello RAID 3 usa un solo disco di parità, facendo affidamento sul disk controller per riuscire a capire quale disco si è guastato. I livelli 4 e 5 usano il data striping a livello di blocco, con il livello 5 che distribuisce dati e informazioni di parità su tutti i dischi. Infine, il livello RAID 6 utilizza il cosiddetto schema di ridondanza *P+Q* che usa i codici di Reed e Solomon per proteggere fino a due guasti di disco usando solo due dischi ridondanti.

La ricostruzione in casi di guasto di disco è più facile per il livello RAID 1, mentre gli altri livelli richiedono la lettura di più dischi. Il livello 1 è usato per applicazioni critiche, come ad esempio per memorizzare log (registrazioni) di transazioni. I livelli 3 e 5 sono preferiti per la memorizzazioni di grandi quantità di dati, con il livello 3 che consente tassi di trasferimento maggiori.

I progettisti di un'organizzazione RAID per una data combinazione applicativa devono confrontare molte decisioni progettuali, come ad esempio il livello di RAID, il numero di dischi, la scelta di schemi di parità e il raggruppamento dei dischi per lo striping a livello di blocco. Dettagliati studi di prestazioni sono stati svolti su piccole letture e scritture (che si riferiscono a richieste di I/O per una sola unità di striping) e grandi letture e scritture (che si riferiscono a richieste di I/O per un'unità di striping da ciascun disco di un gruppo a correzione d'errore).



## 5. STRUTTURA DI INDICI PER I FILE

Esistono altre **strutture ausiliarie di accesso** chiamate **indici**, usate per velocizzare la ricerca dei record in risposta a determinate condizioni di ricerca. Gli indici forniscono **percorsi di accesso secondari**, che offrono metodi alternativi per accedere ai record senza influenzare la posizione fisica dei record sul disco. Permettono un accesso efficace sulla base di **campi d'indicizzazione** che vengono utilizzati per costruire l'indice. Qualsiasi campo del file può essere usato per creare un indice e sullo stesso file possono essere costruiti **più indici** su campi differenti.

### TIPI DI INDICI ORDINATI A UN SOLO LIVELLO:

La struttura di accesso degli **indici ordinati** è simile a quella dell'indice di un libro, il quale riporta in ordine alfabetico concetti fondamentali con indicazione del numero di pagina in cui compaiono. La ricerca nell'indice permette di consultare un elenco di **indirizzi** (numeri di pagina) e di utilizzare quest'ultimi per individuare un termine nel testo *cercandolo* alle pagine specificate. Se non ci fosse l'indice bisognerebbe esaminare tutto il volume, parola per parola, per trovare il termine a cui si è interessati, ma quest'operazione corrisponde a eseguire una ricerca lineare su un file.

Per un file con una data struttura di record che consiste di parecchi campi (o attributi), la struttura di accesso a indice di solito è definita su un unico campo, chiamato **campo di indicizzazione** (o **attributo di indicizzazione**). L'indice memorizza ogni valore del campo di indicizzazione corredandolo di un elenco di puntatori a tutti i blocchi del disco che contengono record con quel valore del campo. I valori dell'indice sono **ordinati** in modo che si possa eseguire una ricerca binaria. Il file dell'indice è molto più piccolo rispetto al file dei dati, quindi una ricerca binaria è ragionevolmente efficace. L'indicizzazione a più livelli evita la necessità di una ricerca binaria, ma richiede la creazione di ulteriori indici all'indice stesso.

Esistono molti tipi di indici ordinati, come l'**indice primario** che è specificato sul **campo chiave di ordinamento** di un file ordinato di record. Il campo chiave di ordinamento è utilizzato per **ordinare fisicamente** i record del file su disco e tutti i record hanno un **valore univoco** per quel campo. Se il campo di ordinamento non è un campo chiave, cioè se molti record nel file possono avere lo stesso valore del campo di ordinamento, può essere usato un altro tipo di indice, chiamato **indice di cluster**. Si noti che un file può avere al massimo un campo di ordinamento fisico, quindi può avere al massimo un indice primario oppure un indice di cluster, *ma non entrambi*. Su qualsiasi campo *non di ordinamento* di un file può essere specificato un terzo tipo di indice, detto **indice secondario**. Un file può avere molti indici secondari oltre al suo metodo di accesso primario.

### INDICI PRIMARI:

Un **indice primario** è un file ordinato i cui record sono di lunghezza fissa e sono costituiti da due campi. Il primo campo è dello stesso tipo di dati del campo chiave di ordinamento del file di dati, chiamato **chiave primaria**, e il secondo campo è un puntatore a un blocco del disco (indirizzo di blocco).

Esiste una **voce** (o **record dell'indice**) nel file dell'indice per ogni **blocco** nel file di dati. Ogni voce è composta da due campi che contengono il valore del campo della chiave primaria del *primo record* del blocco e un puntatore al corrispondente blocco su disco.

Nel seguito, si farà riferimento ai due valori della voce *i* come  $\langle K(i), P(i) \rangle$ .

Per creare l'indice primario sul file ordinato mostrato in Figura, si usa il campo NOME come chiave primaria, perché questo è il campo di ordinamento del file (dando per scontato che ciascun valore di NOME è unico). Ciascuna voce dell'indice contiene un valore NOME e un puntatore. Le prime tre voci dell'indice sono:

- $\langle K(1) = (\text{Aaron, Ed}), P(1) = \text{indirizzo del blocco 1} \rangle$
- $\langle K(2) = (\text{Adams, John}), P(2) = \text{indirizzo del blocco 2} \rangle$
- $\langle K(3) = (\text{Alexander, Ed}), P(3) = \text{indirizzo del blocco 3} \rangle$

Il numero complessivo di voci dell'indice è uguale al *numero di blocchi su disco* nel file di dati ordinato. Il primo record in ciascun blocco del file di dati è chiamato **record ancora** del blocco o semplicemente **punto ancora**.

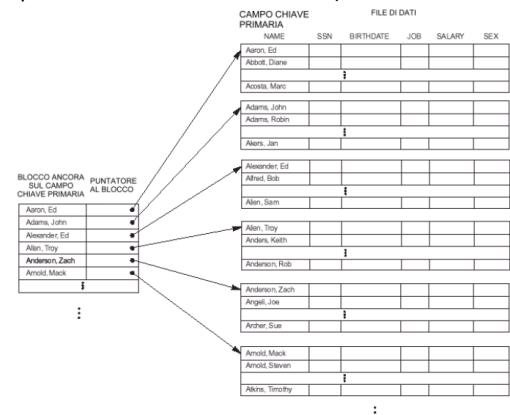
Gli indici di questo tipo possono essere un **indice denso**, che contiene una voce per ogni *valore della chiave di ricerca* (e quindi ogni record) nel file di dati, o un indice **sparso** (o **non denso**), che contiene voci solo per alcuni valori di ricerca. L'indice primario è quindi un indice non denso (sparso), poiché include una voce per ogni blocco del file di dati piuttosto che per ogni valore di ricerca o per ogni record.

Il file indice di un indice primario occupa meno blocchi rispetto al file dei dati per due motivi, il primo è che vi sono *meno voci* che record nel file di dati, il secondo è che ogni voce è di *dimensioni inferiori* rispetto a un record di dati perché ha solo due campi, di conseguenza un blocco su disco può contenere più voci dell'indice che record di dati. Una ricerca binaria nel file dell'indice richiede meno accessi al blocco rispetto ad una ricerca binaria nel file di dati. Un record il cui valore della chiave primaria è *K*, si trova nel blocco il cui indirizzo è *P(i)*, tale che  $K(i) \leq K < K(i+1)$ . Il blocco *i*-esimo del file di dati contiene tutti questi record a causa dell'ordinamento fisico dei suoi record rispetto al campo della chiave primaria. Per recuperare un record, dato il valore *K* della chiave primaria, si esegue una ricerca binaria nel file dell'indice per trovare la voce *i* corrispondente dell'indice e poi si prende il blocco del file di dati il cui indirizzo è *P(i)*.

#### ESEMPIO 1:

Si supponga di avere un file ordinato con  $r = 30.000$  record memorizzati su disco con dimensione del blocco  $B = 1024$  byte. I record del file hanno una dimensione fissa e sono indivisibili, con lunghezza del record  $R = 100$  byte. Il fattore di blocco del file risulta essere  $bfr = [(B/R)] = [(1024/100)] = 10$  record per blocco. Il numero di blocchi necessari per il file è  $b = [(r/bfr)] = [(30.000/10)] = 3000$  blocchi. Una ricerca binaria sul file di dati richiede approssimativamente  $[\log_2 b] = [\log_2 3000] = 12$  accessi ai blocchi.

Si supponga ora che il campo della chiave di ordinamento del file sia lungo  $V=9$  byte, un puntatore a blocco sia lungo  $P=6$  byte e si ipotizzi di aver costruito un indice primario per il file. La dimensione di ciascuna voce dell'indice è  $R_i = (9+6) = 15$  byte, quindi il fattore di blocco per l'indice è  $bfr_i = [(B/R_i)] = [(1024/15)] = 68$  voci per blocco. Il numero totale delle voci dell'indice  $r_i$  è uguale al numero di blocchi nel file di dati, che è 3000. Il numero di blocchi dell'indice, quindi, è  $b_i = [(r_i/bfr_i)] = [(3000/68)] = 45$  blocchi. Per eseguire una ricerca binaria nel file dell'indice sono necessari  $[(\log_2 b_i)] = [(\log_2 45)] = 6$  accessi ai blocchi. Per cercare un record usando l'indice è necessario un ulteriore accesso ai blocchi del file di dati, per un totale di  $6+1 = 7$  accessi, che corrisponde a un miglioramento rispetto alla ricerca binaria nel file di dati, che richiede 12 accessi.

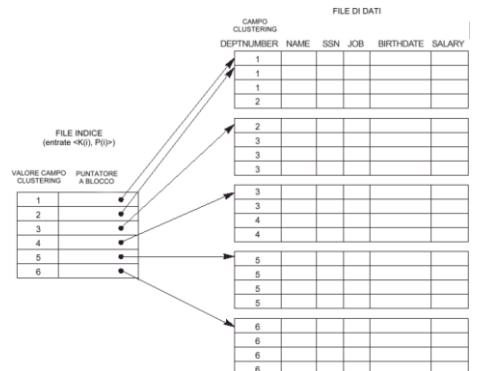


Un problema che si verifica con l'indice primario, come con qualsiasi file ordinato, è l'inserimento e l'eliminazione dei record. Se si tenta di inserire un record nella sua posizione corretta nel file di dati, si devono non solo spostare gli altri record per creare spazio per quello nuovo, ma anche cambiare alcune voci dell'indice, perché lo spostamento dei record modificherà i punti di aggancio di alcuni blocchi. Usando un file di overflow non ordinato si può limitare questo problema. Un'altra possibilità è utilizzare una lista di record di overflow per ciascun blocco del file di dati, simile al metodo di gestione dei record di overflow. I record all'interno di ogni blocco e la corrispondente lista di overflow possono essere ordinati per migliorare il tempo di recupero. L'eliminazione dei record è gestita utilizzando degli indicatori di eliminazione.

## INDICI DI CLUSTER:

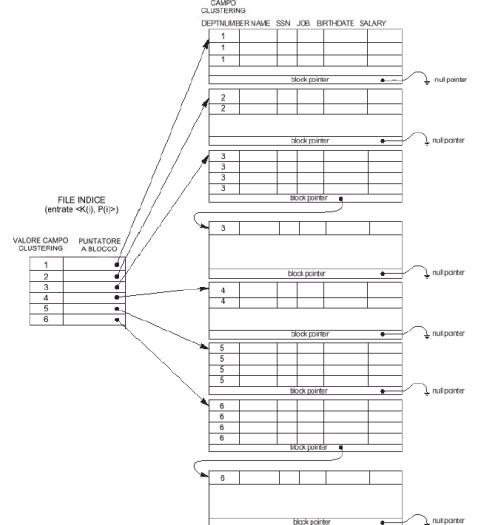
Se i record di un file sono ordinati fisicamente rispetto a un campo che non è chiave, cioè che *non* ha un valore distinto per ciascun record, quel campo è chiamato **campo di raggruppamento**. È possibile creare un apposito tipo di indice, detto **indice di cluster**, per velocizzare il recupero dei record che hanno lo stesso valore del campo di raggruppamento. Questo indice differisce da un indice primario, che richiede che il campo di ordinamento del file di dati abbia un *valore distinto* per ogni record.

Anche l'indice di cluster è un file ordinato con due campi: il primo è dello stesso tipo del campo di raggruppamento del file di dati, mentre il secondo è un puntatore a un blocco. Vi è una **voce** nell'indice di cluster per ogni *valore distinto* del campo di raggruppamento, la quale contiene il valore del campo e un puntatore al *primo blocco* nel file di dati che contiene un record con quel valore del campo di raggruppamento.



Un esempio nella prima Figura, si noti che l'inserimento e l'eliminazione dei record causano ancora dei problemi perché i record dei dati sono fisicamente ordinati. Per ridurre il problema dell'inserimento è possibile riservare un intero blocco (oppure un raggruppamento di blocchi contigui) per *ciascun valore* del campo di raggruppamento, tutti i record con quel valore del campo sono posti nel blocco (o raggruppamento di blocchi). Questo rende l'inserimento e l'eliminazione relativamente semplici (seconda Figura).

Un indice di cluster è un altro esempio di indice *non denso*, perché contiene una voce per ogni valore distinto del campo di indicizzazione piuttosto che per ogni record nel file. Un indice è qualcosa di simile alle strutture delle directory usate per l'hash estendibile. In entrambi i casi, infatti, si esegue una ricerca per trovare il puntatore al blocco di dati che contiene il record desiderato. La differenza principale è che la ricerca tramite indice utilizza i valori del campo di ricerca, mentre la ricerca delle directory hash usa un valore hash che è calcolato applicando la funzione hash al campo di ricerca.



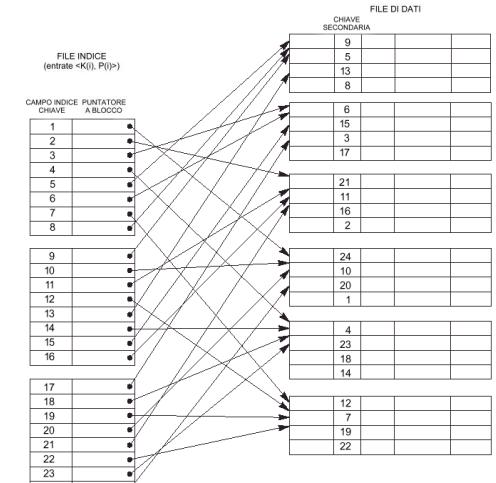
## INDICI SECONDARI:

Un **indice secondario** è anch'esso un file ordinato con due campi. Il primo campo è dello stesso tipo di dati di un campo del file di dati, che non viene utilizzato per effettuare l'ordinamento del file di dati, chiamato **campo di indicizzazione**. Il secondo campo è un puntatore a *blocco* oppure un puntatore a *record*. Vi possono essere *più* indici secondari (e quindi campi di indicizzazione) per lo stesso file.

Si consideri una struttura di accesso del tipo indice secondario su un campo chiave che un *valore distinto* per ciascun record. Un campo di questo tipo talvolta è chiamato **chiave secondaria**. In questo caso vi è una voce dell'indice per *ciascun record* del file di dati, la quale contiene il valore della chiave secondaria del record e un puntatore al blocco in cui il record è memorizzato oppure al record stesso. Questo indice, quindi, è **denso**.

Si indica nuovamente i valori dei due campi di una voce  $i$  con  $\langle K(i), P(i) \rangle$ . Le **voci** sono **ordinate** rispetto al valore di  $K(i)$ , quindi si può eseguire una ricerca binaria. Visto che i record del file di dati *non* sono fisicamente ordinati rispetto ai valori del campo della chiave secondaria, non è possibile utilizzare i **punti ancora** dei blocchi. Questo perché viene creata una voce dell'indice per ciascun record nel file di dati invece che per ogni blocco come nel caso di un indice primario. In Figura è illustrato un indice secondario in cui i puntatori  $P(i)$  nelle voci dell'indice sono *puntatori a blocco, non puntatori a record*. Una volta che il blocco appropriato è trasferito in memoria centrale, può essere eseguita la ricerca del record desiderato all'interno del blocco.

Un indice secondario ha bisogno di più spazio di memorizzazione e di un tempo di ricerca più lungo rispetto a un indice primario a causa del suo maggiore numero di voci. Il *miglioramento* nel tempo di ricerca di un record arbitrario, tuttavia, è decisamente maggiore per un indice secondario che per un indice primario, visto che se l'indice secondario non esistesse si dovrebbe svolgere una *ricerca lineare* nel file di dati. Nel caso dell'indice primario, invece, si potrebbe eseguire una ricerca binaria sul file principale, anche se l'indice non esistesse.



## ESEMPIO 2:

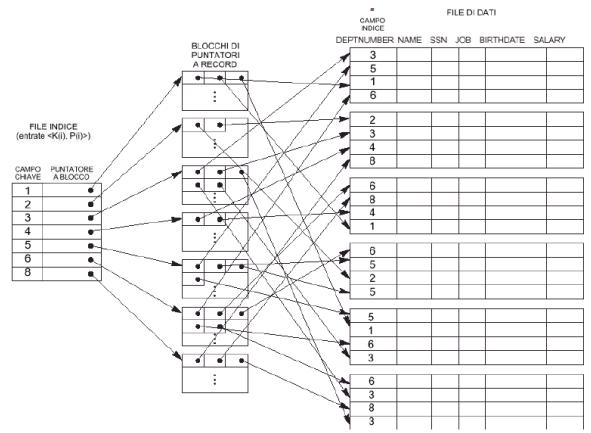
Si prenda in considerazione il file dell'Esempio 1 con  $r=30.000$  record di lunghezza fissa di dimensioni  $R=100$  byte memorizzati su un disco con dimensioni di blocco  $B=1024$  byte. Il file contiene  $b=3000$  blocchi, come calcolato nell'Esempio 1. Per eseguire una ricerca lineare sul file in media sarebbero necessari  $b/2 = 3000/2 = 1500$  accessi a blocco. Si supponga di creare un indice secondario su un campo chiave del file, sul quale non è stato effettuato un ordinamento e che è lungo  $V=9$  byte. Come nell'Esempio 1, il puntatore ai blocchi è lungo  $P=6$  byte, quindi ciascuna voce dell'indice è  $R_i = (9+6) = 15$  byte e il fattore di blocco per l'indice è  $bfr_i = [(B/R_i)] = [(1024/15)] = 68$  voci per blocco. In un indice secondario denso come questo, il numero totale di voci dell'indice  $r_i$  è uguale al *numero di record* nel file di dati che è 30.000. Il numero di blocchi necessari per l'indice è quindi  $b_i = [(r_i/bfr_i)] = [(30.000/68)] = 442$  blocchi.

Una ricerca binaria su questo indice secondario richiede  $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 442) \rceil = 9$  accessi di blocchi. Per cercare un record usando l'indice è indispensabile un ulteriore accesso a blocco del file di dati per un totale di  $9+1 = 10$  accessi, un enorme miglioramento rispetto ai 1500 accessi mediamente necessari per una ricerca lineare, ma leggermente peggiore dei sette accessi necessari usando l'indice primario.

Si può creare anche un indice secondario su un *campo non chiave* di un file. In questo caso numerosi record nel file di dati possono avere lo stesso valore del campo di indicizzazione. Vi sono varie tecniche per realizzare un indice di questo tipo:

- **Opzione 1:** inserire più voci di indice con lo stesso valore  $K(i)$ , uno per ciascun record: essa dà come risultato un indice denso;
- **Opzione 2:** usare record di lunghezza variabile per le voci dell'indice con un campo ricorrente per il puntatore. Nella voce dell'indice per  $K(i)$  viene tenuta una lista di puntatori  $\langle P(i,1), \dots, P(i,k) \rangle$ , la lista contiene un puntatore a ciascun blocco che ospita un record il cui valore del campo di indicizzazione è uguale a  $K(i)$ . Per questa tecnica e per la precedente l'algoritmo di ricerca binaria sull'indice deve essere fatto in modo adeguato;

- Opzione 3:** mantenere le voci dell'indice a una lunghezza fissa e avere una singola voce per ciascun *valore del campo di indicizzazione*, ma creare un ulteriore livello di gestione e accesso ai puntatori multipli. In questo schema non denso, il puntatore  $P(i)$  nella voce dell'indice  $\langle K(i), P(i) \rangle$  fa riferimento a un *blocco di puntatori ai record*, ogni puntatore ai record in quel blocco si riferisce a uno dei record di dati con valore  $K(i)$  per il campo di indicizzazione. Se troppi record condividono lo stesso valore  $K(i)$ , così che i loro puntatori ai record non possono essere contenuti in un singolo blocco del disco, viene utilizzato un agglomerato o una lista concatenata di blocchi. Si tratta della tecnica usata più comunemente (Figura). Il recupero attraverso l'indice richiede uno o più accessi a blocchi aggiuntivi a causa del livello extra, ma gli algoritmi per eseguire ricerche nell'indice e per inserire nuovi record nel file di dati sono semplici. Inoltre, le ricerche con complicate condizioni di selezione possono essere gestite facendo riferimento ai soli puntatori ai record, senza dover recuperare molti record di dati non necessari.



Si noti che un indice secondario fornisce un **ordinamento logico** dei record attraverso il campo di indicizzazione. Se si accede ai record secondo l'ordine delle voci dell'indice secondario, li si ottiene nell'ordine del corrispondente campo di indicizzazione.

In Tabella sono mostrate le caratteristiche del campo di indicizzazione di ogni tipo di indice ordinato a un solo livello

	Campo ordering	Campo non ordering
Campo chiave	Indice Primario	Indice Secondario (chiave)
Campo non chiave	Indice Clustering	Indice Secondario (non chiave)

In quest'altra Tabella vengono riepilogate le proprietà di ciascun tipo di indice confrontando il numero di voci dell'indice e specificando quali indici sono densi e quali utilizzano ancora ai blocchi del file di dati. Sì, se ogni valore distinto del campo di ordinamento è collocato all'inizio di un nuovo blocco; altrimenti no.

	Numero di entry di 1° livello	Denso o non denso	Ancoraggio dei blocchi del file dati
Primario	Numero di blocchi del file dati	Non denso	Si
Clustering	Numero di valori distinti del campo indexing	Non denso	Si/No*
Secondario (chiave)	Numero di record del file dati	Denso	No
Secondario (non chiave)	Num. record del file dati (caso 1) oppure num. valori distinti del campo indexing (caso 2 e 3)	Denso o non denso	No

#### INDICE MULTIVALORE:

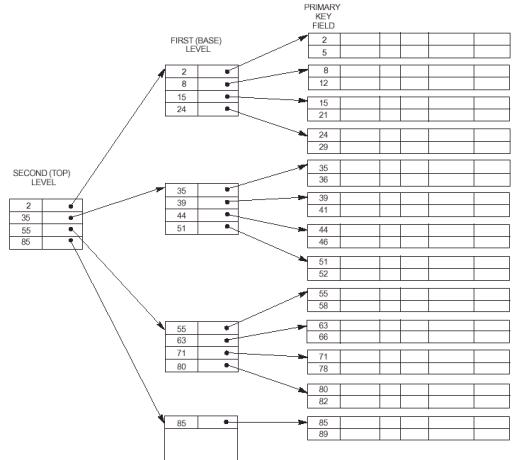
Gli schemi d'indicizzazione descritti finora richiedono che il file dell'indice sia ordinato. Si usa la ricerca binaria per individuare i puntatori a un blocco sul disco o a un record (o ai record) nel file che ha il valore specificato del campo di indicizzazione. La ricerca binaria richiede ( $\log_2 b_i$ ) accessi ai blocchi per un indice con  $b_i$  blocchi, perché ogni passo dell'algoritmo riduce la parte dell'indice in cui si deve continuare a cercare di un fattore pari a 2. Questo spiega perché si considera la funzione logaritmo in base 2. L'idea che sta alla base di un **indice multilivello** è ridurre a ogni passo la parte dell'indice in cui si continua a cercare di un fattore  $b_{fr_i}$ , il fattore di blocco dell'indice, che è maggiore di 2. Lo spazio di ricerca viene ridotto molto più velocemente. Il valore  $b_{fr_i}$  è chiamato il **fan-out** dell'indice multilivello e si farà riferimento ad esso col il simbolo **fo**. La ricerca in un indice multilivello richiede approssimativamente ( $\log_{fo} b_i$ ) accessi ai blocchi, che è un numero inferiore rispetto a quello della ricerca binaria se il fan-out è maggiore di 2.

Un indice multilivello considera il file dell'indice, al quale si fa riferimento ora come indice di primo livello (o livello base) di un indice multilivello, come un *file ordinato* con un *valore distinto* per ogni  $K(i)$ . A questo punto è possibile creare un indice primario per l'indice di primo livello, questo indice dell'indice di primo livello è detto **secondo livello** dell'indice multilivello. Poiché il secondo livello è un indice primario, si possono utilizzare i **punti ancora** dei blocchi in modo che il secondo livello contenga una voce per *ciascun blocco* del primo livello. Il fattore di blocco  $b_{fr_i}$ , per il secondo livello, e per tutti i livelli successivi, è lo stesso dell'indice di primo livello, perché tutte le voci dell'indice hanno la medesima dimensione, ciascuna ha un valore del campo e un indirizzo del blocco. Se il primo livello ha  $r_1$  voci e il fattore di blocco, che è anche il fan-out, è  $b_{fr_i} = fo$ , il primo livello necessita di  $[(r_1/fo)]$  blocchi, che è quindi il numero di voci  $r_2$  necessarie al secondo livello dell'indice. Si può ripetere questo procedimento per il secondo livello. Il **terzo livello**, che è un indice primario per l'indice di secondo livello, ha una voce per ogni blocco del secondo livello, il numero delle voci del terzo livello è quindi  $r_3 = [(r_2/fo)]$ . Si noti che è richiesto un secondo livello solo se il primo necessita di più di un blocco di disco e, in modo simile, è richiesto un terzo livello solo se il secondo occupa più di un blocco.

Si può ripetere il procedimento precedente finché tutte le voci di un livello  $t$  dell'indice possono essere contenute in un singolo blocco. Questo blocco di livello  $t$ -esimo viene chiamato **livello superiore** dell'indice. Ogni livello riduce il numero di voci del livello precedente di un fattore di  $fo$  (fan-out dell'indice) cosicché è possibile usare la formula  $1 \leq (r_t / (fo)^t)$  per calcolare  $t$ . Quindi un indice multilivello con  $r_1$  voci del primo livello avrà approssimativamente  $t$  livello, dove  $t = [\log_{fo}(r_1)]$ . Lo schema multilivello qui descritto può essere usato su qualsiasi tipo di indice, cioè primario, di cluster, o secondario, purché l'indice di primo livello abbia *valori distinti per  $K(i)$  e voci di lunghezza fissa*.

#### ESEMPIO 3:

Si supponga che l'indice secondario denso dell'Esempio 2 sia trasformato in un indice multilivello. Si è già calcolato il fattore di blocco dell'indice  $b_{fr_i}=68$  voci dell'indice per blocco, che è anche il fan-out  $fo$  per l'indice multilivello, è stato calcolato anche il numero di blocchi del primo livello  $b_1=442$ . Il numero di blocchi del secondo livello sarà  $b_2=[(b_1/fo)]=[(442/68)]=7$  blocchi e il numero di blocchi del terzo livello sarà  $b_3=[(b_2/fo)]=[(7/68)]=1$  blocco. Il terzo livello, quindi, è il livello superiore dell'indice e  $t=3$ . Per accedere a un record eseguendo una ricerca nell'indice multilivello, si deve accedere a un blocco di ciascun livello più al blocco del file di dati, quindi servono  $t+1 = 3+1 = 4$  accessi a blocchi. Si confronti questo risultato con quello dell'Esempio 2, in cui erano necessari 10 accessi dei blocchi poiché venivano usati un indice a un solo livello e la ricerca binaria.

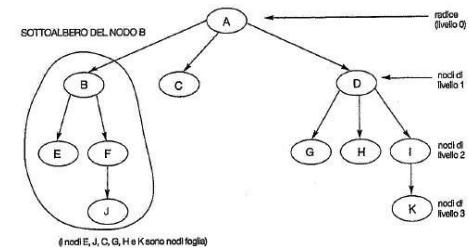


Come si è visto, un indice multilivello riduce il numero degli accessi a blocchi durante la ricerca di un record, per un dato valore del campo di indicizzazione. Occorre ancora affrontare i problemi di gestione delle eliminazioni e degli inserimenti nell'indice, perché tutti i livelli dell'indice sono *file fisicamente ordinati*. Per mantenere i vantaggi dell'utilizzo dell'indicizzazione multilivello senza incorrere nei problemi di eliminazione e di inserimento, i progettisti hanno adottato un tipo di indice multilivello che lascia dello spazio libero in ciascuno dei suoi blocchi per l'immissione di nuove voci. Questo è detto **indice dinamico multilivello** ed è spesso implementato usando le strutture di dati chiamate alberi B e alberi B<sup>+</sup>.

## INDICI DINAMICI MULTILIVELLO IMPLEMENTATI DA ALBERI B E ALBERI B<sup>+</sup>:

Gli alberi B e gli alberi B<sup>+</sup> sono casi speciali delle strutture dati albero. Un **albero** è formato da **nodi**, ognuno di essi ha un **padre** e zero o più nodi **figlio**, tranne un nodo chiamata **radice** che non ha il nodo padre e un nodo che non ha alcun nodo figlio è chiamato **foglia**, mentre un nodo che non è foglia è un **nodo interno**. Il **livello** di un nodo è sempre superiore di una unità rispetto al livello del suo nodo padre, mentre il livello del nodo radice è zero. Un **sottoalbero** di un nodo è costituito da quel nodo e da tutti i suoi nodi discendenti. Una precisa definizione ricorsiva di un sottoalbero dice che un sottoalbero è costituito da un nodo  $n$  e dai sottoalberi di tutti i nodi figli  $n$ .

In Figura è illustrata una struttura di dati ad albero: il nodo radice è A e i suoi nodi figli sono B, C e D, i nodi E, J, C, G, H e K sono nodi foglia. Un modo per implementare un albero è inserire in ciascun nodo tanti puntatori quanti sono i nodi figli di quel nodo. In alcuni casi in ciascun nodo è anche memorizzato un puntatore al padre. Oltre ai puntatori, un nodo di solito contiene qualche tipo di informazioni. Quando un indice multilivello viene implementato come albero, le informazioni includono i valori del campo di indicizzazione usati per guidare la ricerca di un particolare record.



## ALBERI DI RICERCA:

Un **albero di ricerca** è un tipo di albero speciale che viene usato per guidare la ricerca di un record, dato il valore di uno dei suoi campi. Gli indici multilivello possono essere pensati come una variante di un albero di ricerca: ogni nodo dell'indice multilivello può avere fino a  $f_o$  puntatori e  $f_o$  valori chiave, dove  $f_o$  è il fan-out dell'indice. I valori dei campi dell'indice di ciascun nodo guidano l'utente al nodo successivo finché raggiunge il blocco del file di dati che contiene i record richiesti. Seguendo un puntatore si limita a ogni passo la ricerca a un sottoalbero dell'albero di ricerca e si ignorano tutti i nodi che non sono di quel sottoalbero. Un **albero di ricerca** di ordine  $p$  è un albero tale che ogni suo nodo contiene al massimo  $p-1$  valori di ricerca e i  $p$  puntatori sono nell'ordine  $\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle$ , in cui  $q \leq p$ , ogni  $P_i$  è un puntatore a un nodo figlio (o puntatore vuoto) e ogni  $K_i$  è un valore di ricerca preso da un insieme ordinato di valori. Si suppone che tutti i valori di ricerca siano univoci.

In Figura viene mostrato un nodo di un albero di ricerca. Due vincoli devono sempre essere rispettati in un albero di ricerca:

1. All'interno di ciascun nodo,  $K_1 < K_2 < \dots < K_{q-1}$ ;
2. Per tutti i valori  $X$  del sottoalberi ai quali si fa riferimento da  $P_i$ , si ha  $K_{i-1} < X < K_i$  per  $1 < i < q$ ,  $X < K_i$  per  $i=1$  e  $K_{i-1} < X$  per  $i=q$ .

Ogni volta che si cerca un valore  $X$ , si segue il puntatore appropriato  $P_i$  secondo le formule esposte nel punto 2. In Figura è rappresentato un albero di ricerca di ordine  $p=3$ , i cui valori di ricerca sono interi. Si noti che alcuni dei puntatori  $P_i$  posti in un nodo possono essere puntatori nulli.

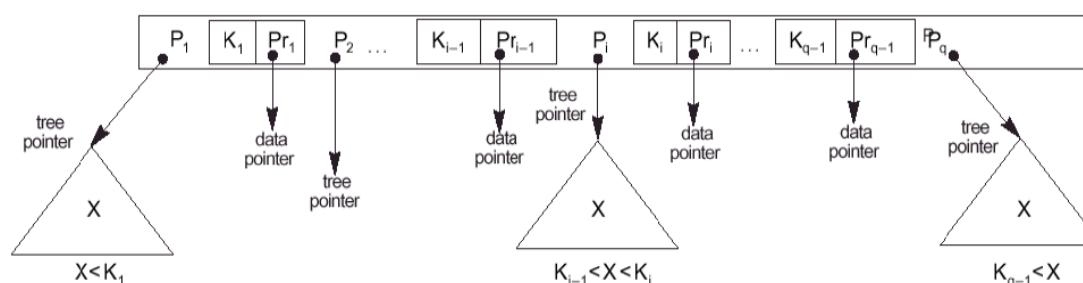
Si può usare un albero di ricerca come meccanismo per ricercare i record immagazzinati in un file su disco. I valori nell'albero possono essere i valori di uno dei campi del file chiamato **campo di ricerca** (o campo di indicizzazione se si usa indice multilivello).

Ogni valore chiave dell'albero è associato a un puntatore al record nel file di dati che ha quel valore. In alternativa il puntatore può puntare al blocco del disco che contiene quel record. Lo stesso albero di ricerca può essere memorizzato su disco assegnando ogni nodo dell'albero a un blocco del disco. Quando viene inserito un nuovo record si deve aggiornare l'albero di ricerca inserendo nell'albero una voce contenente il valore del campo di ricerca del nuovo record e un puntatore al nuovo record. Nuovi algoritmi sono necessari per inserire e cancellare i valori di ricerca, mentre si continuano a rispettare i due vincoli. Questi algoritmi non garantiscono che un albero di ricerca sia **bilanciato**, cioè che tutti i suoi nodi foglia siano allo stesso livello. L'albero in Figura a inizio pagina non è bilanciato perché ha nodi foglia ai livelli 1, 2 e 3. Mantenere un albero di ricerca bilanciato è importante, perché ciò garantisce che nessun nodo si troverà a livelli molto alti e quindi richiederà molti accessi ai blocchi durante la ricerca. Un altro problema con gli alberi di ricerca è che l'eliminazione dei record possa lasciare alcuni nodi nell'albero quasi vuoti, sprecando quindi spazio per la memorizzazione e aumentando il numero dei livelli. La tecnica degli alberi B affronta entrambi questi problemi specificando ulteriori vincoli sull'albero di ricerca.

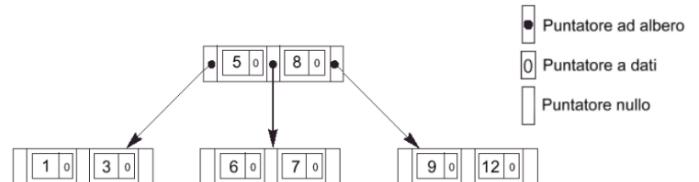
## ALBERI B:

L'**albero B** soddisfa ulteriori vincoli che assicurano che l'albero sia sempre bilanciato e che lo spazio sprecato a seguito di cancellazione non diventi mai eccessivo. Gli algoritmi per l'inserimento e la cancellazione diventano più complessi allo scopo di soddisfare questi vincoli. Ciononostante, gli inserimenti e le cancellazioni sono processi semplici, diventano complicati solo in particolari circostanze, cioè ogni volta che si tenta un inserimento in un nodo che è già completo oppure una cancellazione da un nodo che così diventa più della metà vuoto. In modo più formale, un **albero B di ordine  $p$** , quando è usato come una struttura di accesso su un **campo chiave** per cercare i record in un file di dati, può essere definito nel seguente modo:

1. Ogni nodo interno dell'albero B ha la forma:  $\langle P_1, \langle K_1, Pr_1 \rangle, P_2, \langle K_2, Pr_2 \rangle, \dots, P_{q-1}, \langle K_{q-1}, Pr_{q-1} \rangle, P_q \rangle$ , dove  $q \leq p$ . Ogni  $P_i$  è un **puntatore a un albero**, cioè un puntatore a un altro nodo nell'albero B. Ogni  $Pr_i$  è un **puntatore ai dati**, cioè un puntatore al record il cui valore del campo chiave è uguale a  $K_i$  (o al blocco del file di dati che contiene quel record);
2. All'interno di ogni nodo,  $K_1 < K_2 < \dots < K_{q-1}$ ;
3. Per tutti i valori  $X$  del campo chiave di ricerca nel sottoalbero a cui si è fatto riferimento da  $P_i$ , si ha:  $K_{i-1} < X < K_i$  per  $1 < i < q$ ,  $X < K_i$  per  $i=1$ ,  $K_{i-1} < X$  per  $i=q$ ;
4. Ogni nodo contiene al massimo  $p$  puntatori dell'albero;
5. Ogni nodo, tranne i nodi radice e i nodi foglia, ha almeno  $\lceil (p/2) \rceil$  puntatori dell'albero, il nodo radice ha almeno due puntatori dell'albero a meno che sia l'unico nodo dell'albero;
6. Un nodo con  $q$  puntatori dell'albero, dove  $q \leq p$ , contiene  $q-1$  valori del campo chiave di ricerca (e quindi ha  $q-1$  puntatori ai dati);
7. Tutti i nodi foglia sono allo stesso livello e hanno la medesima struttura dei nodi interni, a parte il fatto che tutti i loro **puntatori a un albero**  $P_i$  sono nulli.



In Figura è illustrato un albero B di ordine  $p=3$ . Si noti che tutti i valori di ricerca  $K$  di un albero B sono univoci perché si suppone che l'albero sia usato come una struttura di accesso su un campo chiave. Se si usa un albero B su un campo non chiave, si deve cambiare la definizione dei puntatori al file  $Pr_i$ , perché i puntatori  $Pr_i$  devono far riferimento a un blocco o raggruppamento di blocchi, che contiene i puntatori ai record del file. Questo ulteriore livello di indici è simile all'opzione 3 presentata per gli indici secondari.



Un albero B inizia con un solo nodo radice (è anche foglia) a livello 0. Quando il nodo radice diventa completo e con  $p-1$  valori della chiave di ricerca e si tenta di inserire un'altra voce nell'albero, il nodo radice si divide in due nodi di livello 1. Solo il valore centrale è tenuto nel nodo radice, mentre il resto dei valori sono divisi equamente tra gli altri due nodi. Quando uno dei nodi non radice è completo e una nuova voce dovrebbe esservi inserita, il nodo è diviso in due nodi dello stesso livello e la voce centrale è spostata verso il nodo padre insieme ai due puntatori ai nuovi nodi ottenuti dalla divisione. Se il nodo padre è completo viene diviso anch'esso. La divisione può propagarsi per tutto il tragitto verso il nodo radice, creando un nuovo livello se anche la radice deve essere divisa. Se la cancellazione di un valore fa sì che un nodo sia completo per meno della metà, viene fuso con i suoi nodi vicini, anche questa fusione può propagarsi lungo tutto il tragitto verso la radice. La cancellazione può ridurre il numero dei livelli dell'albero. È stato mostrato attraverso che, dopo numerosi inserimenti e cancellazione casuali in un albero B, i nodi sono approssimativamente completi al 69% quando il numero dei valori nell'albero si stabilizza. Questo vale anche per gli alberi B+. Se ciò avviene, la divisione e l'unione dei nodi si verificano solo raramente, quindi l'inserimento e la cancellazione diventano abbastanza efficaci. Se il numero dei valori aumenta ancora, l'albero si espanderà senza problemi, anche se si può verificare la divisione dei nodi e così alcuni inserimenti richiederanno più tempo.

L'Esempio 4 mostra come si effettua il **calcolo dell'ordine  $p$**  di un albero B memorizzato su disco.

#### ESEMPIO 4:

Si supponga che il campo di ricerca sia lungo  $V = 9$  byte, la dimensione del blocco del disco sia  $B = 512$  byte, il puntatore a un record (di dati) sia  $P_r=7$  byte e il puntatore a un blocco sia  $P = 6$  byte. Ogni nodo dell'albero B può avere al massimo  $p$  puntatori dell'albero,  $p-1$  puntatori ai dati e  $p-1$  valori del campo della chiave di ricerca. Questi devono essere contenuti in un singolo blocco del disco, perché ciascun nodo dell'albero B deve corrispondere a un blocco del disco. Quindi si deve avere:

$$(p*P) + ((p-1)*(P_r+V)) \leq B \quad \rightarrow \quad (p*6) + ((p-1)*(7+9)) \leq 512 \quad \rightarrow \quad (22 * p) \leq 528 \quad \rightarrow \quad p \leq 24$$

Il valore  $p$  deve soddisfare la diseguaglianza precedente, il che dà  $p = 23$  ( $p = 24$  non viene scelto per motivi forniti in seguito).

In generale un nodo dell'albero B può contenere ulteriori informazioni necessarie per gli algoritmi che manipolano l'albero, ad esempio il numero di voci  $q$  nel nodo e un puntatore al nodo padre. Prima di eseguire il calcolo precedente per  $p$ , quindi, si dovrebbe ridurre la dimensione del blocco della quantità di spazio necessario per tutte queste informazioni.

Nel seguito si illustra come calcolare il **numero di blocchi e di livelli per un albero B**.

#### ESEMPIO 5:

Si supponga che il campo di ricerca dell'Esempio 4 sia un campo chiave ma non di ordinamento e che sia necessario costruire un albero B su questo campo. Si ipotizzi che ciascun nodo dell'albero B sia completo al 69%. Ogni nodo, in media, avrà  $p*0,69 = 23*0,69 = 16$  puntatori e, quindi, 15 valori della chiave di ricerca. Il **fan-out medio** è  $fo = 16$ . Si può iniziare dalla radice e vedere quanti valori e puntatori possono esistere in media a ogni livello successivo:

Radice:	1 nodo	15 entry	16 ptr
livello 1:	16 nodi	240 (16*15) entry	256 (16*16) ptr
livello 2:	256 nodi	3840 (256*15) entry	4096 ptr
livello 3:	4096 nodi	61440 (4096 *15) entry	-

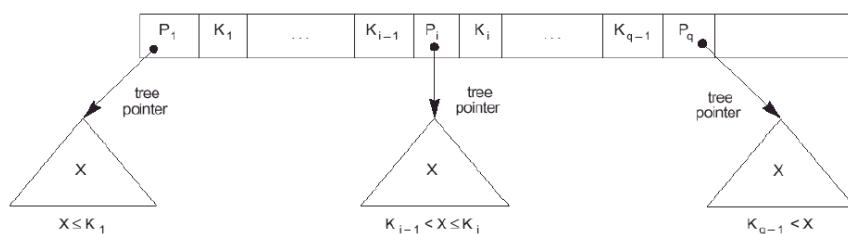
A ciascun livello si è calcolato il numero di voci moltiplicando il numero totale dei puntatori del livello precedente per 15, il numero medi di voci in ciascun nodo. Quindi, per le date dimensioni del blocco, dimensione del puntatore e dimensione del campo della chiave di ricerca, un albero B a due livelli contiene  $3840 + 240 + 15 = 4095$  voci in medi, un albero B a tre livelli contiene 65.535 voci in media.

Gli alberi B sono usati talvolta come organizzazione del file di dati. In questo caso tutti i record sono memorizzati all'interno dei nodi dell'albero B invece che solo le voci <chiave di ricerca, puntatore del record>. Questo funziona bene per i file con un *numero relativamente piccolo di record* e una *dimensione ridotta del record*, altrimenti il fan-out e il numero di livelli diventano troppo grandi per permettere un accesso efficace.

Riassumendo, gli alberi B forniscono una struttura di accesso multilivello che produce un albero bilanciato in cui ogni nodo è riempito almeno per metà. Ciascun nodo di un albero B di ordine  $p$  può avere al massimo  $p-1$  valori di ricerca.

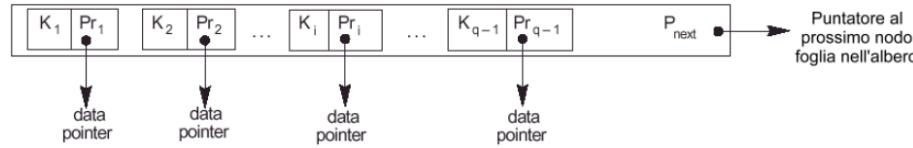
#### ALBERI B<sup>+</sup>:

Gli **alberi B<sup>+</sup>** sono una variante degli alberi B in cui i puntatori ai blocchi dei dati di un file sono memorizzati solo nei nodi foglia, questo può richiedere meno livelli e permette di ottenere indici di capacità più elevata. La maggior parte delle odierne implementazioni di **indici multilivello dinamici** utilizza questa variante della struttura di dati ad albero. In un **albero B**, tutti i valori del campo di ricerca appaiono una volta a un dato livello nell'albero, insieme a un puntatore ai dati. In un **albero B<sup>+</sup>** i puntatori ai dati sono memorizzati **solo nei nodi foglia dell'albero**, quindi la struttura dei nodi foglia differisce dalla struttura dei nodi interni. I nodi foglia contengono una voce per *ogni* valore del campo di ricerca, insieme a un puntatore al record dei dati (o al blocco che contiene il record) se il campo di ricerca è un campo chiave. Per un campo di ricerca non chiave, il puntatore fa riferimento a un blocco che contiene i puntatori ai record del file di dati, creando un livello ulteriore di indici. I nodi foglia dell'albero B<sup>+</sup> di solito sono collegati tra loro per fornire un accesso ordinato al campo di ricerca e ai record. Questi nodi foglia sono simili al primo livello (base) di un indice. I nodi interni dell'albero B<sup>+</sup> corrispondono agli altri livelli di un indice multilivello. Alcuni valori del campo di ricerca dai nodi foglia sono ripetuti nei nodi interni dell'albero B<sup>+</sup> per guidare la ricerca.



La struttura dei **nodi interni** di un albero B<sup>+</sup> di ordine  $p$  (Figura sopra) è la seguente:

1. Ogni nodo interno ha la seguente forma:  $\langle P_1, K_1, P_2, K_2, \dots, P_{q-1}, K_{q-1}, P_q \rangle$ , in cui  $q \leq p$  e ogni  $P_i$  è un **puntatore a un albero**;
2. All'interno di ogni nodo interno,  $K_1 < K_2 < \dots < K_{q-1}$ ;
3. Per tutti i valori  $X$  del campo di ricerca nel sottoalbero a cui si fa riferimento da  $P_i$ , si ha  $K_{i-1} < X \leq K_i$  per  $1 < i < q$ ,  $X \leq K_i$  per  $i=1$ ,  $K_{i-1} < X$  per  $i=q$ ;
4. Ogni nodo interno ha al massimo  $p$  puntatori ad un albero;
5. Ogni nodo interno, tranne la radice, ha almeno  $\lceil (p/2) \rceil$  puntatori dell'albero. Il nodo radice ha almeno due puntatori dell'albero se è un nodo interno;
6. Un nodo interno con  $q$  puntatori,  $q \leq p$ , ha  $q - 1$  valori del campo di ricerca.



La struttura dei **nodi esterni** di un albero B<sup>+</sup> dell'ordine  $p$  (Figura sopra) è la seguente:

1. Ogni nodo foglia ha la forma:  $\langle \langle K_1, Pr_1 \rangle, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_{next} \rangle$ , in cui  $q \leq p$ , ogni  $Pr_i$  è un puntatore ai dati e  $P_{next}$  si riferisce al **nodo foglia successivo** dell'albero B<sup>+</sup>;
2. All'interno di ogni nodo foglia,  $K_1 < K_2 < K_{q-1}$ , con  $q \leq p$ ;
3. Ciascun  $Pr_i$  è un **puntatore ai dati** che fa riferimento al record il cui valore del campo di ricerca è  $K_i$  o a un blocco del file che contiene il record (o a un blocco di puntatori a record che fa riferimento ai record il cui valore del campo di ricerca è  $K_i$  se il campo di ricerca non è una chiave);
4. Ogni nodo foglia ha almeno  $\lceil (p/2) \rceil$  valori;
5. Tutti i nodi foglia sono allo stesso livello.

I puntatori nei nodi interni sono **puntatori a un albero** di blocchi che sono nodi dell'albero, mentre i puntatori nei nodi foglia sono **puntatori ai dati** che fanno riferimenti ai record o ai blocchi del file di dati, a eccezione del puntatore  $P_{next}$  che è un puntatore a un albero del successivo nodo esterno. Iniziando dal nodo foglia più a sinistra è possibile attraversare i nodi foglia come se fossero una lista concatenata, usando i puntatori  $P_{next}$ . Questo fornisce un accesso ordinato ai record di dati basato sul campo di indicizzazione. Può essere incluso anche un puntatore  $P_{prev}$ . Per un albero B<sup>+</sup> su un campo non chiave, è necessario un livello ulteriore di indici (come quello mostrato in Figura dell'indice secondario opzione 3), così che i puntatori  $Pr$  sono puntatori ai blocchi che contengono un insieme di puntatori ai record effettivi del file di dati.

Poiché le voci nei *nodi interni* di un albero B<sup>+</sup> includono valori di ricerca e puntatori a un albero senza alcun puntatore ai dati, un nodo interno può contenere più voci del nodo corrispondente di un albero B. A parità di dimensione del blocco (nodo), l'ordine  $p$  sarà più grande per un albero B<sup>+</sup> rispetto a un albero B, come illustrato nell'Esempio 6. Questo fa sì che l'albero B<sup>+</sup> abbia meno livelli, migliorando il tempo di ricerca. Poiché le strutture dei nodi interni e foglia di un albero B<sup>+</sup> sono diverse, l'ordine  $p$  può essere differente nei due casi. Si userà  $p$  per denotare l'ordine dei *nodi interni* e  $p_{foglia}$  per denotare l'ordine dei *nodi foglia* e definito come il numero massimo dei puntatori ai dati presenti in un nodo foglia.

#### ESEMPIO 6:

Per calcolare l'ordine  $p$  di un albero B<sup>+</sup> si supponga che la lunghezza del campo della chiave di ricerca sia  $V = 9$  byte, la dimensione del blocco sia  $B = 512$  byte, il puntatore a un record sia  $P_r = 7$  byte e il puntatore a un blocco sia  $P = 6$  byte, come nell'Esempio 4. Un nodo interno dell'albero B<sup>+</sup> può avere fino a  $p$  puntatori a un albero e  $p - 1$  valori del campo di ricerca, questi devono essere contenuti in un singolo blocco. Quindi si avrà:

$$(p*P) + ((p-1)*V) \leq B \rightarrow (p*6) + ((p-1)*9) \leq 512 \rightarrow (15 * p) \leq 512$$

Si può scegliere che  $p$  sia il valore più grande che soddisfa la diseguaglianza precedente, il che dà  $p = 34$ . Ciò è maggiore rispetto al valore 23 calcolato per l'albero B e dà come risultato un fan-out più ampio e un maggior numero di voci in ogni nodo interno di un albero B<sup>+</sup> rispetto al corrispondente albero B. I nodi foglia dell'albero B<sup>+</sup> avranno lo stesso numero di valori e puntatori, a eccezione del fatto che i puntatori sono puntatori ai dati, e un puntatore al nodo successivo. Quindi l'ordine  $p_{foglia}$  per i nodi foglia può essere calcolato nel modo seguente:

$$(P_{foglia} * (Pr + V)) + P (\text{per } P_{next}) \leq B \rightarrow (P_{foglia} * (7 + 9)) + 6 \leq 512 \rightarrow (P_{foglia} * 16) \leq 506$$

Ciascun nodo foglia può quindi contenere fino a  $p_{foglia} = 31$  combinazioni di puntatori ai dati o valori chiave, dando per scontato che i puntatori ai dati siano i puntatori a record.

Anche nel caso dell'albero B<sup>+</sup>, può essere necessario che ogni nodo contenga ulteriori informazioni per implementare gli algoritmi d'inserimento e di cancellazione. Queste informazioni possono comprendere il tipo di nodo (interno o foglia), il numero di voci  $q$  correnti del nodo e i puntatori ai nodi padre e fratelli. Prima di eseguire i calcoli precedenti per  $p$  e  $p_{foglia}$ , quindi, si dovrebbe ridurre la dimensione del blocco della quantità di spazio necessario per contenere tutte queste informazioni. L'esempio successivo mostra come si calcola il numero di voci di un albero B<sup>+</sup>.

#### ESEMPIO 7:

Si supponga di costruire un albero B<sup>+</sup> sul campo dell'Esempio 6. Per calcolare il numero approssimativo di voci nell'albero B<sup>+</sup> si supponga che ogni nodo sia completo al 69%. In media ciascun nodo interno avrà  $34 * 0,69$  o approssimativamente 23 puntatori e quindi 22 valori. Ogni nodo foglia in media conterrà  $0,69 * p_{foglia} = 0,69 * 31$  o approssimativamente 21 puntatori ai record di dati. L'albero B<sup>+</sup> avrà il seguente numero medio di voci a ogni livello:

Radice:	1 nodo	22 entry	23 ptr
livello 1:	23 nodi	506 (22*23) entry	529 (23*23) ptr
livello 2:	529 nodi	11638 (22*529) entry	12167 ptr
livello foglie:	12167 nodi	255507 (21 * 12167) puntatori a record	

Per la dimensione del blocco, la dimensione del puntatore e la dimensione del campo di ricerca indicati precedentemente, l'albero B<sup>+</sup> a tre livelli contiene in media fino a 255.507 puntatori ai record. Si confronti questo valore con le 65.535 voci necessarie per il corrispondente albero B dell'Esempio 5.

## INSERIMENTO CON ALBERI B<sup>+</sup>:

Si vogliono inserire dei record in un albero B<sup>+</sup> di ordine  $p=3$  e  $p_{\text{foglia}}=2$ , nella sequenza: 8, 5, 1, 7, 3, 12, 9, 6.

L'inserimento dei valori 8 e 5 non provoca overflow: sono entrambi inseriti nella radice.

L'inserimento di 1 provoca overflow. Il nodo è scisso ed il valore centrale è ripetuto in un nuovo nodo radice. Alcuni valori sono replicati nei nodi interni per guidare la ricerca.

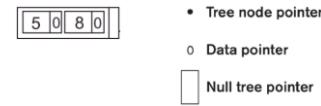
L'inserimento di 7 non provoca overflow. Si noti che tutti i valori sono a livello foglia, perché i puntatori ai dati sono tutti a quel livello.

L'inserimento di 3 provoca overflow nel nodo (1, 5), che viene scisso, propagando il valore 3 nel nodo padre. Si noti che un valore che compare nel nodo interno, compare anche come valore più a destra del sottoalbero referenziato dal puntatore alla sinistra di tale valore.

L'inserimento di 12 provoca un overflow nel nodo (7, 8), che viene scisso, propagando il valore 8 nel nodo padre. La scissione si propaga fino alla radice, creando un nuovo livello nell'albero B<sup>+</sup>.

L'inserimento del 9 non provoca overflow.

L'inserimento di 6 provoca un overflow nel nodo (7, 8), che viene scisso, propagando il valore 7 nel nodo padre.



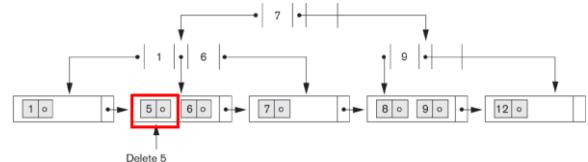
Prima di tutto si osservi che la radice è l'unico nodo nell'albero, quindi è anche una foglia. Non appena si crea più di un livello, l'albero viene diviso in nodi interni e foglia. Si noti che ***ogni valore chiave deve esistere a livello delle foglie***, perché tutti i puntatori ai dati si trovano a livello delle foglie. Tuttavia, solo alcuni valori sono presenti nei nodi interni per guidare la ricerca. Si consideri anche che ogni valore che appare in un nodo interno compare anche come *il valore più a destra* tra i nodi foglia del sottoalbero a cui fa riferimento il puntatore all'albero posto a sinistra del valore. Quando un *nodo foglia* è completo e viene inserita una nuova voce, il nodo **trabocca** (overflow) e il suo contenuto viene diviso in due. Le prime  $j=[(p_{\text{foglia}}+1)/2]$  voci del nodo originale sono mantenute in questo stesso nodo, mentre le restanti voci sono spostate in un nuovo nodo foglia. Il valore di ricerca  $j$ -esimo è replicato nel nodo interno padre e un ulteriore puntatore al nuovo nodo è creato nel padre. Questi valori devono essere inseriti nel nodo padre nella sequenza corretta. Se il nodo interno padre è completo, il nuovo valore gli causerà un trabocco, quindi anch'esso dovrà essere diviso in due. Le voci nel nodo interno fino a  $P_j$ , il puntatore a un albero  $j$ -esimo dopo l'inserimento del nuovo valore e del puntatore, con  $j=[(p+1)/2]$ , sono mantenuti, mentre il valore di ricerca  $j$ -esimo è **spostato** nel padre, ma non ripetuto. Un nuovo nodo interno conterrà le voci rimanenti da  $P_{j+1}$  in poi. Questa divisione può propagarsi lungo tutto il tragitto fino a creare un nuovo nodo radice e quindi un nuovo livello dell'albero B<sup>+</sup>.

## CANCELLAZIONE CON ALBERI B<sup>+</sup>:

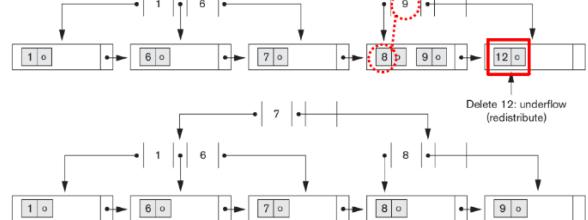
Dato il seguente albero B<sup>+</sup>, di ordine  $p=3$  e  $p_{\text{foglia}}=2$ , si vuol cancellare i record

5, 12, 9 e 6.

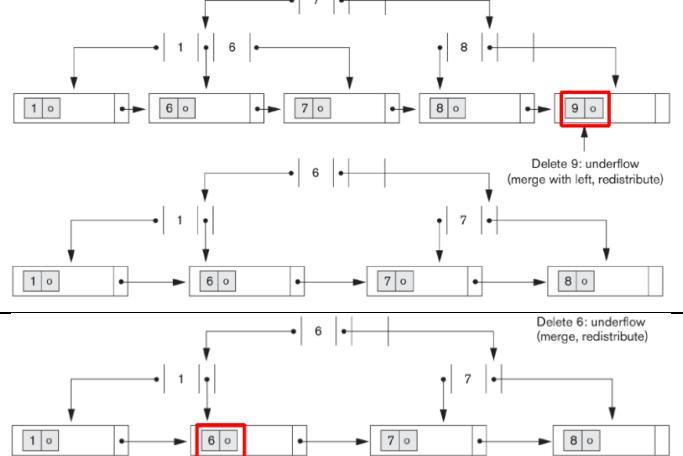
La cancellazione di 5 non pone alcun problema.



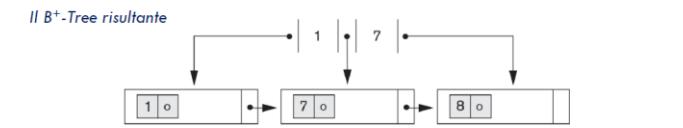
La cancellazione di 12 viene risolta con una ridistribuzione, e l'aggiornamento del valore del nodo interno da 9 a 8.



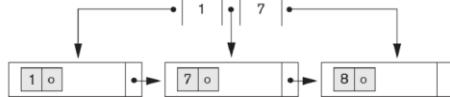
La cancellazione di 9 causa un underflow. I puntatori del nodo padre vengono redistribuiti con quelli del fratello sinistro.



La cancellazione di 6 causa un underflow. La redistribuzione dei nodi utilizzando i fratelli a destra e a sinistra causa un altro underflow nel nodo padre, che causa un underflow nel nodo radice, portando ad una riduzione del numero di livelli.



Il B<sup>+</sup>-Tree risultante



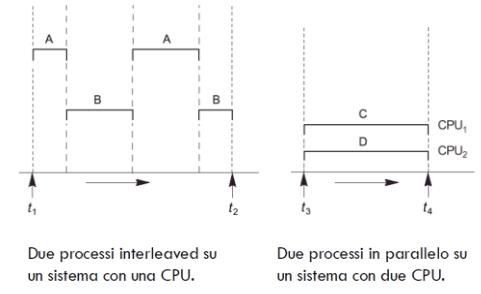
Quando una voce è cancellata, è sempre rimossa dal livello delle foglie. Se capita che si trovi in un nodo interno, deve essere eliminata anche da lì. In questo caso il valore alla sua sinistra nel nodo foglia deve sostituirla nel nodo interno, perché quel valore ora è la voce più a destra nel sottoalbero. La cancellazione può causare uno **svuotamento** riducendo il numero di voci nel nodo foglia sotto il minimo richiesto. In questo caso si cerca di trovare un nodo foglia **fratello**, cioè un nodo esterno esattamente a sinistra o a destra del nodo con lo svuotamento, e di **ridistribuire** le voci tra il nodo e suo fratello in modo che entrambi siano almeno completi a metà, altrimenti il nodo è fuso con i suoi fratelli e il numero di nodi foglia si riduce. Un metodo comune è provare a ridistribuire le voci nel fratello a sinistra, se non è possibile, viene fatto un tentativo di ridistribuzione con il fratello a destra. Se anche questo non è possibile, i tre nodi vengono fusi ottenendo due nodi esterni. In questo caso lo svuotamento può propagarsi ai nodi **interni** perché sono necessari un puntatore a un albero e un valore di ricerca in meno. Questo effetto può propagarsi e ridurre i livelli dell'albero.

## 6. GESTIONE DELLE TRANSAZIONI

La **transazione** fornisce un meccanismo per descrivere le unità logiche di elaborazione delle basi di dati. I sistemi di gestione delle transazioni sono sistemi con grandi basi di dati e centinaia di utenti che eseguono transazioni contemporaneamente, come sistemi di prenotazione, bancari, ecc... I database system sono classificati in base al numero di utenti che utilizzano il sistema in **modo concorrente**:

- Un DBMS è **single-user** (*monoutente*) se al più un utente per volta può usare il sistema;
- Un DBMS è **multi-user** se più utenti possono usare il sistema concorrentemente.

Più utenti possono accedere al database simultaneamente grazie al concetto di **multiprogrammazione**, che consente ad un computer di elaborare più programmi o transazioni simultaneamente. Quando si ha una sola CPU, necessariamente si elabora un processo alla volta, infatti, l'illusione di avere più programmi che vengono eseguiti contemporaneamente è fornita dai *sistemi operativi* multiprogrammati, che eseguono alcune istruzioni da un programma, per poi sospenderlo ed eseguire altre istruzioni da altri programmi e così via. Quando un programma viene riattivato, esso riparte dal punto in cui era stato sospeso. Su *sistemi monoprocesso*, l'esecuzione concorrente dei programmi è quindi intervallata (*interleaved*), l'interleaving ha luogo quando un programma effettua operazioni di I/O che automaticamente lasciano la CPU inattiva. Su *sistemi multiprocesso* l'esecuzione dei programmi avviene realmente in *parallelismo*.



### TRANSAZIONI:

Una **transazione** è un insieme di operazioni che accedono al database, viste logicamente come un'istruzione singola ed indivisibile.

Per mostrare la gestione delle transazioni, verrà utilizzata una visione semplificata di un database system, dove un database è visto come una collezione di data item, ognuno con un nome. La dimensione del data item è detta **granularità**, ovvero può essere un singolo campo di un record, così come un intero blocco di un disco. Con tale semplificazione, le **possibili operazioni** di accesso al database, che una transazione può effettuare sono:

- **Read\_item(X)**: legge l'item di nome X in una variabile di programma. Sapendo che l'unità di trasferimento di dati è il blocco:
  1. Trovare l'indirizzo del blocco che contiene X.
  2. Copiare tale blocco in un buffer in memoria centrale (*se tale blocco non è già in un buffer*).
  3. Copiare l'elemento X dal buffer alla variabile di programma X.
- **Write\_item(X)**: scrive il valore della variabile di programma X nell'elemento X del database. I passi per eseguire un comando Write\_item(X) sono:
  1. Trovare l'indirizzo del blocco che contiene X.
  2. Copiare tale blocco in un buffer in memoria centrale (*se tale blocco non è già in un buffer*).
  3. Copiare l'elemento X dalla variabile di programma di nome X nella sua locazione nel buffer.
  4. Memorizzare il blocco aggiornato dal buffer al disco.

### CONTROLLO DELLA CONCORRENZA:

Il **controllo della concorrenza** e dei **meccanismi di recovery** (recupero) riguardano principalmente i comandi di accesso al database in una transazione. Transazioni inviate da più utenti, che possono accedere e aggiornare gli elementi del database, vengono eseguite in modo concorrente. Se l'esecuzione concorrente non è controllata, si possono avere problemi di **database inconsistente** (due dati che rappresentano la stessa informazione hanno valori diversi, così facendo i dati non sono più affidabili). Di seguito viene mostrato un esempio di problemi con le transazioni:

Si supponga di avere un database per la prenotazione di posti in aereo, in cui è memorizzato un record per ogni volo. Si supponga di avere due transazioni, T<sub>1</sub> e T<sub>2</sub>, dove la transazione T<sub>1</sub> cancella N prenotazioni da un volo il cui numero di posti occupati è memorizzato nell'item del database di nome X, e riserva lo stesso numero su un altro volo il cui numero di posti occupati è memorizzato nell'item di nome Y. T<sub>2</sub> prenota M posti sul primo volo referenziato nella transazione T<sub>1</sub>.

Si osservino i possibili **problem**i che possono sorgere eseguendo T<sub>1</sub> e T<sub>2</sub> concorrentemente.

(a)	T <sub>1</sub>	(b)	T <sub>2</sub>
	read_item(X); X:=X-N; write_item(X); read_item(Y); Y:=Y+N; write_item(Y);	read_item(X); X:=X+M; write_item(X);	read_item(X); X:=X+M; write_item(X);

#### Problema dell'aggiornamento perso:

Supponendo che T<sub>1</sub> e T<sub>2</sub> siano avviate contemporaneamente e che le loro operazioni siano interleaved (interfogliate) dal sistema operativo nel modo seguente:

Tempo	T <sub>1</sub>	T <sub>2</sub>	
	read_item(X); X:=X-N;  write_item(X); read_item(Y);  Y:=Y+N; write_item(Y);	read_item(X); X:=X+M;  write_item(X);	Il valore finale di X è scorretto perché T <sub>2</sub> legge il valore di X prima che T <sub>1</sub> lo salvi: l'aggiornamento di T <sub>1</sub> è quindi perso.

#### Problema dell'aggiornamento temporaneo (o lettura sporca):

Una transazione aggiorna un elemento ma poi essa fallisce per un qualche motivo, non riuscendo a salvare tale aggiornamento. L'elemento aggiornato è però letto da un'altra transazione prima che esso sia riportato al suo valore originario. Il valore di X letto da T<sub>2</sub> è detto **dato sporco**, perché è stato creato da una transazione annullata.

Tempo	T <sub>1</sub>	T <sub>2</sub>	
	read_item(X); X:=X-N; write_item(X);	read_item(X); X:=X+M; write_item(X);	read_item(X); X:=X+M; write_item(X);

#### Problema della totalizzazione scorretta:

Se una transazione sta calcolando una funzione di aggregazione su un certo insieme di record, mentre altre transazioni stanno aggiornando alcuni di tali record, la funzione può calcolare alcuni valori prima dell'aggiornamento ed altri dopo.

Una transazione T<sub>3</sub> sta calcolando il numero totale di prenotazioni su tutti i voli mentre T<sub>1</sub> è in esecuzione:

Tempo	T <sub>1</sub>	T <sub>3</sub>	
	read_item(X); X:=X-N; write_item(X);  ⋮  read_item(Y); Y:=Y+N; write_item(Y);	sum:=0; read_item(A); sum:=sum+A;  ⋮  read_item(X); sum:=sum+X; read_item(Y); sum:=sum+Y;	Il risultato di T <sub>3</sub> è sbagliato, poiché T <sub>3</sub> legge il valore di X dopo che sono stati sottratti N posti, e prima che gli stessi siano sommati ad Y.

#### Problema delle letture non ripetibili:

Tale problema avviene se una transazione T<sub>1</sub> legge due volte lo stesso item, ma tra le due letture una transazione T<sub>2</sub> ne ha modificato il valore. Ad esempio: durante una prenotazione di posti aerei, un cliente chiede informazioni su più voli. Quando il cliente decide, la transazione deve rileggere il numero di posti disponibili sul volo scelto per completare la prenotazione, ma potrebbe non trovare più la stessa disponibilità.

## TECNICHE DI RECOVERY:

È necessario l'operazione di **recovery** perché quando viene inoltrata una transazione, il sistema deve far sì che tutte le operazioni siano completate con successo ed il loro effetto sia registrato permanentemente nel database oppure la transazione annullata non abbia effetti né sul database né su qualunque altra transazione. Vi possono essere diverse failure, ed esse in genere vengono suddivise in fallimenti di transazione, di sistemi e di media. Le possibili ragioni di una **failure** sono (i problemi 1-4 sono i più frequenti, ed è più facile effettuarne il recovery):

1. Un **crash di sistema** durante l'esecuzione della transazione.
2. **Errore di transazione** o di **sistema** (overflow, divisione per zero, valori errati di parametri, ecc...).
3. **Errori locali o condizioni eccezionali** rilevati dalla transazione (ad esempio, i dati per la transazione possono non essere trovati o essere non validi, tipo un ABORT programmato a fronte di una richiesta di un prelievo da un fondo scoperto).
4. **Controllo della concorrenza**: il metodo di controllo della concorrenza può decidere di abortire la transazione perché viola la serializzabilità o perché varie transazioni sono in deadlock.
5. **Fallimento di disco**: alcuni blocchi di disco possono perdere i dati per un malfunzionamento in lettura/scrittura, a causa di un crash della testina.
6. **Problemi fisici e catastrofi** (fuoco, sabotaggio, furto, caduta di tensione, errato montaggio di nastro da parte dell'operatore, ecc...).

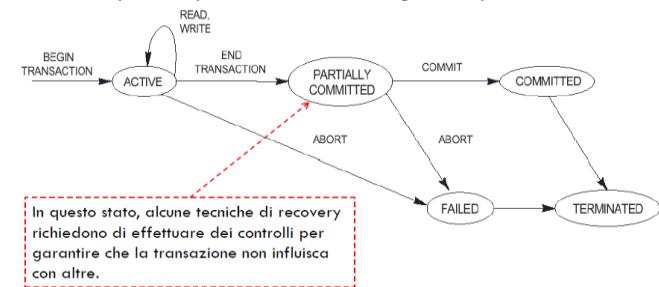
## CONCETTI SULLE TRANSAZIONI:

Una **transazione** è un'**'unità atomica di lavoro** che, o è completata nella sua interezza o è integralmente annullata. Per motivi di recovery, il sistema deve tenere traccia dell'inizio, della fine o dell'abort di ogni transazione. Il manager di recovery tiene quindi traccia delle seguenti operazioni:

- **BEGIN\_TRANSACTION**: marca l'inizio dell'esecuzione della transazione.
- **READ** o **WRITE**: specifica operazioni di lettura o scrittura sul database, eseguite come parte di una transazione.
- **END\_TRANSACTION**: specifica che le operazioni di READ e WRITE sono finite e marca il limite di fine esecuzione della transazione.
- **COMMIT\_TRANSACTION**: segnala la fine con successo della transazione, in modo che qualsiasi cambiamento può essere reso permanente, senza possibilità di annullarlo.
- **ROLL-BACK** (o **ABORT**): segnala che la transazione è terminata senza successo e tutti i cambiamenti o effetti nel database devono essere annullati.

Operazioni addizionali:

- **UNDO**: simile al roll-back, eccetto che si applica ad un'operazione singola piuttosto che a una intera transazione.
- **REDO**: specifica che certe operazioni devono essere ripetute.



## PROPRIETÀ DELLE TRANSAZIONI:

Le transazioni dovrebbero possedere alcune proprietà (dette **ACID properties**, dalle loro iniziali):

- **Atomicità**: l'atomicità rappresenta il fatto che una transazione è un'unità *indivisibile* di esecuzione, cioè vengono resi visibili tutti gli effetti di una transazione oppure la transazione non deve avere alcun effetto sulla base di dati, con un approccio "tutto o niente". In pratica non è possibile lasciare la base di dati in uno stato intermedio attraversato durante l'elaborazione della transazione (*responsabilità del recovery Subsystem*).
- **Consistency preserving**: l'esecuzione di una transazione non viola i vincoli di integrità definiti sulla base di dati. Quando il sistema rileva che una transazione sta violando uno dei vincoli, esso interviene per annullare la transazione o per correggere la violazione del vincolo. Quindi una transazione deve far passare il database da uno stato consistente ad un altro (*responsabilità dei programmati*).
- **Isolamento**: l'isolamento richiede che l'esecuzione di una transazione sia indipendente dalla contemporanea esecuzione di altre transazioni, rendendo invisibili i vari aggiornamenti finché non è committed. In particolare, si richiede che il risultato dell'esecuzione concorrente di un insieme di transazioni sia analogo al risultato che le stesse transazioni otterrebbero qualora ciascuna di esse fosse eseguita da sola (*responsabilità del sistema per il controllo della concorrenza*).
- **Durability**: la persistenza richiede che l'effetto di una transazione che ha eseguito il commit correttamente non venga più perso. In pratica, una base di dati deve garantire che nessun dato venga perso per nessun motivo (*responsabilità del sistema di gestione dell'affidabilità*).

## SYSTEM LOG:

Per effettuare il recovery di transazioni abortite, il sistema mantiene un **log** per tenere traccia delle operazioni che modificano il database. Il **system log** contiene informazioni su tutte le transazioni che modificano i valori degli item del database. Il **log** è strutturato come una lista di record, in ognuno di essi è memorizzato un **ID univoco** della transazione **T**, generato in automatico dal sistema. Alcuni tipi di entry possibili nel log sono:

- **[start\_transaction, T]**: la transazione T ha iniziato la sua esecuzione.
- **[write\_item, T, X, old\_value, new\_value]**: la transazione T ha cambiato il valore dell'item X da old\_value a new\_value.
- **[read\_item, T, X]**: la transazione T ha letto l'item X.
- **[commit, T]**: la transazione T è terminata con successo e le modifiche possono essere memorizzate in modo permanente.
- **[abort, T]**: la transazione T è fallita.

Il **file di log** deve essere tenuto su disco. Aggiornare il log implica copiare il blocco dal disco al buffer in memoria, aggiornare il buffer e riscrivere il buffer su disco. In caso di una failure, solo le entry su disco vengono usate nel processo di recovery, poiché un blocco viene tenuto in memoria finché non è pieno, prima che una transazione raggiunga il punto di commit, ogni parte del log in memoria deve essere scritta (**force writing**).

## GLI SCHEDULE:

Uno **schedule** è l'ordine in cui sono eseguite le operazioni di più transazioni processate in modo interleaved. Formalmente, uno schedule S di n transazioni  $T_1, T_2, \dots, T_n$  è un ordinamento delle operazioni delle transazioni, soggetto al vincolo che per ogni transazione  $T_i$  che partecipa in S, le operazioni in  $T_i$  in S devono apparire nello stesso ordine di apparizione in  $T_i$ . Si supponga che l'ordinamento delle operazioni in S sia totale.

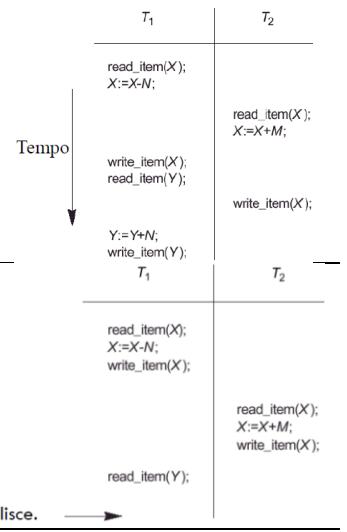
Due operazioni in uno schedule sono in **confitto** se:

1. Appartengono a differenti transazioni;
2. Accedono allo stesso elemento X;
3. Almeno una delle due operazioni è una write\_item(X). In verde le coppie che non generano conflitti e in rosso le coppie che generano conflitti.

$S_a : r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1$

Lo schedule per queste transazioni, che vengono chiamate  $S_a$ , può essere espresso come ( $r$  = read,  $w$  = write):

$S_a : r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y)$



Lo schedule  $S_b$  per queste transazioni può essere espresso come:

$S_b : r_1(X); w_1(X); r_2(X); w_2(X); r_1(Y); a_1$

### SCHEDULE COMPLETO:

Uno schedule  $S$  di  $n$  transazioni  $T_1, T_2, \dots, T_n$  è uno **schedule completo** se valgono le seguenti condizioni:

- Le operazioni in  $S$  sono esattamente quelle in  $T_1, T_2, \dots, T_n$ , inclusa operazione di *commit* o di *abort* come ultima operazione di ogni transazione in  $S$ .
- Per ogni coppia di operazioni della stessa transazione  $T_i$ , il loro ordine di occorrenza in  $S$  è lo stesso che è in  $T_i$ .
- Per ogni coppia di operazioni in conflitto, una deve occorrere prima dell'altra nello schedule.

Uno **schedule completo** non contiene transazioni attive, perché sono tutte committed o aborted. Dato uno schedule  $S$ , si definisce **proiezione committed C(S)**, uno schedule che contiene solo le operazioni in  $S$  che appartengono a transazioni committed. È importante caratterizzare i tipi di schedule in base alla possibilità di effettuare il recovery. Infatti, si vuol garantire che *per una transazione committed non è mai necessario il roll-back*.

Uno schedule con tale proprietà è detto **recoverable**, alternativamente, uno schedule  $S$  è detto recoverable se nessuna transazione  $T$  in  $S$  fa un commit finché tutte le transazioni  $T'$ , che hanno scritto un elemento letto da  $T$ , hanno fatto un commit. Si mostra un esempio di schedule recoverable:

Sia  $S_a' : r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1$

→ Risulta essere recoverable, ma soffre del problema dell'aggiornamento perso.

Sia  $S_c' : r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1$

→ Esso non è recoverable perché  $T_2$  legge  $X$  da  $T_1$  e poi fa il commit prima di  $T_1$ . Se  $T_1$  abortisce dopo  $c_2$ , allora il valore di  $X$  che  $T_2$  legge non è più valido e  $T_2$  deve essere abortito dopo aver fatto commit.

Se si modifica  $S_c$  in:  $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2$

→  $S_c$  diviene uno schedule recoverable.

### ROLL-BACK IN CASCATA:

Negli schedule recoverable nessuna transazione committed ha necessità di roll-back. Si possono però avere **roll-back in cascata** se una transazione non committed legge un dato scritto da una transazione fallita. Uno schedule è detto **cascadeless (evitare il roll-back in cascata)** se ogni transazione nello schedule legge elementi scritti solo da transazioni committed. Viene mostrato un esempio di Cascadeless Schedule:

Sia  $S_e : r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_1; c_2$

→ Esso non è cascadeless:  $T_2$  legge  $X$  scritto da  $T_1$  prima che  $T_1$  raggiunga il commit o l'abort.

$S_e' : r_1(X); w_1(X); r_1(Y); w_1(Y); c_1; r_2(X); w_2(X); c_2$

### SCHEDULE STRETTI:

Uno schedule è detto **stretto** se le transazioni non possono né leggere né scrivere un elemento  $X$  finché l'ultima transazione che ha scritto  $X$  non è completata (*commit* o *abort*). Tali schedule stretti semplificano il processo di recovery poiché occorre solo ripristinare la **before image (old\_value)**.

Sia  $S_f : w_1(X, 5); w_2(X, 8); a_1$  → Si supponga che inizialmente  $X = 9$ .  $S_f$  è cascadeless ma non stretto. Se  $T_1$  fallisce, la procedura di recovery ripristina il valore di  $X$  a 9, anche se è stato modificato da  $T_2$ .

### SERIALIZZABILITÀ DI SCHEDULE:

Oltre a caratterizzare gli schedule in base alla possibilità di recovery, si vorrebbe poterli classificare anche in base al loro comportamento in ambiente concorrente. Uno schedule è **seriale** se per ogni transazione  $T$  nello schedule, tutte le operazioni di  $T$  sono eseguite senza interleaving, altrimenti non risulta essere seriale.

Ad esempio: i due possibili casi di schedule seriali con due transazioni  $T_1$  e  $T_2$ :

- $T_1$  eseguito prima di  $T_2$
- $T_1$  eseguito dopo  $T_2$

Gli **schedule seriali** sollevano alcuni problemi, infatti essi limitano la concorrenza o le operazioni di interleaving, diventando **inaccettabili**:

- Se una transazione aspetta una operazione di I/O, non si può allocare la CPU ad un'altra transazione.
- Se una transazione  $T$  dura a lungo, le altre transazioni devono aspettare che finisca.

Anche nel caso degli **schedule non seriali** vi possono essere problemi di aggiornamento perso, aggiornamento temporaneo, somma scorretta, ecc...

### SCHEDULE SERIALIZZABILI:

Uno schedule  $S$  di  $n$  transazioni è **serializzabile** se è "equivalente" a qualche schedule seriale delle stesse  $n$  transazioni. Dati  $n$  schedule, abbiamo  $n!$  possibili seriali. Due schedule sono detti **result equivalent** (equivalenti) se producono lo stesso stato finale del database, ma non è una definizione accettabile, perché la *produzione dello stesso stato può essere accidentale*.

I due schedule sono result equivalent solo se  $X = 100$ :

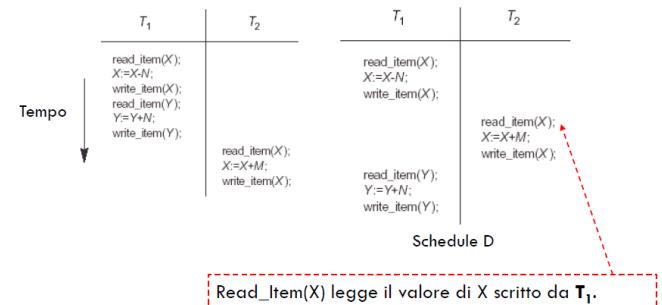
$a)$	$T_1$	$T_2$	$b)$	$T_1$	$T_2$
read_item( $X$ ); $X:=X-N;$	read_item( $X$ ); $X:=X-M;$	write_item( $X$ ); read_item( $Y$ ); $Y:=Y+N;$ write_item( $Y$ );	read_item( $X$ ); $X:=X-N;$	read_item( $X$ ); $X:=X+M;$	write_item( $X$ );
read_item( $Y$ ); $Y:=Y+N;$ write_item( $Y$ );	read_item( $Y$ ); $Y:=Y+N;$ write_item( $Y$ );		read_item( $X$ ); $X:=X+M;$ write_item( $X$ );	read_item( $Y$ ); $Y:=Y+N;$ write_item( $Y$ );	

Una definizione più appropriata è quella di **conflict equivalent**, cioè due schedule sono conflict equivalent se l'ordine di ogni coppia di operazioni in conflitto è lo stesso in entrambi gli schedule. Ad esempio, in  $S_1$  sono presenti  $r_1(X); w_2(X)$  e nello schedule  $S_2$  sono in ordine inverso,  $w_2(X); r_1(X)$ . Il valore  $r_1(X)$  può essere differente nei due schedule.

$S_1$	$S_2$
read_item( $X$ ); $X:=X+10;$ write_item( $X$ );	read_item( $X$ ); $X:=X*1.1;$ write_item( $X$ );

Uno schedule è **conflict serializable** se è **conflict equivalent** a qualche schedule seriale S'. È possibile determinare, per mezzo di un semplice algoritmo, la **conflict serializzabilità** di uno schedule. Molti metodi per il controllo della concorrenza non testano la serializzabilità, piuttosto utilizzano protocolli che garantiscono che uno schedule sarà serializzabile. Si mostra un esempio di conflict serializzabilità:

Lo schedule D è conflict serializzabile.



Read\_item(X) legge il valore di X scritto da  $T_1$ .

### ALGORITMO PER LA SERIALIZZABILITÀ:

L'algoritmo cerca solo le operazioni di `read_item` e `write_item`, per costruire un **grafo di precedenza** (o **grafo di serializzazione**).

Un **grafo di precedenza** è un grafo diretto, dove  $G=(N, E)$ , con un insieme di nodi:  $N = \{T_1, T_2, \dots, T_n\}$  ed un insieme di archi  $E=\{e_1, e_2, \dots, e_m\}$ .

Ogni arco è della forma  $(T_j \rightarrow T_k)$ , con  $1 \leq j, k \leq n$ , ed è creato se un'operazione in  $T_j$  appare nello schedule prima di qualche operazione in conflitto in  $T_k$ .

#### Algoritmo 1:

- Per ogni transazione  $T_i$  nello schedule S, creare un nodo  $T_i$  nel grafo;
- Per ogni caso in S dove  $T_j$  esegue una `read_item(X)` dopo che  $T_i$  esegue una `write_item(X)`, creare un arco  $(T_i \rightarrow T_j)$  nel grafo;
- Per ogni caso in S dove  $T_j$  esegue una `write_item(X)` dopo che  $T_i$  esegue una `read_item(X)`, creare un arco  $(T_i \rightarrow T_j)$  nel grafo;
- Per ogni caso in S dove  $T_j$  esegue una `write_item(X)` dopo che  $T_i$  esegue una `write_item(X)`, creare un arco  $(T_i \rightarrow T_j)$  nel grafo;
- Lo schedule è serializzabile se e solo se il grafo non contiene cicli.

#### Esempio1:

Uno schedule seriale con due transazioni, ed il relativo grafo di precedenza:

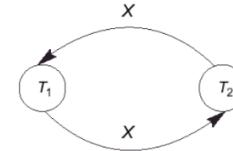
$T_1$	$T_2$
<code>read_item(X); X:=X-N; write_item(X); read_item(Y); Y:=Y+N; write_item(Y);</code>	<code>read_item(X); X:=X+M; write_item(X);</code>



#### Esempio2:

Uno schedule non seriale con due transazioni, ed il relativo grafo di precedenza che ne evidenzia la non serializzabilità.

$T_1$	$T_2$
<code>read_item(X); X:=X-N;</code>	<code>read_item(X); X:=X+M;</code>
<code>write_item(X); read_item(Y);</code>	<code>write_item(X);</code>
<code>Y:=Y+N; write_item(Y);</code>	

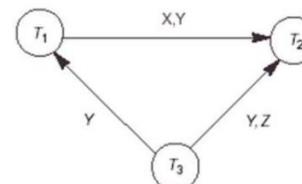


### COSTRUZIONE DI UNO SCHEMA SERIALE:

Se non ci sono cicli nel **grafo di precedenza** relativo ad uno schedule S si può **creare uno schedule seriale equivalente**  $S'$  ordinando le transazioni che partecipano allo schedule come segue:

- Se esiste un arco fra  $T_i$  e  $T_j$ ;  $T_i$  deve apparire prima di  $T_j$  nello schedule seriale equivalente.

transaction $T_1$	transaction $T_2$	transaction $T_3$
<code>read_item(X); write_item(X);</code>		<code>read_item(Y); read_item(Z);</code>
	<code>read_item(Z);</code>	<code>write_item(Y); write_item(Z);</code>



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

### SUPPORTO ALLE TRANSAZIONI IN SQL:

Il concetto di transazione in SQL è simile a quanto visto finora: *una transazione è una singola unità logica di lavoro con la proprietà dell'atomicità*. Di default, in SQL ogni singola istruzione è una transazione. Non esiste uno Statement di `Begin_Transaction`, poiché l'inizio di una transazione viene determinato implicitamente. Deve esserne però esplicitata la fine, con le istruzioni `COMMIT` o `ROLLBACK`.

Ogni transazione in SQL ha tre caratteristiche, specificate per mezzo dell'istruzione **SET TRANSACTION** che inizia una transazione:

- Modalità di accesso**: specifica se l'accesso ai dati è in sola lettura o in lettura/scrittura.
- Dimensione dell'area diagnostica**: specifica lo spazio da usare per informazioni all'utente sull'esecuzione delle transazioni.
- Isolation Level**: specifica la politica di gestione delle transazioni concorrenti.

È molto importante utilizzare due commit (uno prima e uno dopo). SET TRANSACTION deve essere la prima istruzione SQL di una transazione.

- Il `COMMIT` appena prima assicura che ciò sia vero.
- Il `COMMIT` alla fine rilascia le risorse possedute dalla transazione.

La **modalità di accesso** può essere specificata come READ ONLY o READ WRITE (di default).

- La modalità READ WRITE permette l'esecuzione di comandi di aggiornamento, inserimento, cancellazione e creazione.
- La modalità READ ONLY serve unicamente per il recupero di dati.

Alcune transazioni effettuano istruzioni di SELECT su diverse tabelle e dovranno vedere dati coerenti, dati che si riferiscono allo stesso istante di tempo. **SET TRANSACTION READ ONLY** specifica questo meccanismo più protetto di gestione dei dati. Nessun comando può modificare i dati di un'area su cui vengono effettuate operazioni di SELECT attraverso questo tipo di transazioni.

L'opzione **dimensione dell'area di diagnosi DIAGNOSTIC SIZE n** specifica un valore intero *n*, che indica il numero di condizioni che possono essere mantenute contemporaneamente nell'area di diagnosi. Tali condizioni forniscono informazioni di feedback (*errori* o *eccezioni*) riguardo i comandi SQL eseguiti più di recente.

L'opzione **livello di isolamento** è specificata usando l'istruzione **ISOLATION LEVEL <isolamento>**.

I possibili valori sono:

- READ UNCOMMITTED
- READ COMMITTED
- REPEATABLE READ
- SERIALIZABLE (è il livello predefinito)

Oltre alle già viste violazioni di "lettura sporche" e "lettura non ripetibili", usando SQL può sorgere il problema delle "lettura fantasma". Ad esempio, si supponga che una transazione **T1** legga una serie di righe basate su una condizione di WHERE. Se una transazione **T2** aggiunge dei valori che soddisfano la condizione di WHERE, una riesecuzione di **T1** vedrà delle righe nuove (*fantasma*), non presenti in precedenza.

Livello di isolamento	Tipo di violazione		
	Lettura sporca	Lettura non ripetibile	Fantasma
READ UNCOMMITTED	SI	SI	SI
READ COMMITTED	NO	SI	SI
REPEATABLE READ	NO	NO	SI
SERIALIZABLE	NO	NO	NO

#### ESEMPIO TRANSAZIONE SQL:

La transazione produce prima l'inserimento di una nuova riga nella tabella EMPLOYEE e successivamente esegue l'aggiornamento dello stipendio di tutti gli impiegati che lavorano nel reparto 2.

In caso di errore in una qualsiasi istruzione SQL, l'intera transazione viene annullata (rollback). Ogni valore di stipendio aggiornato viene ripristinato al suo valore precedente e la nuova riga viene rimossa.

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;  
EXEC SQL SET TRANSACTION  
    READ WRITE  
    DIAGNOSTICS SIZE 5  
    ISOLATION LEVEL SERIALIZABLE;  
  
EXEC SQL INSERT INTO EMPLOYEE (FNAME, LNAME  
                                SSN, DNO, SALARY)  
VALUES ('Robert','Smith','991004321',2,35000);  
  
EXEC SQL UPDATE EMPLOYEE  
    SET SALARY = SALARY * 1.1 WHERE DNO = 2;  
  
EXEC SQL COMMIT;  
GOTO THE_END;  
UNDO: EXEC SQL ROLLBACK;  
THE_END: ...
```

## 7. TECNICHE PER IL CONTROLLO DELLA CONCORRENZA

L'esecuzione di transazioni concorrenti senza alcun controllo può comportare svariati problemi al database, e perciò è necessario evitare che esse interferiscano fra di loro garantendo l'**isolamento**. Per favorire ciò si utilizzano delle tecniche di gestione delle transazioni, per garantire che il database sia sempre in uno stato consistente, tali tecniche garantiscono la serializzabilità degli schedule, usando particolari **protocolli**.

- **Tecniche di locking:** i data item sono bloccati per prevenire che transazioni multiple accedano allo stesso item concorrentemente.
- **Timestamp:** un identificatore unico per ogni transazione, generato dal sistema. Un protocollo può usare l'ordinamento dei timestamp per assicurare la serializzabilità.

Un fattore importante è la **granularità**, infatti la granularità di un data item è la porzione del database rappresentata dal data item stesso. Esso può essere della dimensione che varia da un attributo ad un singolo blocco di un disco o anche un intero file, o un intero database.

### TECNICHE DI LOCKING PER IL CONTROLLO DELLA CONCORRENZA:

Le **tecniche di locking** si basano sul concetto di "blocco" (*lock*) di un item. Un **lock** è una variabile associata ad un data item nel database, e descrive lo stato di quell'elemento rispetto alle possibili operazioni applicabili ad esso. I lock sono un mezzo per sincronizzare l'accesso da parte di transazioni concorrenti agli elementi del database. Possono essere utilizzati diversi tipi di lock per il controllo della concorrenza, in particolare verranno esaminati:

- **Lock binari:** risultano essere semplici ma molto restrittivi e **non vengono** molto usati nella pratica.
- **Lock shared/esclusivi:** vengono usati molto nei DBMS commerciali e forniscono maggiori capacità di controllo e concorrenza.

#### LOCK BINARI:

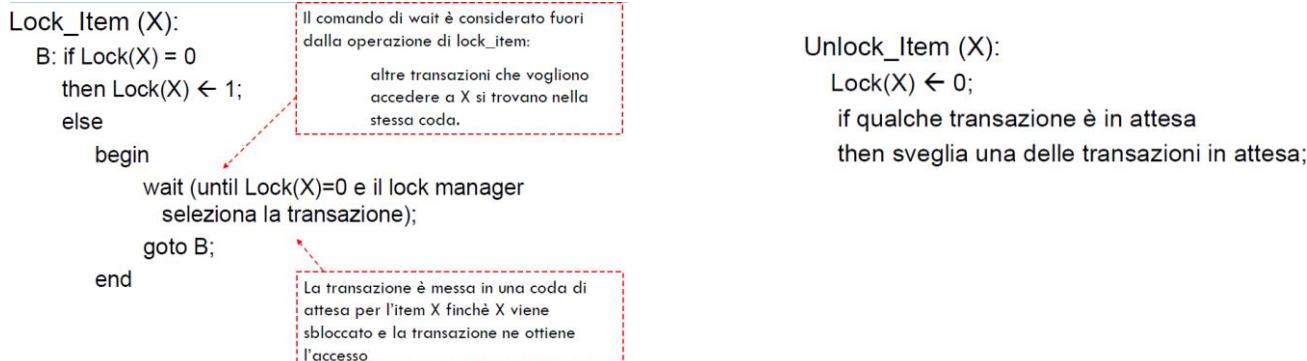
Un **lock binario** può assumere due valori (o stati): **locked** (o 1) e **unlocked** (o 0). A ciascun elemento X del database viene associato un distinto lock:

- Se  $lock(X) = 1$ , le operazioni del database non possono accedere all'elemento X.
- Se  $lock(X) = 0$ , si può accedere all'elemento X quando richiesto.

Le transazioni che usano lock binari devono contenere operazioni di **lock\_item** e **unlock\_item**.

Una transazione chiede di accedere a un elemento X con l'istruzione **lock\_item(X)**: se  $lock(X) = 1$ , la transazione è forzata ad attendere altrimenti se così non fosse viene posto il  $lock(X)$  a 1, ottenendo l'accesso all'elemento. Al termine dell'utilizzo di X, la transazione invia un'istruzione di **unlock\_item(X)**, che pone  $lock(X)$  a 0, permettendo l'accesso all'item ad altre transazioni.

Un lock binario rafforza la **mutua esclusione** di un data item. Le operazioni di **lock** e **unlock\_item** devono essere implementate come **unità indivisibili** (**sezioni critiche**), nel senso che non è consentito alcun interleaving dall'avvio fino al termine dell'operazione di lock/unlock o all'intervento della transazione di una coda di attesa. Il DBMS dispone di un sottosistema di **lock manager** per seguire e controllare gli accessi ai lock.



Per implementare un **lock binario** è necessario solo una variabile binaria **LOCK** associata ad ogni data item X del database. Ogni lock può essere visto come un record con tre campi: **<nome data item, LOCK, transazione >**

Con associata una coda delle transazioni che stanno provando ad accedere all'elemento. Gli elementi che non sono nella **lock table** sono considerati non bloccati (**unlocked**). L'organizzazione della tabella è basata su **hash file**. Usando uno schema di lock binario, ogni transazione deve obbedire alle seguenti **regole**:

1. Una transazione T deve impartire l'operazione di **lock\_item(X)** prima di eseguire una **Read\_item(X)** o **Write\_item(X)**.
2. Una transazione T deve impartire l'operazione di **unlock\_item(X)** dopo aver completato tutte le operazioni di **Read\_item(X)** e **Write\_item(X)**.
3. Una transazione T non impartirà un **lock\_item(X)** se già vale il lock sull'elemento X.
4. Una transazione T non impartirà un **unlock\_item(X)** a meno che non valga già un lock sull'elemento X.

Al più una transazione può mantenere il lock su un elemento X, cioè due transazioni non possono accedere allo stesso elemento concorrentemente.

#### LOCK SHARED / ESCLUSIVI:

Il **lock binario** è troppo restrittivo, poiché l'accesso ad un data item è consentito ad una sola transazione per volta, è possibile consentire l'accesso in sola lettura a più transazioni contemporaneamente. Se una transazione deve scrivere un data item X, deve avere un **accesso esclusivo** su X. Per questo motivo si utilizza un **multiple mode lock**, cioè un lock che può avere più stati.

In questo caso le operazioni di lock diventano tre: **Read\_lock(X)**, **Write\_lock(X)**, **Unlock(X)**.

Ed un lock ha tre possibili stati: **Read\_locked** (share locked), **Write\_locked** (exclusive locked), **Unlocked**.

Ciascuna delle tre operazioni, **Read\_lock(X)**, **Write\_lock(X)**, **Unlock\_item(X)**, deve essere considerata indivisibile: nessun interleaving deve essere consentito dall'inizio dell'operazione fino al completamento o all'inserimento della transazione in una coda di attesa per quell'elemento.

Una possibile implementazione dei **lock shared/esclusivi** è che ogni lock è rappresentato da un record con quattro campi:

**<Nome data item, Lock, Numero di read, Transazione/i bloccante/i >**

Dove Lock assume un valore che permette di distinguere tra **Read\_locked**, **Write\_locked** e **Unlocked**. Per risparmiare spazio, il sistema mantiene nella **lock table** i record per gli elementi locked.

```

Read_Lock(X):
B: if LOCK(X) = "unlocked"
then begin
    LOCK(X) ← "read_locked";
    numero_di_read (X) ← 1;
end;
else
if LOCK(X) = "read_locked"
then numero_di_read (X) = numero_di_read (X) + 1;
else
begin
    wait (until LOCK(X) = "unlocked" and il gestore di lock sceglie la transazione);
    goto B;
end;

```

```

Write_Lock(X):
B: if LOCK(X) = "unlocked"
then LOCK(X) ← "write_locked";
else
begin
    wait (until LOCK(X) = "unlocked" e il gestore di lock sceglie la transazione);
    goto B;
end;

```

```

Unlock(X):
if LOCK(X) = "write_locked"
then
begin
    LOCK(X) ← "unlocked";
    sveglia una delle transazioni in attesa se ne esistono;
end;
else if LOCK(X) = "read_locked"
then
begin
    numero_di_read (X) = numero_di_read (X) - 1;
    if numero_di_read (X) = 0
    then
        begin
            LOCK(X) = "unlocked";
            sveglia una delle transazioni in attesa se ne esistono;
        end
end;

```

Usando uno schema di **shared/exclusive**, ogni transazione deve obbedire alle seguenti **regole**:

- Una transazione T deve impartire l'operazione di Read\_Lock(X) o Write\_Lock(X) prima di eseguire una Read\_item(X).
- Una transazione T deve impartire l'operazione di Write\_Lock(X) prima di eseguire una Write\_item(X).
- Una transazione T deve impartire l'operazione di Unlock(X) dopo aver completato tutte le operazioni di Read\_item(X) o Write\_item(X).
- Una transazione T non impartirà un Read\_Lock(X) se già è in possesso di un lock condiviso in lettura o lock esclusivo in scrittura sull'elemento X.
- Una transazione T non impartirà un Write\_Lock(X) se già vale il lock in lettura o scrittura sull'elemento X.
- Una transazione T non impartirà un Unlock(X) a meno che non valga già un lock sull'elemento X.

I vincoli 4 e 5 possono essere tralasciati per permettere **conversioni di lock**.

#### CONVERSIONI DI LOCK:

Una transazione può invocare un Read\_Lock(X) e poi successivamente **incrementare** il lock, invocando un Write\_Lock(X). Tale conversione è possibile solo se T è l'unica transazione che ha un Read\_Lock su X, altrimenti deve aspettare. È possibile anche **decrementare** un lock, se una transazione T invoca una Write\_Lock(X) e successivamente una Read\_Lock(X).

Per permettere tali conversioni, è necessario che sia mantenuto un identificatore della transazione nella struttura del record per ciascun lock. È ovviamente necessario modificare le operazioni di Read\_Lock, Write\_Lock e Unlock per supportare l'informazione aggiuntiva.

#### LOCK E SERIALIZZABILITÀ:

Lock binari e multiple-mode non garantiscono la serializzabilità degli schedule.

Esempio di schedule seriali:

T <sub>1</sub>	T <sub>2</sub>	Dati i valori iniziali X=20 e Y=30:
read_lock(Y); read_item(Y); unlock(Y);  write_lock(X); read_item(X); X:=X+Y; write_item(X); unlock(X);	read_lock(X); read_item(X); unlock(X);  write_lock(Y); read_item(Y); Y:=X+Y; write_item(Y); unlock(Y);	Se T <sub>1</sub> è seguito da T <sub>2</sub> : X=50, Y=80
		Se T <sub>2</sub> è seguito da T <sub>1</sub> : X=70, Y=50

Esempio di schedule non serializzabile:

T <sub>1</sub>	T <sub>2</sub>	Dati i valori iniziali X=20 e Y=30:  Risultato dello schedule S: <b>X=50, Y=50</b>
read_lock(Y); read_item(Y); unlock(Y);	read_lock(X); read_item(X); unlock(X);  write_lock(Y); read_item(Y); Y:=X+Y; write_item(Y); unlock(Y);	Y è stato sbloccato troppo presto!

Occorre un **protocollo** per stabilire il posizionamento delle operazioni di lock/unlock in ogni transazione.

#### PROTOCOLLO TWO-PHASE LOCKING:

Una transazione T segue il protocollo **Two-Phase Locking (2PL)** se tutte le operazioni di locking (Read\_lock o Write\_Lock) precedono la prima operazione di Unlock nella transazione. Una transazione del genere può essere divisa in due fasi,

- Expanding phase** (*espansione*), possono essere *acquisiti nuovi lock* su elementi ma nessuno può essere rilasciato.
- Shrinking phase** (*contrazione*), i lock esistenti possono essere *rilasciati* ma non possono essere acquisti nuovi lock.

Se la conversione di lock è permessa, l'upgrading deve essere fatto durante la fase di espansione ed il downgrading durante la contrazione.

T <sub>1</sub>	T <sub>1</sub> '	T <sub>2</sub>	T <sub>2</sub> '
read_lock(Y); read_item(Y); unlock(Y);  write_lock(X); read_item(X); X:=X+Y; write_item(X); unlock(X);	read_lock (Y); read_item (Y); write_lock (X); unlock (Y);	read_lock(X); read_item(X); unlock(X);  write_lock(Y); read_item(Y); Y:=X+Y; write_item(Y); unlock(Y);	read_lock (X); read_item (X); write_lock (Y); unlock (X);

#### 2PL E SERIALIZZABILITÀ:

È dimostrabile che se **ogni transazione** in uno schedule segue il protocollo 2PL, allora lo schedule è serializzabile. 2PL può però **limitare la concorrenza** in uno schedule: la garanzia della serializzabilità viene pagata al costo di non consentire alcune situazioni di concorrenza possibili, poiché alcuni elementi possono essere bloccati più del necessario, finché la transazione necessita di effettuare letture e scritture.

## 2PL CONSERVATIVO:

Il protocollo 2PL appena visto è detto **2PL di base**. Una variazione del 2PL è nota come **2PL conservativo** (o **statico**). Esso richiede che una transazione, prima di iniziare, blocchi tutti gli elementi a cui accede, pre-dichiarando i propri **read\_set** e **write\_set**:

- Il **read\_set** è l'insieme di tutti i data item che saranno *letti* dalla transazione.
- Il **write\_set** è l'insieme di tutti i data item che saranno *scritti* dalla transazione.

Se qualche data item dei due insiemi non può essere bloccato, la transazione resta in attesa finché tutti gli elementi necessari non divengono disponibili. Il 2PL conservativo è un protocollo **deadlock-free**. Non usato nella pratica perché è necessario pre-dichiarare il read-set ed il write-set.

## 2PL STRETTO:

La variazione più diffusa del protocollo 2PL è il **2PL stretto**, che garantisce **schedule stretti**, in cui le transazioni non possono né scrivere né leggere un elemento X finché l'ultima transazione che ha scritto X non termina (con commit o abort). Nel 2PL stretto, quindi, una transazione non rilascia nessun lock esclusivo finché non termina, ed esso non risulta essere deadlock-free.

## GENERAZIONE AUTOMATICA DI RICHIESTE DI READ E WRITE LOCK:

In molti casi il **sottosistema per il controllo della concorrenza** genera automaticamente le richieste di lock:

- Quando la transazione effettua una `Read_item(X)`, il sistema genera una operazione `Read_Lock(X)`.
- Quando la transazione effettua una `Write_item(X)`, il sistema genera una operazione `Write_Lock(X)`.

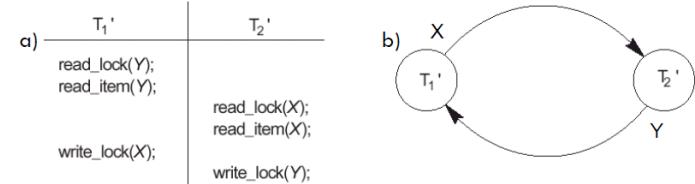
Se T invoca un `Read_item(X)`, il sistema invoca un `Read_Lock(X)` per T. Se lo stato di `LOCK(X)` = "write\_locked" da una T', il sistema pone T nella coda di attesa per X, altrimenti esegue il `Read_Lock(X)` e quindi l'operazione di `Read_item(X)` per T.

Se T invoca un `Write_item(X)`, il sistema invoca un `Write_Lock(X)` per T. Se lo stato di `LOCK(X)` = "write\_locked" OR "read\_locked" da una T', il sistema pone T nella coda di attesa per X. Se lo stato di `LOCK(X)` = "read\_locked" dall'unica transazione T, il sistema promuove il lock a "write\_locked" ed esegue l'operazione di `Write_item(X)` per T. Se lo stato di `LOCK(X)` = "unlocked", il sistema esegue la `Write_Lock(X)` e quindi l'operazione di `Write_item(X)` per T. Il protocollo di lock a due fasi garantisce la serializzabilità, ma non consente tutti i possibili schedule serializzabili, cioè alcuni schedule serializzabili vengono vietati da protocollo. Essi possono causare **deadlock** e **starvation**.

Si ha un **deadlock** quando due o più transazioni aspettano qualche item bloccato da altre transazioni T' in un insieme. Ogni transazione T' è in una coda di attesa e aspetta che un elemento sia rilasciato da un'altra transazione in attesa. Per prevenire il deadlock, occorre usare un protocollo apposito (**deadlock prevention protocol**). Il protocollo a prevenzione di deadlock usato nel 2PL conservativo richiede che ogni transazione blocchi tutti i data item di cui ha bisogno in anticipo, se qualcosa non può essere ottenuta, nessun elemento è bloccato per cui la transazione aspetta e riprova in seguito. Lo **svantaggio** è la limitazione della concorrenza.

Sono stati proposti molti altri schemi per la prevenzione di deadlock:

- **Basati su timestamp;**
- **Senza timestamp:** Algoritmo non-waiting e Algoritmo cautious waiting;
- **Basati su time-out**, dove T richiede un lock aspettando fino ad un certo time-out. Se il lock non viene assegnato in tempo, T subisce un abort, esegue un rollback e ricomincia (indipendentemente dalla presenza di deadlock).



Esempio di deadlock:

- a) Uno schedule di  $T_1'$  e  $T_2'$  in deadlock.
- b) Il grafo wait-for (delle attese) corrispondente.

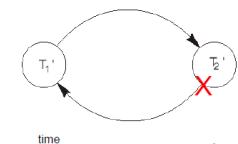
## PREVENZIONE DI DEADLOCK BASATA SU TIMESTAMP:

Il **Timestamp TS(T)** di una transazione T è un identificatore unico associato ad ogni transazione, e si basa sull'ordine di partenza delle transazioni, ovvero se  $T_1$  inizia prima di  $T_2$ , allora  $TS(T_1) < TS(T_2)$ .

$T_i$  prova a bloccare X che è bloccato da  $T_j$ .

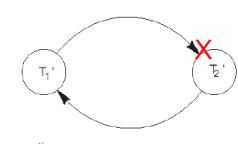
Schema **wait-die**: se  $TS(T_i) < TS(T_j)$ , ( $T_i$  più vecchia) allora  $T_i$  aspetta, altrimenti ( $T_i$  più giovane)  $T_i$  viene abortita (**abort**) e ripartirà *con lo stesso timestamp*.

wait-die



Schema **wound-wait**: se  $TS(T_i) < TS(T_j)$ , ( $T_i$  più vecchia) allora  $T_j$  fallisce (**preempt**) e viene riavviata successivamente *con lo stesso timestamp*, altrimenti ( $T_i$  più giovane)  $T_i$  aspetta.

wound-wait



Entrambi gli schemi fanno fallire la transazione **più giovane**. Sono deadlock-free, ma possono causare l'abort di transazioni senza necessità (*non provocherebbero realmente un deadlock*).

- $T'_1$  prova a bloccare X che è bloccato da  $T'_2$ .
- $T'_2$  prova a bloccare Y che è bloccato da  $T'_1$ .

## SCHEMI SENZA TIMESTAMP:

**No-waiting (NG):** Se una transazione T non è in grado di ottenere un lock viene immediatamente interrotta e successivamente fatta ripartire dopo un intervallo di tempo, senza effettuare controlli sulla possibilità che si possa verificare un deadlock. Nessuna T aspetta, quindi non esiste la possibilità di deadlock. Questo metodo, però, non è pratico e può causare la terminazione e il riavvio di transazioni inutilmente.

**Cautious Waiting (CW):** Se  $T_i$  cerca un lock sull'item X ma non è in grado in quanto X è bloccato da  $T_k$ : Se  $T_k$  non è in attesa di qualche altro item bloccato, allora  $T_i$  aspetta, altrimenti  $T_i$  viene terminata.

Un approccio più pratico risulta essere quello del **Deadlock Detection**, dove il sistema controlla l'esistenza di un deadlock. Per riconoscere uno stato di deadlock, si utilizza il **grafo wait\_for** (delle attese):

- Si crea un nodo per transazione in esecuzione.
- Si aggiunge un arco tra il nodo  $T_i$  e il nodo  $T_j$ , se  $T_i$  aspetta di bloccare un elemento usato da  $T_j$ .
- Si cancella un arco tra  $T_i$  e  $T_j$  appena l'elemento richiesto da  $T_i$  viene allocato a  $T_j$ .

Se c'è un **ciclo nel grafo**, si è in uno stato di deadlock, dopodiché, si sceglie quale transazione abortire, usando un **criterio di selezione della vittima**.

L'algoritmo che seleziona la vittima in genere evita di scegliere transazioni che sono in esecuzione da molto tempo o hanno effettuato molti aggiornamenti. La **deadlock detection** è una soluzione valida quando non ci sono molte interferenze tra le transazioni. Le transazioni sono brevi ed ognuna di essa blocca pochi elementi, in alternativa per transazioni lunghe si usa il **deadlock prevention**.

## STARVATION:

La **starvation** è un altro problema che può sorgere con l'utilizzo dei lock. Una transazione è nello stato di starvation se non può procedere per un tempo indefinito mentre altre transazioni nel sistema continuano normalmente. La causa è nello schema di waiting non sicuro che dà la precedenza ad alcune transazioni invece di altre.

Un possibile schema di waiting sicuro (*safe*) usa una coda **first-come-first-serve (FCFS)** dove le transazioni bloccano gli elementi rispettando l'ordine con cui hanno richiesto il lock. Un altro schema è basato su **priorità**, che aumenta proporzionalmente al tempo atteso dalla transazione. Starvation si può avere anche negli algoritmi per il trattamento del deadlock, se l'algoritmo seleziona ripetutamente la stessa transazione come vittima. Per porre una soluzione a ciò, l'algoritmo può usare priorità più alte per transazioni che sono state abortite più volte.

Gli schemi **wait-die** e **wound-wait** escludono la starvation.

## GRANULARITÀ DEGLI ITEM:

Tutte le tecniche per il controllo della concorrenza assumono che il database sia formato da un insieme di data item. Un data item può essere:

- Un record del database.
- Un campo di un record del database.
- Un blocco del disco.
- Un intero file.
- L'intero database.

La dimensione di un data item è detta **granularità di un data item**. Tale granularità può essere:

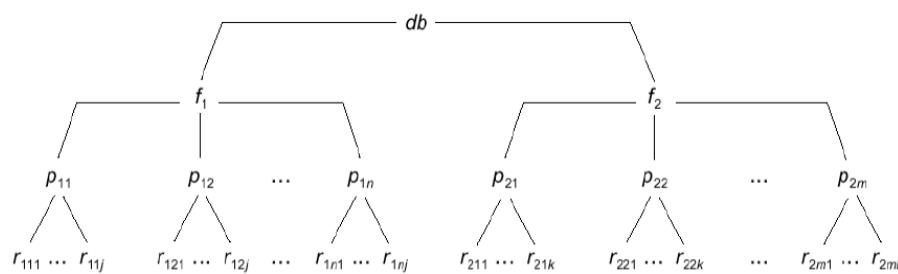
- **Fine**: riferita a data item di piccole dimensioni come un campo di un record.
- **Grossa**: riferita a data item di dimensioni maggiori come file o database intero.

La granularità **influenza le prestazioni** nel controllo della concorrenza e del recovery. Maggiore è il livello di granularità, minore è la concorrenza permessa. Se la dimensione di un data item è un blocco del disco, una transazione che necessita di leggere un record X in un blocco B effettuerà un lock dell'intero blocco. Altre transazioni, interessate a record diversi da X ma ugualmente in B, resteranno quindi inutilmente in attesa.

Per contro, a un data item di dimensioni inferiori corrisponde un numero maggiore di item nel database: ci sarà quindi una quantità superiore di lock ed il lock manager introdurrà un overhead nel sistema a causa delle molte operazioni che dovrà gestire. Sarà richiesto molto spazio per gestire la tabella dei lock. La taglia migliore dipende dal tipo di transazioni coinvolte. Se una transazione tipica accede a:

- Un piccolo numero di record, è vantaggioso avere una granularità di un record.
- Molti record nello stesso file, è vantaggioso avere una granularità a livello blocco o a livello file.

La soluzione è nella possibilità di definire **granularità multiple**: un lock può essere chiesto su item a qualsiasi livello di granularità.



Una gerarchia di granularità

## 8. TECNICHE DI RECOVERY

Se viene sottomessa una transazione T, possono esserci solo due possibilità, o tutte le operazioni di T sono completate ed il loro effetto è registrato permanentemente nel database, oppure T non ha nessun effetto né sul database né sulle altre transazioni (in caso di *failure*).

Nel secondo caso il sistema non può permettere che alcune operazioni di T siano portate a termine ed altre no. Effettuare un recovery (o recupero) da una transazione fallita significa **ripristinare** il database al più recente stato *consistente* appena prima del failure. Per fare ciò, il sistema deve tener traccia dei cambiamenti causati dall'esecuzione di transazioni sui dati. Tali informazioni sono memorizzate nel **system log**. Il log è strutturato come una lista di record, dove in ogni record è memorizzato un **ID univoco** della transazione **T**, generato in automatico dal sistema. Tipi di entry possibili nel log:

- [start\_transaction, T]
- [write\_item, T, X, old\_value, new\_value]
- [read\_item, T, X]
- [commit, T]
- [abort, T]

Esistono varie **strategie di recovery**, le tipiche:

1. Se il danno è notevole, a causa di un **failure catastrofico** (ad esempio il crash di un disco), si ripristina una precedente copia di back-up da una memoria di archivio e si ricostruisce lo stato più vicino a quello corrente riapplicando (REDO) tutte le transazioni completate (*committed*) salvate nel log fino al momento del failure.
2. Se, a seguito di un **failure non catastrofico**, il database non è danneggiato fisicamente, ma è solo in uno stato inconsistente, si effettua l'UNDO delle operazioni che hanno causato l'inconsistenza.

Eventualmente si effettua il REDO di alcune operazioni, non è necessario effettuare il restore del database, poiché basta consultare il system log.

### TECNICHE DI RECOVERY DA FAILURE NON CATASTROFICI:

Concettualmente si possono distinguere due tecniche principali per il recovery da failure non catastrofici:

- **Tecniche ad aggiornamento differito** (*deferred update*) – Algoritmo NO-UNDO/REDO:

Con questa tecnica, i dati non sono fisicamente aggiornati fino all'esecuzione del commit di una transazione. Le modifiche effettuate da una transazione sono memorizzate in un suo spazio di lavoro locale (workspace o buffer). Durante il commit, gli aggiornamenti sono salvati persistentemente prima nel log e poi nel database. Se la transazione fallisce, non è necessario l'UNDO, può però essere necessario il REDO di alcune operazioni (commit con scrittura nel log ma non nel database).

- **Tecniche ad aggiornamento immediato** (*immediate update*) – Algoritmo UNDO/REDO:

Il database può essere aggiornato fisicamente prima che la transazione effettui il commit. Le modifiche sono registrate prima nel log (con un force-writing) e poi sul database, permettendo comunque il recovery. Se una transazione fallisce dopo aver effettuato dei cambiamenti, ma prima del commit, l'effetto delle sue operazioni nel database deve essere annullato, occorre quindi effettuare il rollback della transazione (UNDO), può però essere necessario il REDO di alcune operazioni (commit con scrittura nel log ma non nel database).

### CACHING DEI BLOCCHI:

Le tecniche di recovery possono essere influenzate da particolari gestioni del file system da parte del sistema operativo, in particolare dal **buffering** e dal **caching** di blocchi del disco in memoria centrale. Per migliorare l'efficienza degli accessi a disco, i blocchi del disco contenenti i dati manipolati spesso dal DBMS, sono conservati (*cached*) in un buffer della memoria centrale. I dati sono quindi aggiornati in memoria, prima di essere riscritti su disco. Sebbene la gestione del caching sia un compito del sistema operativo, spesso è il DBMS ad occuparsene esplicitamente, a causa dello stretto accoppiamento con le tecniche di recovery. Il DBMS gestisce direttamente una serie di buffer, che formano la **cache del DBMS**.

Per tenere traccia dei data item presenti in cache, si usa una **directory**, ovvero una tabella con entry del tipo:

**<indirizzo pagina su disco, locazione nel buffer>**

### ACCESSO AI DATI CON CACHE:

Quando il DBMS richiede l'accesso ad un data item, controlla prima la presenza in cache, se è presente il DBMS ne ottiene l'accesso, altrimenti:

1. Deve essere individuato il blocco su disco contenente l'item.
2. Il blocco deve essere copiato nella cache.

3. Se la cache è piena, sono necessarie strategie tipiche dei sistemi operativi per il rimpiazzamento delle pagine (LRU - least recently used, FIFO, ecc.).

Ad ogni blocco nella cache si può associare un **dirty bit**, per evidenziare se qualche elemento del buffer è stato modificato o meno. Al caricamento di un blocco in un buffer, il suo dirty bit è posto a 0, quando avviene una modifica del contenuto del buffer, il suo dirty bit è posto a 1. Un buffer deve essere salvato su disco solo se il suo dirty bit vale 1.

### STRATEGIE PER LO SVUOTAMENTO DELLA CACHE:

Esistono due strategie principali per lo svuotamento di buffer modificati:

1. **In-place updating**: il buffer è riscritto nella stessa posizione originaria sul disco. Si mantiene in cache una singola copia di ogni blocco del disco. È necessario usare il system log per il recovery.
2. **Shadowing**: un buffer può essere riscritto in una locazione differente, permettendo la presenza su disco di più versioni di un data item. Sia il vecchio valore (BFIM – *before image*), sia quello aggiornato (AFIM, *after image*) di un data item possono essere presenti sul disco contemporaneamente.

Utilizzando l'In-place updating è necessario il system log per il recovery. Il log contiene due tipi di informazioni: quelle per l'UNDO e quelle per il REDO.

- Un'entry del log di tipo UNDO include il vecchio valore (BFIM) dell'item salvato (necessario per effettuare un UNDO).
- Un'entry del log di tipo REDO include il nuovo valore (AFIM) dell'item salvato (necessario per effettuare un REDO).

### PROTOCOLLO WAL (WRITE-AHEAD LOGGING):

Per permettere il recovery con l'In-place updating, le entry appropriate devono essere salvate nel log su disco prima di applicare i cambiamenti nel database. In questo protocollo:

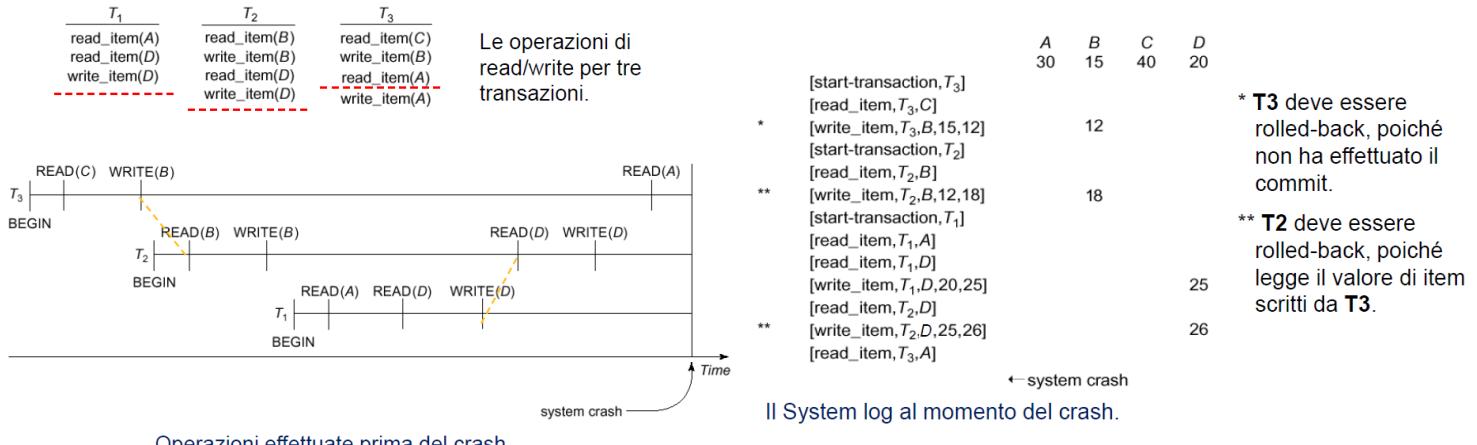
1. L'AFIM non può **sovrascrivere** la BFIM di un elemento sul disco finché non sono stati memorizzati i record di log di tipo UNDO della transazione.
2. L'operazione **commit** non può essere completata finché non sono scritti su disco tutti i record di log di tipo UNDO e di tipo REDO.

## CHECKPOINT NEL LOG:

Nel log vengono utilizzate particolari entry, dette **checkpoint**. Nel log viene registrato un **[checkpoint]** periodicamente, quando il DBMS salva su disco tutti i blocchi modificati in cache. In questo modo, tutte le transazioni che hanno effettuato la *commit prima del checkpoint* non richiedono operazioni di REDO in caso di crash, poiché le loro modifiche sono già state rese permanenti. Il recovery manager crea dei checkpoint ad intervalli regolari (ad esempio ogni **m** minuti o ogni **t** transazioni terminate). Per creare un checkpoint si effettuano le seguenti operazioni:

1. Sospendere temporaneamente l'esecuzione di tutte le transazioni.
2. Scrivere su disco il contenuto di tutti i buffer modificati (scrittura forzata).
3. Scrivere una entry di checkpoint nel log.
4. Riprendere l'esecuzione delle transazioni sospese.

Se una transazione fallisce per una qualsiasi ragione, può essere necessario effettuarne il **rollback**. Il rollback di una transazione **T** richiede il rollback di tutte le transazioni che hanno letto il valore di qualche dato scritto da **T**, e così via (**rollback in cascata**). È complesso da gestire e richiede molto tempo, infatti i meccanismi di recovery sono progettati per evitare di usarlo. Di seguito un esempio:



## TECNICHE DI RECOVERY BASATE SU DEFERRED UPDATE:

Si definisce un **protocollo di deferred update**:

1. Una transazione non può cambiare il database finché non raggiunge il punto di commit.
2. Una transazione raggiunge il punto di commit quando tutte le sue operazioni di aggiornamento sono registrate nel log e tale log è scritto su disco.

L'algoritmo è NO-UNDO/REDO, infatti non è mai necessario l'UNDO, dato che gli aggiornamenti vengono fatti prima nel buffer, ed il REDO è richiesto solo se il crash si ha dopo il commit, ma prima dell'aggiornamento del database.

## RECOVERY CON DEFERRED UPDATE IN AMBIENTE MONOUTENTE (SINGLE-USER):

In questo caso l'algoritmo di recovery è abbastanza semplice: l'**algoritmo RDU\_S** (*Recovery usando la Deferred Update in ambiente Single-user*) chiama una procedura REDO per rieseguire delle operazioni di write\_item. La procedura RDU\_S mantiene due liste di transazioni:

1. Transazioni committed a partire dall'ultimo checkpoint;
2. Transazioni attive (contiene al più una transazione, perché il sistema è single-user).

## ALGORITMO RDU\_S:

Applicare l'operazione REDO a tutte le operazioni write\_item delle transazioni committed nel log, nell'ordine in cui sono scritte nel log.

**REDO (WRITE-OP):** per effettuare il REDO dell'operazione WRITE-OP bisogna esaminare la sua entry nel log **[write\_item, T, X, new\_value]** e porre il valore dell'elemento X a new-value (l'AFIM).

L'algoritmo rieffettuerà l'operazione **[write\_item, T1, D, 20]**, poiché **T1** era committed.

**NOTA:** La REDO è **idempotente**, perché se il sistema fallisce durante il recovery, deve essere possibile rifare il recovery, ed il risultato, con o senza crash, deve essere lo stesso.

T <sub>1</sub>	T <sub>2</sub>	System log
read_item(A)		[start-transaction, T <sub>1</sub> ]
read_item(D)		[write_item, T <sub>1</sub> , D, 20]
write_item(D)		[commit, T <sub>1</sub> ]
	read_item(B)	[start-transaction, T <sub>2</sub> ]
	read_item(D)	[write_item, T <sub>2</sub> , B, 10]
	write_item(D)	[write_item, T <sub>2</sub> , D, 25] ← system crash

Le operazioni di read/write per due transazioni.

Il System log.

## RECOVERY CON DEFERRED UPDATE IN AMBIENTE MULTITENTE:

In ambienti multiutente, i processi di recovery e di controllo della concorrenza sono interrelati. Maggiore è il grado di concorrenza, più tempo viene impiegato per effettuare il recovery. Si consideri un sistema multiutente così gestito: Controllo della concorrenza **2PL stretto** (two-phase locking), ovvero un lock mantenuto fino al punto di commit.

## ALGORITMO RDU\_M:

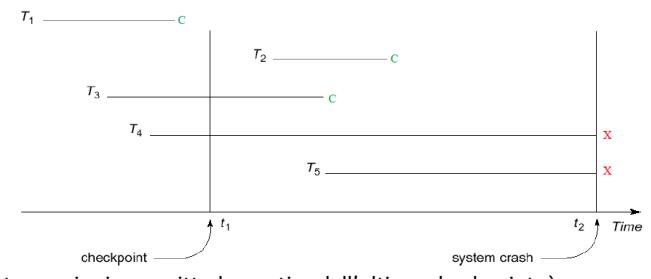
Esso utilizza due liste di transazioni:

1. Transazioni committed T a partire dall'ultimo checkpoint.
2. Transazioni attive T'.

Fare il REDO di tutte le operazioni di write\_item delle transazioni T committed nel log, nell'ordine in cui sono state scritte nel log. Le transazioni T' attive e non ancora committed devono essere rilanciate.

Non c'è bisogno di fare il REDO delle write per **T1**, poiché è stata committed (nel database) prima del crash. Si deve effettuare il REDO delle write di **T2** e **T3**. **T4** e **T5** devono essere rilanciate.

L'algoritmo può essere migliorato. Se un elemento X è stato aggiornato più volte da transazioni committed a partire dall'ultimo aggiornamento, è sufficiente effettuare il REDO solo dell'ultimo aggiornamento.



#### **SVANTAGGI DEL DEFERRED UPDATE:**

- Limitazione dell'esecuzione concorrente delle transazioni dovuto al 2PL stretto.
- Eccessivo spazio richiesto per memorizzare gli elementi aggiornati prima del commit di una transazione.

#### **VANTAGGI DEL DEFERRED UPDATE:**

- Se una transazione è abortita, viene risottomessa senza che sia stato alterato il database su disco. Non è necessario il rollback.
- Una transazione non leggerà mai un dato che sia stato modificato da una transazione non committed (2PL stretto). È escluso il **cascading rollback**.

#### **TECNICHE DI RECOVERY BASATE SU IMMEDIATE UPDATE:**

Quando una transazione effettua un comando di aggiornamento:

1. L'operazione viene registrata nel log (write-ahead logging protocol).
2. L'operazione viene applicata nel database.

In questo caso si **necessita di rollback**. I tipi di Immediate Update si dividono in due categorie:

1. Se tutti gli aggiornamenti di una transazione sono riportati nel database prima del commit, non è mai richiesto il REDO (algoritmo di recovery di tipo UNDO/NO-REDO).
2. Se la transazione raggiunge il commit prima che tutti gli aggiornamenti siano riportati nel database può essere necessario il REDO (algoritmo di recovery di tipo UNDO/REDO).

#### **ALGORITMO RIU\_S:**

L'algoritmo RIU\_S (*Recovery usando l'Immediate Update in ambiente Single-User*) utilizza due liste di transazioni:

1. Transazioni committed T a partire dall'ultimo checkpoint.
2. Transazione attiva T'.

L'algoritmo RIU\_S permette di:

- Eseguire l'UNDO delle operazioni write\_item della transazione attiva T' contenuta nel log.
- Eseguire il REDO delle operazioni write\_item delle transazioni committed T nell'ordine scritto nel log.

#### **ALGORITMO RIU\_M:**

L'algoritmo RIU\_M utilizza due liste di transazioni:

1. Transazioni committed T a partire dall'ultimo checkpoint.
2. Transazioni attive T'.

L'algoritmo RIU\_M permette di:

- Eseguire l'UNDO delle operazioni write\_item delle transazioni attive T' contenute nel log nell'ordine inverso rispetto a quello del log.
- Eseguire il REDO delle operazioni write\_item delle transazioni committed T nell'ordine scritto nel log.

#### **RECOVERY NEI SISTEMI MULTIDATABASE:**

Una Transazione Multidatabase è una *transazione che richiede l'accesso a database multipli*. Questi database possono essere gestiti da DBMS differenti (relazionali, O.O., gerarchici, ecc...). Vi è un meccanismo di recovery a due livelli:

1. Local recovery manager.
2. Global recovery manager (*coordinatore*).

#### **Protocollo di commit a due fasi: fase 1**

1. Tutti i database coinvolti dalla transazione segnalano al coordinatore di aver completato la loro parte.
2. Il coordinatore manda ad ogni partecipante il messaggio "prepare for commit".
3. Ogni partecipante forza su disco tutte le informazioni per il recovery locale e risponde "OK" al coordinatore. Se per qualche ragione non può esser fatto il commit, vi sarà una risposta "NOT OK".

#### **Protocollo di commit a due fasi: fase 2**

Se il coordinatore riceve "OK" da tutti i partecipanti, invia un comando di commit ai database coinvolti. Ogni partecipante segnala il [**commit**] nel system log, dove necessario, ed aggiorna il database.

Se qualche partecipante ha fornito un "NOT OK", la transazione fallisce e il coordinatore invia un messaggio UNDO ai partecipanti.

#### **BACKUP E RECOVERY DI DATABASE DA FAILURE CARASTROFICI:**

La principale tecnica è quella del back-up di database. Periodicamente il database e il system log sono ricopiatati su un device di memoria economico. Il system log è back-upped più di frequente rispetto al database intero. In caso di failure catastrofico, tutto il database viene ricaricato su dischi e, seguendo il system log, gli effetti delle transazioni committed vengono ripristinati.

Per non perdere tutte le transazioni effettuate dall'ultimo back-up, i file di log sono ricopiatati molto frequentemente. È possibile fare ciò grazie alle ridotte dimensioni di tali file rispetto alla taglia dell'intero database.

## 9. BASE DI DATI DISTRIBUITI ED ARCHITETTURA CLIENT-SERVER

La tecnologia per i **DDB (Database Distribuiti)** è il risultato dell'unione di tecnologia delle **basi di dati** e tecnologia delle **reti** e della **comunicazione**.

Utilizzare un DDB consente alle organizzazioni di effettuare un processo di **decentralizzazione** che viene raggiunta a livello di sistema, e di **integrare** le informazioni a livello logico. Un **database distribuito** è un singolo database logico che è sparso fisicamente attraverso computer in località differenti e connessi attraverso una rete di comunicazione dati. Invece, un **sistema per la gestione di basi di dati distribuite (DDBMS)** è un sistema software che gestisce un database distribuito rendendo la distribuzione **trasparente** all'utente. La rete deve quindi consentire agli utenti di **condividere dati**, ovvero un utente della località A deve essere in grado di accedere (e aggiornare) i dati della località B.

Vi sono diverse tipologie di architetture multiprocessore:

- Architettura a **memoria sparsa (shared nothing)**: ogni processore possiede la sua memoria primaria e secondaria e comunicano attraverso la rete. Non esiste una memoria comune e i nodi sono simmetrici ed omogenei;
- Architettura a **memoria condivisa (shared memory)**: diversi processori condividono sia la memoria primaria che la secondaria;
- Architettura a **disco condiviso (shared disk)**: i processori condividono la memoria secondaria, ma ognuno possiede la propria memoria primaria.

Nelle ultime due, i processori possono comunicare senza utilizzare la rete. DBMS basati su queste architetture si chiamano **Parallel DBMS**.

### ARCHITETTURA DI UN DDB:

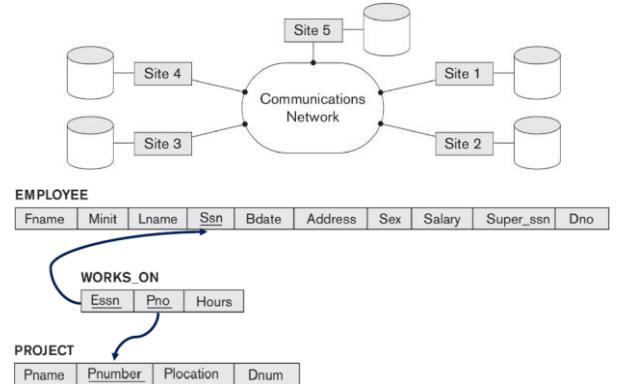
Un database distribuito è caratterizzato dall'**eterogeneità** dell'hardware e dei sistemi operativi di ogni nodo.

Un database distribuito richiede **DDBMS multipli**, funzionanti ognuno su di un sito remoto (nodo). Gli ambienti di database distribuiti si differenziano in base a:

- Il grado di cooperazione dei DDBMS.
- La presenza di un **sito master** che coordina le richieste che coinvolgono dati memorizzati in siti multipli.

I **vantaggi** dei Database Distribuiti, innanzitutto partono dalla gestione dei dati distribuiti a diversi **livelli di trasparenza**, ovvero sono nascosti i dettagli riguardo l'ubicazione di ogni data item presente nel sistema.

Le tabelle potrebbero essere frammentate orizzontalmente o memorizzate con replicazione.



### LIVELLI DI TRASPARENZA:

**Trasparenza di distribuzione o di rete**, dove vi è libertà dell'utente dai dettagli della rete. Essa si suddivide in:

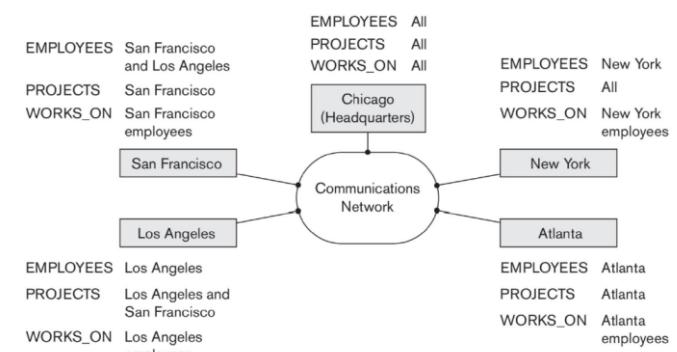
- **Trasparenza di locazione**: un comando utilizzato è indipendente dalla località dei dati e dalla località di emissione del comando.
- **Trasparenza di naming**: la specifica di un nome di un oggetto implica che una volta che un nome è stato fornito, gli oggetti possono essere referenziati usando quel nome, senza dover fornire dettagli addizionali.

**Trasparenza di replicazione**: dove delle copie dei dati possono essere mantenute in siti multipli per una migliore disponibilità, prestazioni ed affidabilità. L'utente non percepisce l'esistenza di tali copie.

**Trasparenza di frammentazione**, sono possibili due tipi di frammentazione:

- Frammentazione **orizzontale** che distribuisce una relazione attraverso insieme di tuple.
- Frammentazione **verticale** che distribuisce una relazione in sotto relazioni formate da un sottoinsieme delle colonne della relazione originale.

Una **query globale** dell'utente deve essere trasformata in una serie di **frammenti di query**. La trasparenza di frammentazione consente all'utente di non percepire l'esistenza di tali frammenti.



### VANTAGGI DEI DDB:

Vi sono vari vantaggi alla base dell'utilizzo dei DDB. Infatti, essi presentano:

- **Affidabilità**: la probabilità che il sistema sia funzionante ad un certo istante.
- **Disponibilità**: la probabilità che il sistema è continuamente disponibile durante un intervallo di tempo.

Quando un sito fallisce, gli altri possono continuare ad operare, solo i dati del sito fallito sono inaccessibili. La replicazione dei dati aumenta ancora di più l'affidabilità e la disponibilità. In un sistema distribuito è più facile espandere il sistema in termini di aggiunta di nuovi dati, accrescere il numero di siti o aggiungere nuovi processori (*miglioramento delle prestazioni*). La **localizzazione dei dati** dove i dati sono mantenuti nei siti più vicini a dove sono più utilizzati, ciò comporta una riduzione di accesso a reti geografiche. Ogni sito ha un database di taglia più piccola e, query e transazioni sono processate più rapidamente. Le transazioni su ogni sito sono in numero minore rispetto ad un database centralizzato.

### FUNZIONALITÀ ADDIZIONALI E CARATTERISTICHE DEI DDBMS:

Rispetto ai tradizionali DBMS, i DDBMS devono fornire le seguenti funzionalità:

- **Mantenere traccia dei dati**, della loro distribuzione, frammentazione e replicazione nel catalogo.
- **Processare query distribuite**, l'abilità di accedere a siti remoti e trasmettere query e dati tra i vari siti attraverso la rete di comunicazione.
- **Gestire le transazioni distribuite**, query e transazioni devono accedere ai dati da più di un sito mantenendo l'integrità del DB.
- **Gestire la replicazione dei dati**, quindi decidere quando replicare un dato e *mantenere la consistenza fra le copie*.
- **Gestire il recovery di un DDB**, ovvero il *fallimento di uno dei siti e dei link di comunicazione*.
- **Sicurezza**, le transazioni distribuite devono essere gestite garantendo la *sicurezza dei dati e l'accesso degli utenti*.
- **Gestire il catalogo distribuito**, il catalogo deve contenere i dati dell'intero DB ed essere globale. Decidere *come e dove distribuire il catalogo*.

A **livello hardware**, si devono considerare le seguenti differenze rispetto ai DBMS:

- Esistono diversi computer, detti siti o nodi.
- I siti devono essere connessi attraverso una rete di comunicazione per trasmettere dati e comandi ad altri siti.

I siti possono essere connessi attraverso una rete locale e/o distribuiti geograficamente e connessi ad una rete geografica.

## DESIGN DEI DDB:

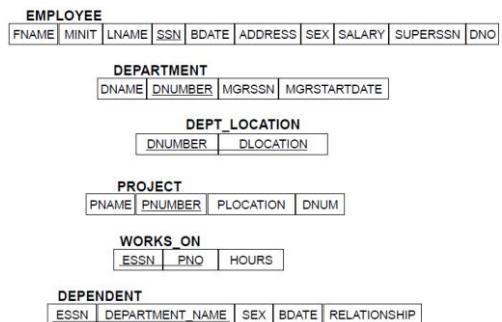
Il design dei DDB prevede l'applicazione di tecniche relative a: **Frammentazione** dei dati, **Replicazione** dei dati e **Allocazione** dei frammenti. Le informazioni relative saranno memorizzate nel catalogo.

### FRAMMENTAZIONE DEI DATI:

Deve essere deciso quale sito deve memorizzare le diverse porzioni del database. Si assume che non ci siano replicazione dei dati. Una relazione può:

1. Essere memorizzata per intero in un sito.
2. Essere divisa in unità più piccole distribuite.

Lo schema del database "Company":



FR1  
FR2  
FR3

### FRAMMENTAZIONE ORIZZONTALE:

Un **frammento orizzontale** di una relazione è un sottoinsieme delle tuple della relazione. Le tuple vengono assegnate ad un determinato frammento orizzontale in base al valore di uno o più attributi. Ad esempio, considerato il database "Company", esso viene distribuito su 3 siti e si definiscono 3 frammenti orizzontali in base al valore del numero del dipartimento (DNO=5) (DNO=4) (DNO=1). Ogni frammento contiene le tuple degli impiegati che lavorano per un particolare dipartimento.

La **segmentazione orizzontale** divide una relazione raggruppando righe per creare sottoinsiemi di tuple, ciascuno con un significato logico. La segmentazione orizzontale **derivata**, applica la partizione di una relazione primaria anche a relazioni secondarie, collegate alla prima con una chiave esterna. Ad esempio, dal partizionamento di DIPARTIMENTO deriva il partizionamento di IMPIEGATO e PROGETTO.

Ogni **frammento orizzontale** su di una relazione **R** può essere specificato attraverso un'operazione  $\sigma_{C_i}(R)$ .

Un insieme di frammenti orizzontali tale che le condizioni  $C_1, \dots, C_n$  includono tutte le tuple della relazione, è detto **frammentazione orizzontale completa**. In molti casi i frammenti sono disgiunti: nessuna tupla in **R** soddisfa  $C_i \text{ AND } C_j$  per  $i \neq j$ .

FR1 FR2 FR3

### FRAMMENTAZIONE VERTICALE:

La **frammentazione verticale** divide una relazione "verticalmente" per colonne. Un frammento verticale di una relazione mantiene solo determinati attributi di una relazione. Ad esempio, IMPIEGATO può essere suddiviso in due frammenti:

- Uno contenente le informazioni personali {NOME, DATA\_DI\_NASCITA, INDIRIZZO E SESSO}.
- L'altro contenente {SSN, SALARIO, SUPERSSN, NUMERO\_DIPARTIMENTO}.

La frammentazione precedente non consente di ricostruire la tupla IMPIEGATO originale, infatti non ci sono attributi in comune fra i due frammenti. La soluzione è aggiungere SSN agli attributi personali. È necessario includere la chiave primaria o una candidata in ogni frammento verticale.

Un **frammento verticale** può essere specificato da una operazione dell'algebra relazionale  $\pi_{L_i}(R)$ . Un insieme di frammenti verticali la cui lista di proiezioni  $L_1, \dots, L_n$  include tutti gli attributi di **R** e condivide solo la **chiave primaria** è detta **frammentazione verticale completa**. Vengono presentate le caratteristiche della frammentazione verticale completa. Devono essere soddisfatte le seguenti condizioni:

- $L_1 \cup L_2 \cup \dots \cup L_n = \text{ATTRIBUTI}(R)$
- $L_i \cap L_j = \text{PK}(R)$  per ogni  $i \neq j$ .

Per ricostruire la relazione **R** dai frammenti verticali è necessario fare l'OUTER JOIN dei frammenti. Ad esempio, i due frammenti  $L_1=\{\text{NOME}, \text{INDIRIZZO}\}$  e  $L_2=\{\text{SSN}, \text{NOME}, \text{SALARIO}\}$ , non costituiscono un frammento verticale completo.

FR1 FR2 FR3

### FRAMMENTAZIONE MISTA:

In questa modalità è possibile combinare la frammentazione orizzontale e verticale. Ad esempio, si combinano le due frammentazioni viste per la relazione IMPIEGATO. La relazione originaria può essere ricostruita applicando OUTER JOIN e UNION nell'ordine appropriato.

### SCHEMA DI FRAMMENTAZIONE:

Uno **schema di frammentazione** di un database è la definizione di un insieme di frammenti che include tutte le tuple e gli attributi del database. Esso, inoltre, consente la ricostruzione dell'intero database applicando una sequenza di operazioni di OUTER JOIN e UNION.

### SCHEMA DI ALLOCAZIONE:

Lo **schema di allocazione** descrive l'allocazione dei frammenti ai siti del database. Associa ad ogni frammento il sito in cui deve essere memorizzato.

### REPLICAZIONE ED ALLOCAZIONE DI DATI:

La replicazione consente di migliorare la disponibilità dei dati. Le modalità di replicazione sono le seguenti:

- Replicare l'intero database in ogni sito, migliora le prestazioni per le query. L'overhead risulta essere eccessivo per l'aggiornamento dei dati e il controllo della concorrenza e recovery sono complessi.
- Nessuna replicazione, frammenti disgiunti (eccetto per la chiave).
- Replicazione parziale, una soluzione intermedia, dove alcuni frammenti possono essere replicati ed altri no.

### SCHEMA DI REPLICAZIONE:

Lo **schema di replicazione** è una descrizione della replicazione dei frammenti. Ogni segmento deve essere assegnato ad un sito del database, e questo processo porta il nome di **allocazione dei dati**. La scelta dei siti e del grado di replicazione dipende da: Prestazioni, Obiettivi di disponibilità del sistema e Tipo e frequenza delle transazioni sottomesse ad ogni sito.

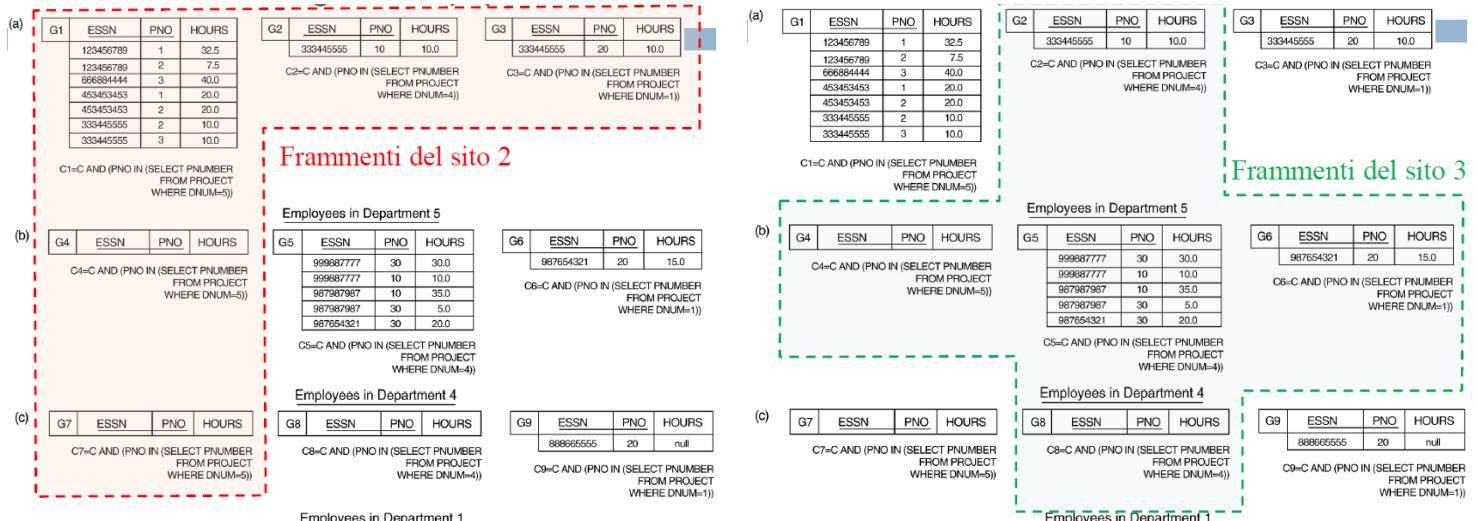
### CRITERI PER DISTRIBUIRE I DATI:

Trovare una soluzione ottima alla distribuzione dei dati è un problema difficile. Se è richiesta un'alta disponibilità e le transazioni possono essere sottomesse ad ogni sito e molte transazioni sono solo di **retrieval**, si può adottare una soluzione di replicazione totale. Se alcune transazioni che accedono a porzioni del database sono sottomesse solo da particolari siti, su quei siti possono essere allocati i frammenti corrispondenti. Se sono richiesti molti **aggiornamenti** è conveniente ridurre la replicazione dei dati.



## ALLOCAZIONE DEI SEGMENTI:

L'unione dei frammenti G1, G2, G3, G4, e G7 sono allocati nel **sito 2**. L'unione dei frammenti G2, G4, G5, G6, e G8 sono allocati nel **sito 3**. I frammenti G2 e G4 sono replicati in entrambi i siti. Questa strategia di allocazione permette di eseguire il JOIN tra i frammenti locali IMPiegato o PROGETTO dei siti 2 e 3 e il frammento locale WORKS\_ON completamente in locale. Ad esempio, per il sito 2, l'unione dei frammenti G1, G4 e G7 restituisce tutte le tuple di WORKS\_ON relativi ai progetti controllati dal dipartimento 5. Per il sito 3 vale lo stesso per i segmenti G2, G5 e G8.



## Esempio conti correnti bancari:

**CONTO-CORRENTE** (NUM-CC, NOME, FILIALE, SALDO)

**TRANSAZIONE** (NUM-CC, DATA, PROGR, AMMONTARE, CASUALE)

Frammentazione orizzontale principale:  $R_i = \sigma_{P_i}(R)$

- CONTO1 =  $\sigma_{\text{Filiale}=1}$  (**CONTO-CORRENTE**)
- CONTO2 =  $\sigma_{\text{Filiale}=2}$  (**CONTO-CORRENTE**)
- CONTO3 =  $\sigma_{\text{Filiale}=3}$  (**CONTO-CORRENTE**)

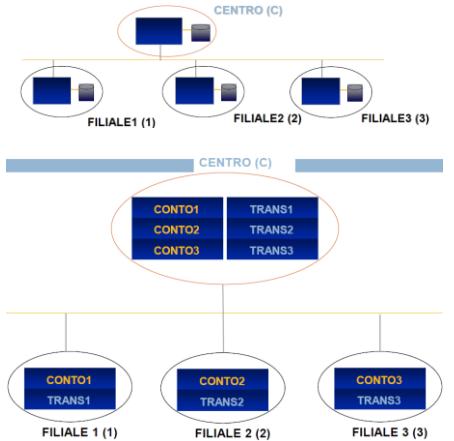
Frammentazione orizzontale derivata:  $S_i = S \triangleright \triangleleft R_i$

- TRANS1 = **TRANSAZIONE**  $\triangleright \triangleleft$  CONTO1
- TRANS2 = **TRANSAZIONE**  $\triangleright \triangleleft$  CONTO2
- TRANS3 = **TRANSAZIONE**  $\triangleright \triangleleft$  CONTO3

Allocazione dei frammenti:

**Rete**: 3 siti periferici e 1 sito centrale.

**Allocazione**: locale e centrale.



## LIVELLI DI TRASPARENZA:

Modalità per esprimere interrogazioni offerte dai DBMS commerciali (LIVELLI):

- **FRAMMENTAZIONE**: il programmatore non si deve preoccupare se o no il database è distribuito o frammentato.

### Esempio:

QUERY:

Estrarre il saldo del conto corrente 45

```
SELECT SALDO
  FROM CONTO-CORRENTE
 WHERE NUM-CC=45
```

QUERY:

Estrarre i movimenti dei conti con saldo negativo

```
SELECT NUM-CC, PROGR, AMMONTARE
  FROM CONTO-CORRENTE AS C
  JOIN TRANSAZIONE AS T
    ON C.NUM-CC=T.NUM-CC
 WHERE SALDO < 0
```

UPDATE:

Sposta il cliente 45 dalla Filiale 1 alla Filiale 2

```
UPDATE CONTO-CORRENTE
  SET FILIALE = 2
 WHERE NUM-CC=45 AND FILIALE=1
```

- **ALLOCAZIONE**: il programmatore dovrebbe conoscere la struttura dei frammenti, ma non deve indicare la loro allocazione.

### Esempio:

IPOTESI:

Conto corrente 45 presso la Filiale 1 (locale)

```
SELECT SALDO
  FROM CONTO1
 WHERE NUM-CC=45
```

IPOTESI:

Allocazione incerta: probabilmente allocato presso la Filiale 1

```
SELECT SALDO FROM CONTO1
 WHERE NUM-CC=45
 IF (NOT FOUND) THEN
   ( SELECT SALDO FROM CONTO2
     WHERE NUM-CC=45
 UNION
   SELECT SALDO FROM CONTO3
     WHERE NUM-CC=45 )
```

Join distribuito

```
SELECT NUM-CC, PROGR, AMMONTARE
  FROM CONTO1 JOIN TRANS1 ON .....
 WHERE SALDO < 0
 UNION
 SELECT NUM-CC, PROGR, AMMONTARE
  FROM CONTO2 JOIN TRANS2 ON .....
 WHERE SALDO < 0
 UNION
 SELECT NUM-CC, PROGR, AMMONTARE
  FROM CONTO3 JOIN TRANS3 ON .....
 WHERE SALDO < 0
```

Replicazione (*Bisogna settare anche la FILIALE a 2*)

```
INSERT INTO CONTO2
  SELECT * FROM CONTO1 WHERE NUM-CC=45
 INSERT INTO TRANS2
  SELECT * FROM TRANS1 WHERE NUM-CC=45
 DELETE FROM CONTO1 WHERE NUM-CC=45
 DELETE FROM TRANS1 WHERE NUM-CC=45
```

- **LINGUAGGIO**: il programmatore deve indicare nella query sia la struttura dei frammenti che la loro allocazione.

### Esempio:

```
SELECT SALDO FROM CONTO1@1
 WHERE NUM-CC=45
 IF (NOT FOUND) THEN
   ( SELECT SALDO FROM CONTO2@2
     WHERE NUM-CC=45
 UNION
   SELECT SALDO FROM CONTO3@3
     WHERE NUM-CC=45 )
```

```
SELECT NUM-CC, PROGR, AMMONTARE
  FROM CONTO1@1 JOIN TRANS1@1 ON .....
 WHERE SALDO < 0
 UNION
 SELECT NUM-CC, PROGR, AMMONTARE
  FROM CONTO2@2 JOIN TRANS2@2 ON .....
 WHERE SALDO < 0
 UNION
 SELECT NUM-CC, PROGR, AMMONTARE
  FROM CONTO3@3 JOIN TRANS3@3 ON .....
 WHERE SALDO < 0
```

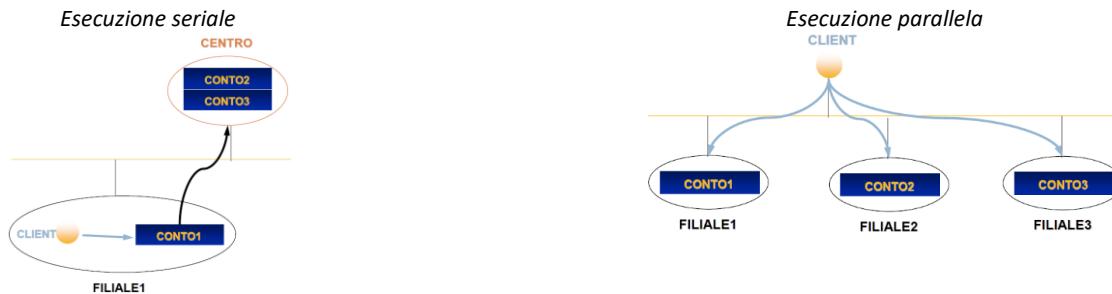
(In modo analogo: 2 copie di DELETE)

Bisogna settare anche la FILIALE a 2

```
INSERT INTO CONTO2@2
  SELECT * FROM CONTO1@1 WHERE NUM-CC=45
 INSERT INTO CONTO2@2
  SELECT * FROM CONTO1@1 WHERE NUM-CC=45
 INSERT INTO TRANS2@2
  SELECT * FROM TRANS1@1 WHERE NUM-CC=45
 INSERT INTO TRANS2@2
  SELECT * FROM TRANS1@1 WHERE NUM-CC=45
 SELECT * FROM TRANS1@3 WHERE NUM-CC=45
```

## EFFICIENZA:

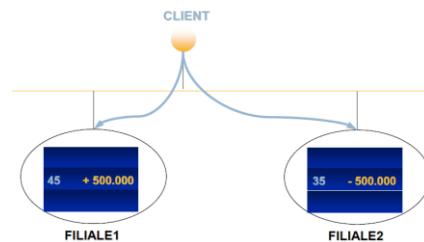
Ottimizzazione delle query. La modalità di esecuzione è **seriale** e **parallela**.



## TRANSAZIONI DISTRIBUITE:

```

BEGIN TRANSACTION
  UPDATE CONTO1@1
  SET SALDO = SALDO + 500.000
  WHERE NUM-CC=45;
  UPDATE CONTO2@2
  SET SALDO = SALDO - 500.000
  WHERE NUM-CC=35;
COMMIT-WORK
END TRANSACTION
  
```

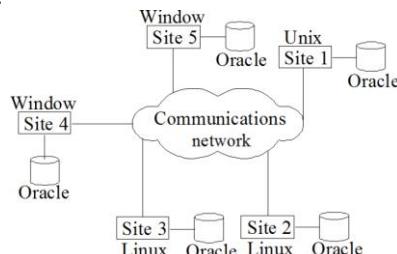


## TIPI DI DDB:

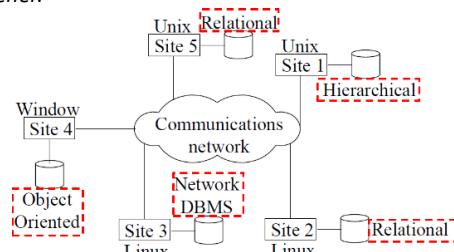
Si differenziano per diversi fattori:

- **Grado di omogeneità** del software dei DDBMS: **DDB omogenei**, server e utenti usano lo stesso software, e **DDB eterogenei**, altrimenti.
- **Grado di autonomia locale**: se il sito locale non può funzionare come un DBMS stand-alone non ha nessuna autonomia locale, e all'utente il sistema "appare" come un DBMS centralizzato. Se alle transazioni locali è permesso l'accesso diretto a un server, ha qualche grado di autonomia locale.

### DDB omogenei:



### DDB eterogenei:



## SISTEMI PER LA GESTIONE DEI DB FEDERATI:

Sistemi di database **federati**: ogni server è un DBMS centralizzato e autonomo con i propri utenti locali, transazioni locali, DBA, e, quindi, un alto grado di autonomia locale. È presente una vista globale o schema della federazione di database. Un sistema multidatabase (DB federati o FDB) non ha uno schema globale e interattivamente ne viene costruito uno (vista globale) in base alle necessità dell'applicazione.

In un FDB eterogeneo, i DB coinvolti possono essere relazionali, reticolari, gerarchici, ecc... Si rende necessario introdurre dei **traduttori di query**.

Vi sono diversi tipi differenti di **eterogeneità**:

- Differenze nei **modelli di dati**.
- Differenze nei **vincoli**: le modalità per la specifica dei vincoli varia da sistema a sistema.
- Differenze nei **linguaggi** di query: anche con lo stesso modello di dati i linguaggi e le loro versioni possono variare.
- Eterogeneità **semantica**: differenze di significato, interpretazione, uso degli stessi dati o di dati correlati.

## SCHEMA A CINQUE LIVELLI (FDBS):

- Lo schema per un gruppo di utenti o un'applicazione.
- Schema globale risultante dall'integrazione di più schemi.
- Sottoinsieme dello schema componente.
- Modello dati comune.
- Schema concettuale.

## CONTROLLO DELLA CONCORRENZA E RECOVERY:

I database distribuiti incontrano un numero di controlli di concorrenza e problemi di recovery che non sono presenti nei database centralizzati, come:

- Trattamento di **copie multiple** di dati: Il controllo della consistenza deve mantenere una **consistenza globale**. Allo stesso modo il meccanismo di recovery deve recuperare tutte le copie e conservare la consistenza dopo il recovery.
- Fallimenti di **singoli siti**: La disponibilità del database non deve essere influenzata dai guasti di uno o due siti e lo schema di recovery li deve recuperare prima che siano resi disponibili.
- Guasto dei **collegamenti di comunicazione**: Tale guasto può portare ad una partizione della rete che può influenzare la disponibilità del database anche se tutti i siti sono in esecuzione.
- **Commit distribuito**: Una transazione può essere frammentata ed essere eseguita su un numero di siti. Questo richiede un approccio basato sul **commit a due fasi** per il commit della transazione.
- **Deadlock distribuito**: Poiché le transazioni sono processate su siti multipli, due o più siti possono essere coinvolti in un **deadlock**. Di conseguenza devono essere considerate le tecniche per il trattamento dei deadlock.
- Controllo della concorrenza distribuito basato su una **copia designata** dei dati: Si designa una particolare copia di ogni dato (**copia designata**). Tutte le richieste di **lock** ed **unlock** vengono inviate solo al sito che la contiene.
- Tecnica del **sito primario**: Un singolo sito è designato come **sito primario**, il quale fa da coordinatore per la gestione delle transazioni.

## TECNICA DEL SITO PRIMARIO:

Nell'ottica della gestione delle transazioni il controllo della concorrenza ed il commit sono gestiti dal **sito primario**. La tecnica del **lock a due fasi (2PL)** è usata per bloccare e rilasciare i data item. Se tutte le transazioni in tutti i siti seguono la politica delle due fasi allora viene garantita la serializzabilità.

- **Vantaggi:** i data item sono bloccati (locked) solamente in un sito ma possono essere usati da qualsiasi altro sito.
- **Svantaggi:** tutta la gestione delle transazioni passa per il sito primario che potrebbe essere **sovraffaticato**. Nel caso in cui il sito primario fallisce, l'intero sistema è inaccessibile.

Per assistere il recovery, un sito di backup viene designato come copia di backup del sito primario. Nel caso di fallimento del sito primario, il sito di backup funziona da sito principale.

## TECNICA DELLA COPIA PRIMARIA:

In questo approccio, invece di un sito, una partizione dei data item è designata come **copia primaria**. Per bloccare un data item soltanto sulla copia primaria di quel data item viene eseguito il lock.

- **Vantaggi:** le copie primarie sono distribuite su vari siti, e un singolo sito **non viene sovraccaricato** da un numero elevato di richieste di lock/unlock.
- **Svantaggi:** l'identificazione della copia primaria è complesso. Una directory distribuita deve essere gestita possibilmente in tutti i siti.

## RECOVERY DAL FALLIMENTO DI UN COORDINATORE:

In entrambi gli approcci un **sito coordinatore** o un **sito copia** possono essere indisponibili. Questo richiede la selezione di un **nuovo coordinatore**.

- **Approccio del sito primario senza sito di backup:** Abortire e far ripartire tutte le transazioni attive in tutti i siti. Si elegge un nuovo coordinatore che inizia il processing delle transazioni.
- **Sito primario con copia di backup:** Sospende tutte le transazioni attive, designa il sito di backup come sito primario e identifica un nuovo sito di backup. Il nuovo sito primario riceve il compito di gestire tutte le transazioni per riprendere il processo.

Se il sito primario e quello di backup falliscono si utilizza un **processo di elezione** per selezionare un nuovo sito coordinatore.

## CONTROLLO DELLA CONCORRENZA BASATA SUL VOTING:

In questo caso **non esiste la copia primaria del coordinatore**. Quindi:

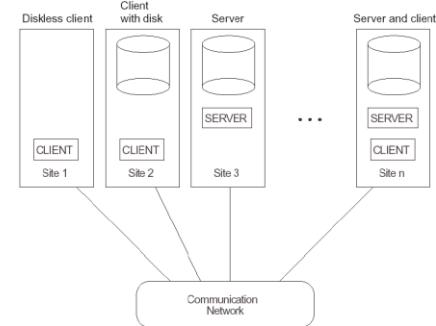
1. Spedire una richiesta di lock ai siti che hanno i data item.
2. Se la maggioranza dei siti concedono il lock allora la transazione richiedente ottiene il data item.
3. Le informazioni di lock (concesse o negative) sono spedite a tutti questi siti.
4. Per evitare un tempo di attesa inaccettabile, viene definito un periodo di **time-out**. Se la transazione richiedente non riceve alcuna informazione di voto allora la transazione viene abortita.

## ARCHITETTURA CLIENT-SERVER:

In un ambiente con un grande numero di PC, stampanti, ecc..., si definiscono dei **server specializzati** con funzionalità specifiche. Ad esempio:

- Il file server mantiene i file delle macchine client.
- Il printer server gestisce la stampa su diverse stampanti.
- L'e-mail server gestisce la posta elettronica.

Le risorse fornite da server specializzati possono essere messe a disposizione di diversi client. Una macchina client fornisce all'utente l'appropriata interfaccia per utilizzare questi server impiegando la potenza di calcolo locale per eseguire applicazioni locali.



## ARCHITETTURA CLIENT-SERVER PER DBMS:

Differenti approcci sono stati proposti su come suddividere le funzionalità tra client e server. Una possibilità è di includere le funzionalità in un DBMS centralizzato a livello server. Un **server SQL** è fornito al client (ogni client), formula la query, fornisce l'interfaccia utente e fornisce funzioni di interfaccia dei linguaggi di programmazione. SQL è uno standard e i server SQL di diversi produttori possono accettare query SQL. I client possono collegarsi al dizionario dei dati che include la distribuzione dei dati fra i vari server SQL.

## INTERAZIONE FRA CLIENT E SERVER:

L'elaborazione di una query avviene:

1. Il client parsa una query e la decomponete in un certo numero di query ai siti indipendenti.
2. Ogni server processa la query locale e manda il risultato al sito client.
3. Il client combina il risultato delle sottoquery per produrre il risultato della query sottomessa in origine.

In questo approccio, il **server SQL** è detto anche **transaction server** o **back-end machine**. Il client è detto **application processor** o **front-end machine**.

In un tipico DDBMS, si è soliti dividere i moduli software su tre livelli:

- **Software server**, responsabile per la gestione dei dati locali di un sito.
- **Software client**, gestisce l'interfaccia utente, accede al catalogo del database e processa tutte le richieste che richiedono l'accesso a più di un sito.
- **Software di comunicazione**, fornisce le primitive di comunicazione che sono usate dal client per trasmettere comandi e dati tra i vari siti (server).

## 10. XMP E BASI DI DATI IN INTERNET

HTML non è idoneo per specificare dati strutturati che sono estratti dal database, quindi viene utilizzato un nuovo linguaggio XML come standard per la strutturazione e lo scambio di dati attraverso il Web. Esso può essere usato per fornire informazioni riguardanti la struttura e il significato dei dati:

- **XML DTD (Document Type Definition)**: linguaggi per la specifica della struttura dei documenti XML.
- **XSL (eXtended Stylesheet Language)**: linguaggio di formattazione per specificare aspetti di formattazione.

### DATI STRUTTURATI, SEMI STRUTTURATI E NON STRUTTURATI:

- **Dati strutturati**: informazioni memorizzate nei database, rappresentate in un formato rigido.
- **Dati non strutturati**: caratterizzati da una presenza molto limitata di informazioni relative ai tipi di dati. Sono dati conservati senza uno schema.
- **Dati semi strutturati**: dati che possono avere una struttura che può essere non identica per tutte le informazioni. Le informazioni sono inframmezzate ai dati. Questi tipi di dati sono anche detti **dati auto descrittivi**. Possono essere rappresentati come **grafi diretti**: i nomi dello schema sono rappresentati dalle **etichette** (o tag) sugli archi orientati, i nodi interni rappresentano oggetti individuali o gli **attributi composti**, i nodi foglia rappresentano i valori degli **attributi di tipo semplice (atomico)**.

### MODELLO DATI XML (AD ALBERO):

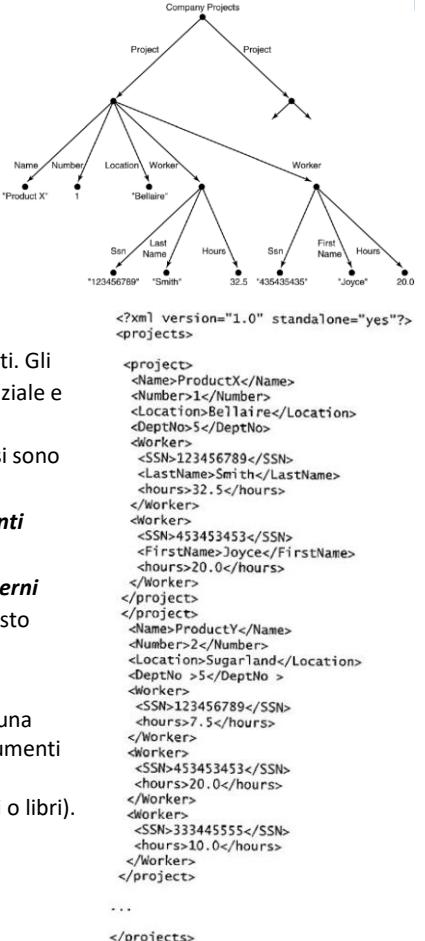
Il **documento XML** è l'oggetto base in XML. Per costruire un documento XML si utilizzano elementi e attributi. Gli **elementi** sono le parti di un documento dotate di un senso proprio. Un elemento è individuato da un tag iniziale e un tag finale. I nomi dei tag sono racchiusi tra `<...>` e i tag di fine sono caratterizzati da uno 'slash' `</...>`. Gli **attributi** sono informazioni aggiuntive sull'elemento che non fanno effettivamente parte del contenuto. Essi sono posti nel tag iniziale dell'elemento.

Gli **elementi complessi** vengono costruiti in modo gerarchico (contengono altri elementi), mentre gli **elementi semplici** contengono solamente valori.

È naturale vedere la corrispondenza tra la rappresentazione XML testuale e la struttura ad albero. I **nodi interni** dell'albero rappresentano gli elementi complessi, i **nodi foglia** rappresentano gli elementi semplici. Per questo motivo il modello XML è chiamato **modello ad albero o modello gerarchico**.

I documenti XML possono essere classificati in tre tipi:

1. Documenti XML **incentrati sui dati**: documenti che hanno molti dati di dimensioni ridotte che seguono una specifica struttura e possono essere estratti da un database strutturato. Essi sono formattati come documenti XML per scambiarli o visualizzarli sul Web.
2. Documenti XML **incentrati sul documento**: documenti con grande quantità di testo (come news, articoli o libri). In questi documenti vi sono pochi, o nessun, dato strutturato.
3. Documenti XML **ibridi**: documenti che possono avere parti che contengono dati strutturati e altre parti prevalentemente testuali o non strutturate.



### Documento XML ben formato (**well-formed**):

- Deve iniziare con la dichiarazione XML per indicare la versione XML utilizzata e altri attributi rilevanti;
- Deve seguire la struttura sintattica del modello ad albero, questo significa che deve contenere un **unico elemento radice**, e che ogni elemento deve contenere una coppia di tag di inizio e fine correlati tra loro, contenuti tra i tag dell'**elemento padre**;
- Un documento ben formato è **sintatticamente corretto**. Questo gli consente di essere processato da generici processori che attraversano il documento e creano la rappresentazione interna ad albero:
  - **DOM (Document Object Model)**: consente ai programmi di manipolare la risultante rappresentazione ad albero corrispondente a documenti XML ben formati. Quando si usa DOM, l'intero documento deve essere analizzato anticipatamente.
  - **SAX**: consente di elaborare i documenti XML "al volo" notificando al programma di elaborazione quando si incontra un tag di inizio o fine.

### Documento XML **valido**:

- Un criterio più forte per un documento è essere valido. Un documento valido è ben formato e deve essere scritto in modo che i nomi utilizzati nelle coppie di tag di inizio e fine siano coerenti con la struttura specificata nel file DTD o in un file XML schema.

### NOTAZIONE XML DTD:

- Un **\*** che segue il nome dell'elemento significa che l'elemento può essere ripetuto **0 o più volte** nel documento. L'elemento in questo caso si dice essere **a valore multiplo (ripetuto) opzionale**: **<!ELEMENT LISTA (PRODOTTO\*)>**
- Un **+** che segue il nome dell'elemento significa che l'elemento può essere ripetuto **1 o più volte** nel documento. L'elemento può essere chiamato **obbligatorio a valore multiplo (ripetuto)**: **<!ELEMENT LISTA (PRODOTTO+)>**
- Un **?** che segue il nome dell'elemento significa che l'elemento può essere ripetuto **0 o 1 volta**. L'elemento può essere chiamato **a valore singolo (non ripetuto) opzionale**: **<!ELEMENT PRODOTTO (DESCRIZIONE?)>**
- Un elemento presente senza **nessuno** dei tre simboli precedenti deve comparire esattamente **1 volta**. Questi elementi sono chiamati **a valore singolo (non ripetuti) necessari**: **<!ELEMENT PRODOTTO (DESCRIZIONE)>**
- Il tipo di un elemento viene specificato tra parentesi di seguito all'elemento stesso. Se le parentesi comprendono nomi di altri elementi, questi ultimi all'interno della struttura ad albero sono i figli dell'elemento a cui le parentesi si riferiscono. Se le parentesi includono la parola chiave **#PCDATA** o uno degli altri dati tipi di dato disponibili in XML DTD, l'elemento è un nodo foglia. PCDATA sta per parsed character data, analogo al tipo di dati stringa.

Elementi con un elemento figlio:

```
<!ELEMENT PRODOTTO (DESCRIZIONE)>
Es:
<PRODOTTO> <DESCRIZIONE>...</DESCRIZIONE> </PRODOTTO>
```

Elementi con una sequenza di elementi figli:

```
<!ELEMENT MAIL (TO, FROM, TITLE, BODY)>
Es:
<MAIL> <TO>...</TO> <FROM>...</FROM> <TITLE>...</TITLE>
<BODY>...</BODY> </MAIL>
```

Elementi con contenuto di tipo PCDATA:

```
<!ELEMENT DESCRIZIONE (#PCDATA)>
Es:
<DESCRIZIONE> Un testo qualsiasi </DESCRIZIONE>
```

- Nella specifica degli elementi le parentesi possono essere nidificate.
- Un simbolo **barra** (`e1|e2`) specifica che nel documento può comparire `e1` o `e2`.

La **dichiarazione di attributi** per ogni elemento dice quali attributi può avere, che valore può assumere, se esiste e qual è il valore di default:

```
<!ATTLIST PRODOTTO
    codice      ID      #REQUIRED
    label       CDATA   #IMPLIED
    status      (disponibile|terminato) 'disponibile'>
```

Contenuto alternativo:

```
<!ELEMENT ARTICOLO (TESTO|FOTO)>
Es:
<ARTICOLO><TESTO> ... </TESTO></ARTICOLO>
<ARTICOLO><FOTO> ... </FOTO></ARTICOLO>
```

Contenuto misto:

```
<!ELEMENT ARTICOLO (#PCDATA|FOTO)*>
```

```
Es:
<ARTICOLO> testo </ARTICOLO>
<ARTICOLO><FOTO> ... </FOTO></ARTICOLO>
```

Elementi vuoti:

```
<!ELEMENT ARTICOLO EMPTY>
```

**Es:**

```
<ARTICOLO/>
```

Contenuto arbitrario:

```
<!ELEMENT NOTA ANY>
```

**Es:**

```
<NOTA> del testo libero</NOTA>
```

```
<NOTA> <AUTORE>Luca</AUTORE> del testo libero</NOTA>
```

## TIPI DI ATTRIBUTI:

- CDATA: dati di tipo carattere;
- (`val1|val2|val3`): un valore nella lista;
- ID: identificatore;
- IDREF, IDREFS: valore di un attributo di tipo ID nel documento (o insieme di valori);
- ENTITY, ENTITIES: nome (nomi) di entità;
- NMOKEN, NMOKENS: caso ristretto di CDATA (una sola parola o insieme di parole)

```
codice      ID      #REQUIRED
label       CDATA   #IMPLIED
status      (disponibile|terminato) 'disponibile'
```

```
<!ELEMENT ELENCO (PRODOTTO+)>
```

```
<!ELEMENT PRODOTTO (DESCRIZIONE, PREZZO?)>
```

```
<!ELEMENT DESCRIZIONE (#PCDATA)>
```

```
<!ELEMENT PREZZO (#PCDATA)>
```

```
<elenco>
```

```
    <prodotto codice="123">
        <descrizione> Forno </descrizione>
        <prezzo> 1040000 </prezzo>
    </prodotto>
    <prodotto codice="432">
        <descrizione> Frigo </descrizione>
    </prodotto>
</elenco>
```

```
<!ELEMENT FAX(..)>
```

```
<!ATTLIST FAX Mittente CDATA #FIXED "Politecnico di Milano">
```

```
<!ELEMENT SCONTI EMPTY>
```

```
<!ATTLIST SCONTI Valore CDATA "10">
```

```
<SCONTO/> → vale 10
```

```
<SCONTO valore="15"/> → vale 15
```

## VINCOLI SUGLI ATTRIBUTI:

- #REQUIRED: il valore deve essere specificato;
- #IMPLIED: il valore può mancare;
- #FIXED "valore": se presente, deve coincidere con "valore";
- Default: si può specificare un valore come default, usato quando l'attributo è mancante.

### Esempio DTD:

```
<!DOCTYPE NEWSPAPER [
    <!ELEMENT NEWSPAPER (ARTICLE+)>
    <!ELEMENT ARTICLE
        (HEADLINE,BYLINE,LEAD,BODY,NOTES)>
    <!ELEMENT HEADLINE (#PCDATA)>
    <!ELEMENT BYLINE (#PCDATA)>
    <!ELEMENT LEAD (#PCDATA)>
    <!ELEMENT BODY (#PCDATA)>
    <!ELEMENT NOTES (#PCDATA)>
    <!ATTLIST ARTICLE AUTHOR CDATA #REQUIRED>
    <!ATTLIST ARTICLE EDITOR CDATA #IMPLIED>
    <!ATTLIST ARTICLE DATE CDATA #IMPLIED>
    <!ATTLIST ARTICLE EDITION CDATA #IMPLIED>
] >
```

Un file DTD chiamato *projects*:

```
<!DOCTYPE projects [
    <!ELEMENT projects (project+)>
    <!ELEMENT project (Name, Number, Location, DeptNo?, Workers)>
    <!ELEMENT Name (#PCDATA)>
    <!ELEMENT Number (#PCDATA)>
    <!ELEMENT Location (#PCDATA)>
    <!ELEMENT DeptNo (#PCDATA)>
    <!ELEMENT Workers (Worker*)>
    <!ELEMENT Worker (SSN, LastName?, FirstName?, hours)>
    <!ELEMENT SSN (#PCDATA)>
    <!ELEMENT LastName (#PCDATA)>
    <!ELEMENT FirstName (#PCDATA)>
    <!ELEMENT hours (#PCDATA)>
] >
```

## LIMITAZIONE DEGLI XML DTD:

I tipi di dati del DTD non sono molto generali. DTD ha la sua particolare sintassi e pertanto richiede processori specializzati. Sarebbe vantaggioso specificare lo schema dei documenti XML usando le regole sintattiche di XML stesso in modo tale che gli stessi processori dei documenti XML possano processare le descrizioni dello schema XML. Tutti gli elementi DTD sono sempre forzati a seguire l'ordine del documento, pertanto non sono permessi elementi non ordinati. Un documento XML può dipendere da un solo DTD che definisce tutta la grammatica applicabile.

## XML SCHEMA:

Lo scopo è definire gli elementi e la composizione di un documento XML in modo più efficace del DTD. **XML Schema** definisce regole riguardanti: elementi, attributi, gerarchia degli elementi, sequenza di elementi figli, cardinalità di elementi figli, tipi di dati per elementi e attributi, valori di default per elementi e attributi.

### XSD (XML SCHEMA DEFINITION):

L'**XSD** (XML Schema Definition) definisce uno schema di un tipo di documenti. Gli XSD sono estendibili (ammettono tipi riusabili definiti dall'utente), in formato XML, più ricchi e completi dei DTD, capaci di supportare tipi di dati diversi da PCDATA, capaci di gestire namespace multipli.

Documento XML con riferimento a XSD:

- **xmlns**: namespace di default.
- **xmlns:xsi**: URI (Universal Resource Identifier) che introduce l'uso dei tag di XML Schema.
- **xsi:schemaLocation**: dichiara dove reperire il file XSD (sempre attraverso URI).

```
<?xml version="1.0"?>
<note
    xmlns="http://www.w3schools.com"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.w3schools.com/note.xsd">
    <to> Tove </to>
    <from> Jani </from>
    <head> Reminder </head>
    <body> Don't forget me this weekend! </body>
</note>
```

## NAMESPACE XML:

È necessario identificare il particolare insieme di elementi (tag) del linguaggio XML schema usati specificando un file memorizzato in un sito Web.

La seconda linea nell'esempio specifica il file usato. In particolare:

- "<http://www.w3.org/2001/XMLSchema>"

Ogni definizione di questo tipo è detta **namespace XML**.

Il nome del file è assegnato alla **variabile xs** usando l'attributo xmlns (XML namespace), e questa variabile è usata come prefisso di tutti i comandi di XML schema.

```
<?xml version="1.0"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="head" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

xs: namespace per XSchema  
(contiene tutti i tag XSD)

## ELEMENTI SEMPLICI:

Possono contenere solo testo (no elementi o attributi), oppure i tipi: stringhe, numerici, date, time, boolean, ecc...

Esempi di definizione di elementi semplici in XSD:

```
<xs:element name="età" type="xs:integer"/>
<xs:element name="cognome" type="xs:string"/>
```

Esempi di elementi semplici XML:

```
<età> 65 </età>
<cognome> Rossi </cognome>
```

## ATTRIBUTI:

Definizione di attributi:

```
<xs:attribute name="name" type="type"/>
```

```
<xs:attribute ... default="xyz" use="required|optional"/>
```

Valore di default o fisso

Obligatorio o opzionale

Esempio di definizione di attributi:

```
<xs:attribute name="lang" type="xs:string"/>
```

Esempio di uso di un attributo:

```
<lastname lang="it"> qwertys </lastname>
```

## ELEMENTI COMPLESSI (COMPLEX ELEMENTS):

Esistono quattro tipi di elementi complessi:

- vuoti (empty)
- contenenti solo altri elementi
- contenenti solo testo
- contenenti testo e/o altri elementi
 

```
<xs:element name="name">
        <xs:complexType>
          . . . element content . .
        </xs:complexType>
      </xs:element>
```

## COSTRUTTO ALL:

Come la sequenza ma gli elementi possono apparire nel documento in qualsiasi ordine:

```
<xs:element name="libro">
  <xs:complexType>
    <xs:all>
      <xs:element name="titolo" type="xs:string"/>
      <xs:element name="autore" type="xs:string"
        maxOccurs="unbounded"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```

## COSTRUTTO CHOICE:

Sequenza (OR), gli elementi appaiono in alternativa nel documento:

```
<xs:complexType>
  <xs:choice>
    <xs:element name="capitolo" type="xs:string"
      maxOccurs="unbounded"/>
    <xs:element name="appendice" type="xs:string"
      maxOccurs="unbounded"/>
  </xs:choice>
</xs:complexType>
```

## SPECIFICA DELLA CARDINALITÀ:

Indica la cardinalità dei sotto elementi. La sintassi prevede l'utilizzo di due attributi:

- **maxOccurs**: massimo numero di occorrenze. Per gli elementi multivaleo si deve impostare **maxOccurs="unbounded"**;
- **minOccurs**: minimo numero di occorrenze. Per i valori nulli si deve impostare **minOccurs=0**.

Se non specificati, default=1:

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="full_name" type="xs:string"/>
      <xs:element name="child_name" type="xs:string"
        maxOccurs="10" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

## ANNOTATIONS AND DOCUMENTATION:

Gli elementi **xsd:annotation** e **xsd:documentation** sono utilizzati per fornire commenti ed altre descrizioni nel documento XML.

L'attributo **xml:lang** dell'elemento **xsd:documentation** specifica il linguaggio usato ("en" specifica la lingua inglese).

## SPECIFICA DELLE CHIAVI:

Per specificare una chiave primaria è utilizzato il tag **xsd:key**. Per specificare le chiavi esterne è usato il tag **xsd:keyref**. Quando si specifica una chiave esterna, l'attributo **refer** del tag **xsd:keyref** indica la chiave primaria referenziata, mentre i tag **xsd:selector** e **xsd:field** specificano il tipo dell'elemento riferito e la chiave esterna.

## COSTRUTTO SEQUENCE:

Sequenza (record), gli elementi devono apparire nell'ordine indicato:

```
<xs:element name="libro">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="titolo" type="xs:string"/>
      <xs:element name="autore" type="xs:string"
        maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

La sequenza può contenere sia sotto elementi che attributi.

## ELEMENTI EMPTY:

Gli elementi senza sotto elementi interni si dichiarano come complexType privi di sotto elementi (element):

```
<xs:element name="product">
  <xs:complexType>
    <xs:attribute name="prodid" type="xs:integer"/>
  </xs:complexType>
</xs:element>
```

## DEFINIZIONE DI TIPI RIUSABILI:

Consente di definire tipi e riusarli per tipare attributi oppure definire altri tipi. Esempio di definizione di tipo (non di elemento):

```
<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Esempio di riuso del tipo per definire elementi/attributi:

```
<xs:element name="employee" type="personinfo"/>
<xs:element name="student" type="personinfo"/>
<xs:attribute name="member" type="personinfo"/>
```

## INTERROGAZIONI IN XML:

Come linguaggi di interrogazione XML sono emersi, tra le varie proposte, due standard:

- **XPath**: fornisce dei costruttori di linguaggio per la specifica di path expression, volti a identificare determinati nodi (o elementi) che corrispondono a particolari pattern contenuti nel documento XML.
- **XQuery**: linguaggio di interrogazione più generale che utilizza le espressioni XPath ma che possiede costrutti aggiuntivi.

## XPATH:

Una espressione XPath restituisce una collezione di nodi che soddisfa pattern specificati in essa. I nomi della espressione XPath sono i nomi dei nodi dell'albero del documento XML, che possono essere i nomi degli elementi o degli attributi, con l'aggiunta di condizioni di qualifica per limitarne il numero di risultati. Nella specifica di un path sono utilizzati due separatori:

- **Slash (/)**: inserito prima di un tag specifica che quel tag deve essere un figlio diretto del tag precedente;
- **Doppio slash (//)**: indica che il tag può essere un discendente di qualunque livello.

Si include il nome del file in ogni interrogazione XPath.

## PATH EXPRESSION IN XPATH:

L'idea è di usare una sintassi per "navigare" la struttura ad albero di un documento. Una espressione XPath è una stringa contenente nomi di elementi e operatori di navigazione e selezione:

- . : nodo corrente;
- .. : nodo padre del nodo corrente;
- / : nodo radice, o figlio del nodo corrente;
- // : discendente del nodo corrente;
- @ : attributo del nodo corrente;
- \* : qualsiasi nodo;
- [p]: predicato (se l'espressione p, valutata, ha valore booleano);
- [n]: posizione (se l'espressione n, valutata, ha valore numerico).

Una path expression può iniziare con [doc\(posizione\\_documento\)](#) che restituisce l'elemento radice del documento specificato e tutto il suo contenuto.

A partire dalla radice del documento si possono specificare delle espressioni per estrarre il contenuto desiderato. Es: [doc\("libri.xml"\)/Elenco/Libro](#)

Restituisce la sequenza di tutti gli elementi di tipo Libro contenuti nel documento "libri.xml".

### CONDIZIONI SU ELEMENTI/ATTRIBUTI:

Esempio:

[doc\("libri.xml"\)/Elenco/Libro\[Editore='Bompiani'\]/Titolo](#)

Restituisce la sequenza di tutti i titoli dei libri dell'editore 'Bompiani' che si trovano nel documento.

### RICERCA DI SOTTO-ELEMENTI A QUAISIASI

#### LIVELLO:

Esempio: [doc\("libri.xml"\)//Autore](#)

Restituisce la sequenza di tutti gli autori che si trovano nel documento "libri.xml", annidati a qualunque livello.

### CONDIZIONE SULLA POSIZIONE DEI SOTTO-ELEMENTI A USO DI WILDCARD:

Esempio: [doc \("libri.xml"\)/Elenco/Libro\[2\]/\\*](#)

Restituisce tutti i sotto elementi (\*) contenuti nel secondo libro (Libro[2]) del documento "libri.xml".

### ESEMPI XPATH:

1. **/company**: restituisce il nodo radice di **company** e tutti i suoi nodi discendenti, ciò implica che restituisce l'intero documento XML;
2. **/company/department**: restituisce tutti i nodi **department** ed il loro sottoalbero;
3. **//employee[employeeSalary gt 70000]/employeeName**: restituisce tutti i nodi **employeeName** che sono figli diretti del nodo **employee**, dei nodi **employee** che abbiano un figlio **employeeSalary** il cui valore è maggiore di 70000;
4. **/company/employee[employeeSalary gt 70000]/employeeName**: restituisce lo stesso risultato del (3), ma specificando l'intero nome del path;
5. **/company/project/projectWorker[hours ge 20.0]**: restituisce tutti i nodi **projectWorker** e i loro sottoalberi dei nodi che abbiano il nodo **hours** maggiore o uguale di 20.0

## XQUERY:

Usa le espressioni XPath ma ha ulteriori costrutti. Una interrogazione XQuery è un'espressione complessa che consente di estrarre parti di un documento e costruire un altro documento. XQuery consente di specificare interrogazioni su uno o più documenti XML. La forma tipica di una interrogazione XQuery è conosciuta come **espressione FLWR**, che indica le principali clausole di XQuery:

- **FOR**: itera i valori delle variabili su sequenze di nodi;
- **LET**: variabili legate a collezioni di nodi;
- **WHERE**: esprime condizioni di qualificazione sui legami;
- **RETURN**: specificazione del risultato dell'interrogazione

### ESPRESSIONI FLWOR:

In una singola XQuery si possono avere **0 o più** istanze della clausola **FOR**, **0 o più** istanze della clausola **LET**, **0 o 1** istanza di ciascuna delle clausole **WHERE** e **ORDER**, esattamente **1** istanza della clausola **RETURN**. Le variabili sono prefissate da \$, la clausola LET assegna una variabile a una particolare espressione.

### ESPRESSIONE FOR:

La clausola **for** valuta la path expression ([doc\("libri.xml"\)//Libro](#)), che restituisce una sequenza di elementi, e la variabile **\$libro** itera all'interno della sequenza, assumendo ad ogni iterazione il valore di un nodo diverso.

La clausola **return** costruisce il risultato, in questo caso restituisce ogni valore legato a **\$libro**, cioè tutti i libri del documento.

**Semantica**: per ogni valore di **\$libro**, per ogni valore di **\$autore** del libro corrente, inserisci nel risultato il **\$autore**.

Esempio:

```
for $libro in doc("libri.xml")//Libro  
return $libro
```

Le espressioni FOR possono essere annidate:

```
for $libro in doc("libri.xml")//Libro  
for $autore in $libro/Autore  
return $autore
```

## ESPRESSIONE LET:

Consentono di introdurre nuove variabili. Esempio:

```
let $libri := doc("libri.xml")//Libro  
return $libri
```

La clausola **let** valuta l'espressione, **//Libro**, e assegna alla variabile **\$libri** l'intera sequenza restituita.

La valutazione di una clausola let assegna alla variabile un singolo valore, cioè l'intera sequenza dei nodi che soddisfano l'espressione.

## CLAUSOLA RETURN:

Genera l'output di un'espressione FLWR che può essere:

- un nodo:

```
(<Autore>F. Dürrenmatt</Autore>)
```

- una foresta ordinata di nodi:

```
(<Autore>J.R.R. Tolkien</Autore>  
<Autore>Umberto Eco</Autore>  
<Autore>F. Dürrenmatt</Autore>)
```

- un valore testuale (PCDATA):

F. Dürrenmatt

Può contenere dei costruttori di nodi, dei valori costanti, riferimenti a variabili definite nelle parti FOR e LET, ulteriori espressioni annidate. Un **costruttore di elemento** consta di un tag iniziale e di un tag finale che racchiudono una lista opzionale di espressioni annidate che ne definiscono il contenuto.

## ORDINARE IL RISULTATO:

I libri vengono ordinati rispetto al titolo. I matching della variabile sono riordinati prima di essere passati alla clausola **return** per generare il risultato. Esempio:

```
for $libro in doc("libri.xml")//Libro  
order by $libro/Titolo  
return  
  <Libro>  
    { $libro/Titolo,  
      $libro>Editore }  
  </Libro>
```

## ESEMPIO DI XQUERY:

```
FOR $x IN  
doc(www.company.com/info.xml)/company/project[projectNumber=5]/projectWorker,  
$y IN doc(www.company.com/info.xml)/company/employee  
WHERE $x/hours gt 20.0 AND $y.ssn=$x.ssn  
RETURN <res> $y/employeeName/firstName, $y/employeeName/lastName, $x/hours </res>
```

Questa query illustra come può essere eseguita un'operazione di **join** usando più di una variabile. La variabile **\$x** è legata a ogni elemento **projectWorker** che è figlio del progetto con **projectNumber=5**, mentre la variabile **\$y** è legata a ogni elemento **employee**. La condizione di join confronta i valori **ssn** al fine di recuperare nome e cognome dell'impiegato e numero di ore.

## CLAUSOLA WHERE:

La clausola **WHERE** esprime una condizione, solo le tuple che soddisfano tale condizione vengono utilizzate per invocare la clausola **RETURN**. Le condizioni nella clausola WHERE possono contenere diversi predicati connessi da AND o OR. Il **not()** è realizzato tramite una funzione che inverte il valore di verità. Esempio:

```
for $libro in doc("libri.xml")//Libro  
where $libro/Editore="Bompiani"  
  and $libro/@disponibilità="S"  
return $libro
```

Restituisce tutti i libri con Editore='Bompiani' e disponibilità='S'.

Esempio:

```
for $libro in doc("libri.xml")//Libro  
where $libro/Editore="Bompiani"  
return <LibroBompiani>  
  { $libro/Titolo }  
</LibroBompiani>
```

nuovo  
elemento

espressione  
annidata

Restituisce:

```
<LibroBompiani><Titolo>Il Signore degli Anelli</Titolo></LibroBompiani>  
<LibroBompiani><Titolo>Il nome della rosa</Titolo></LibroBompiani>
```

## FUNZIONE DI AGGREGAZIONE:

Restituisce gli editori con più di 100 libri in elenco. La "cardinalità" del risultato, cioè il numero di editori restituiti, dipende da quante volte è eseguita la return, e questo, a sua volta, dipende dalla clausola for. (Ogni editore risultante, viene restituito oltre cento volte). Esempio:

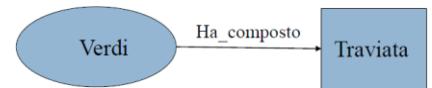
```
for $e in doc("libri.xml")//Editore  
let $libro := doc("libri.xml")//Libro[Editore = $e]  
where count($libro) > 100  
return $e
```

## 11. BASI DI DATI SEMANTICHE, SPARQL E LINKED OPEN DATA

Presente fin dagli inizi dell'evoluzione del settore. L'uso dei modelli semanticci è alla base di vari processi, quali: Durante la progettazione concettuale, Gestione dei dati e delle interrogazioni ad un livello astratto, Sistemi che supportano direttamente i modelli semanticci. Spinta allo sviluppo degli aspetti semanticci direttamente nei sistemi di gestione dati è stata: Evoluzione dei linguaggi e delle basi dati ad oggetti verso la fine degli anni Ottanta, Evoluzione dei <<Semantic Web>>. Sviluppo delle cosiddette raccolte aperte e connesse <<Linked Open Data>>.

### MODELLO RDF (RESOURCE DESCRIPTION FRAMEWORK):

È il modello astratto proposto dal W3C per esprimere affermazioni sul mondo. Le informazioni vengono rappresentate sotto forma di triple, che possono essere interpretate come **soggetto-verbo-oggetto**. Una tripla può essere rappresentata come un grafo, in cui i nodi corrispondono ai soggetti e agli oggetti, gli archi ai verbi.



### DATI RDF:

I dati rappresentabili in RDF sono di tre categorie:

- **IRI** (Internationalized Resource Identifier): sono stringhe che identificano univocamente tutte le risorse disponibili sul Web.
- **Letterali**: descrivono le risorse presenti nell'istanza RDF. È possibile assegnare ai letterali un tipo, se inclusi tra apici sono implicitamente di tipo *stringa*, mentre numeri senza apici, con segno, sono implicitamente di tipo *intero*.
- **Blank Nodes**: identificatori usati per rappresentare risorse anonime.

Nelle triple RDF il primo termine (soggetto) può essere un IRI o un blank node, il secondo termine (predicato) è un IRI, e il terzo termine (oggetto) può essere un IRI, un letterale o un blank node.

### NAMESPACE NEL MODELLO RDF:

È possibile utilizzare uno o più vocabolari per i termini usati nell'istanza RDF, introdotti dall'istruzione <<prefix>> che li associa ad un prefisso da usare nelle definizioni e query: `@prefix o: http://www.sitoweb.com/path`

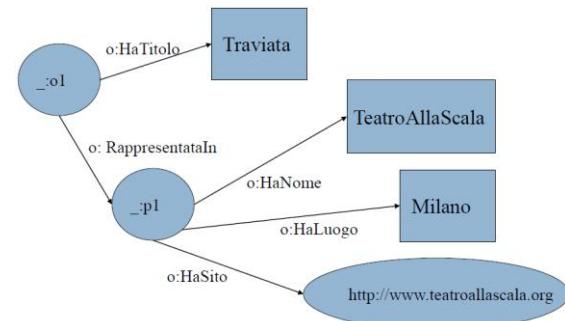
Nel definire un'istanza RDF, si possono indicare più vocabolari.

#### Esempio istanza RDF:

##### Istanza RDF: Notazione N3

```
_:o1 o:HaTitolo 'Traviata';
      o:RappresentataIn _:p1.
_:p1 o:HaNome 'TeatroAllaScala';
      o:HaLuogo 'Milano';
      o:HaSito http://www.teatroallascala.org.
```

Viene adottata la notazione **N3**, che utilizza la punteggiatura per sintetizzare le parti comuni delle triple. Quando due triple condividono il soggetto si usa il separatore '. Quando condividono sia soggetto sia predicato, si utilizza il separatore ';



```
<?xml version="1.0"?>
<rdf:RDF xmlns:o="http://www.polimi.it/ceri/ope...
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:ceri="http://www.polimi.it/ceri/">
<rdf:Description>
  <ceri:operaHaTitolo>Traviata</ceri:operaHaTitolo>
  <ceri:operaRappresentataIn>
    <rdf:Description>
      <ceri:operaHaNome>TeatroAllaScala</ceri:operaHaNome>
      <ceri:operaHaLuogo>Milano</ceri:operaHaLuogo>
      <ceri:operaHaSito rdf:resource="http://www.teatroallascala.org"/>
    </rdf:Description>
  </ceri:operaRappresentataIn>
</rdf:Description>
</rdf:RDF>
```

```
  _:Valchiria o:haTitolo 'Valchiria' ;
    o:dura 310 ;
    o:haAtti 3 ;
    o:haPersonaggio 'Brunilde' ;
    o:haPersonaggio 'Wotan' ;
    o:haPersonaggio 'Sieglinde' .
```

```
  _:Sigfrido o:haTitolo 'Sigfrido' ;
    o:dura 320 ;
    o:haAtti 3 ;
    o:haPersonaggio 'Brunilde' ;
    o:haPersonaggio 'Sigfrido' .
```

```
  _:Crepuscolo o:haTitolo 'Crepuscolo degli Dei' ;
    o:dura 360 ;
    o:haAtti 3 ;
    o:haPrologo 'si' ;
    o:haPersonaggio 'Brunilde' ;
    o:haPersonaggio 'Sigfrido' ;
    o:haPersonaggio 'Waltraute' .
```

#### Altri esempi:

```
@prefix o: <http://www.polimi.it/ceri/ope...
  _:Ring o:compostoDa 'Wagner' .
  _:Ring o:prodottoDa 'ScalaDiMilano' .
  _:Ring o:condottoDa 'DanielBarenboim' .
  _:Ring o:messolnScenaDa 'GuyCassiers' .
  _:Ring o:haTitolo 'Ring'.
```

```
  _:Ring o:comprende _:OroDelReno,
    _:Valchiria,
    _:Sigfrid,
    _:Crepuscolo.
```

```
  _:OroDelReno o:haTitolo 'Oro Del Reno' ;
    o:dura 150 ;
    o:haPersonaggio 'Wotan' .
```

## RDF SCHEMA:

Estensione di RDF come raccomandazione del W3C dal 2004. Esso aggiunge a RDF costrutti per descrivere metadati, ovvero struttura e proprietà di istanze RDF. I principali costrutti messi a disposizione da RDFS sono le classi e le proprietà:

- Una classe RDFS consente di definire insiemi di risorse di un'istanza RDF che hanno caratteristiche comuni.
- Una proprietà RDFS consente invece di definire il soggetto e l'oggetto dei predicati in un'istanza RDF.

## RDF SCHEMA - COSTRUTTI:

- **rdfs:Resource**, tutto ciò che viene descritto in RDF è detto **risorsa**. Ogni risorsa è istanza della classe rdfs:Resource.
- **rdfs:Literal**, sottoclasse di rdfs:Resource, rappresenta un letterale, una stringa di testo.
- **rdf:Property**, rappresenta le proprietà. È sottoclasse di rdfs:Resource.
- **rdfs:Class**, corrisponde al concetto di *tipo* e di *classe* della programmazione object-oriented. Quando viene definita una nuova classe, la risorsa che la rappresenta deve avere la proprietà rdf:type impostata a rdfs:Class.
- **rdfs:subClassOf**, specifica la relazione di ereditarietà fra classi. Questa proprietà può essere assegnata solo a istanze di rdfs:Class. Una classe può essere sottoclasse di una o più classi (*ereditarietà multipla*)
- **rdfs:range**(codominio), usato come predicato di una risorsa r, indica le classi che saranno oggetto di un'asserzione che ha r come predicato.
- **rdfs:domain** (dominio), usato come predicato di una risorsa r, indica le classi (soggetto) a cui può essere applicata r.
- **rdfs:subPropertyOf**, specifica la relazione di ereditarietà fra proprietà.

### Esempi RDF schema:

Esempio di classe:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix o: <http://www.polimi.it/ceri/opera> .
o:Compositore rdf:type rdfs:Class.
_:Wagner rdf:type o:Compositore.
o:nome a rdfs:Property;
rdfs:domain o:Compositore;
rdfs:range rdfs:Literal.
_:Ring a o:Opera.
o:compostoDa a rdfs:Property;
rdfs:domain o:Opera;
rdfs:range o:Compositore.
o:Artista rdf:type rdfs:Class.
o:Compositore rdfs:subClassOf o:Artista.
```

Esempio di proprietà:

È possibile definire in RDFS relazioni di generalizzazione tra classi:

La semantica di RDFS definisce l'ereditarietà: se i compositori sono artisti e gli artisti sono persone, si deduce che i compositori sono persone; se le persone possiedono come proprietà una città e una data di nascita, anche i direttori d'orchestra hanno queste proprietà. Questi meccanismi di inferenza vengono chiamati **entailments**.

## ALTRE ASTENSIONI SEMANTICHE:

Una maggior ricchezza semantica rispetto ad RDFS è offerta dal **Web Ontology Language (OWL)**. Esso è un linguaggio di markup per rappresentare esplicitamente le *ontologie* (ovvero il significato dei termini e relazioni tra i termini). L'obiettivo è supportare l'elaborazione automatica del contenuto delle informazioni dei documenti scritti in OWL ed il *reasoning* su di essi. Esistono tre versioni:

- OWL Lite pone delle restrizioni sui costrutti OWL utilizzabili che garantiscono un calcolo efficiente delle inferenze.
- OWL DL estende OWL Lite garantendo solo che tutte le inferenze possono essere calcolate in un tempo finito.
- OWL Full fornisce il massimo dell'espressività senza garantire la computabilità completa degli entailments.

## SPARQL (SIMPLE PROTOCOL AND RDF QUERY LANGUAGE):

È un linguaggio di interrogazione simile a SQL proposto dal W3C per interrogare dati descritti in RDF, ricevendo il risultato in un formato XML.

Esistono quattro tipi di query:

- **SELECT** per interrogazioni classiche.
- **DESCRIBE**, per ottenere una descrizione delle risorse presenti presso un database RDF interrogabili in SPARQL (detto <>).
- **ASK**, per sapere se specifici termini sono disponibili nell'endpoint.
- **CONSTRUCT**, per costruire un nuovo grafo RDF a partire da una interrogazione.

Vi sono due versioni del linguaggio: **1.0** (raccomandazione W3C del 2008) e **1.1** (nuova versione più potente del 2013).

## SINTASSI SPARQL:

Una query **SPARQL 1.0** è composta da cinque parti:

- La clausola opzionale PREFIX per introdurre vocabolari.
- Il risultato prodotto dalla query (una delle quattro clausole SELECT, DESCRIBE, ASK, e CONSTRUCT).
- Gli endpoint consultati, tramite le clausole FROM e FROM NAMED.
- La parte centrale della query, introdotta dalla clausola WHERE consente di esprimere condizioni di pattern matching tra la query stessa e il grafo RDF su cui la query opera. Elemento centrale del pattern matching è il **triple pattern**.
- Modificatori opzionali, che includono i costrutti ORDER BY, DISTINCT e LIMIT.

## TRIPLE E GRAPH PATTERNS:

Un **triple pattern** è una tripla nelle cui posizioni è possibile far comparire, in aggiunta a IRI, letterali e blank nodes, anche variabili, introdotte dal simbolo '?', durante la valutazione delle query le variabili vengono <> (binding). Ad esempio: <?o1 o:HaPersonaggio 'Sigfrido'>

Valutato sull'istanza RDF, la variabile ?o1 è legata ai blank node \_:Sigfrido e \_:Crepuscolo.

Possono essere combinati insieme vari triple pattern, formando un **graph pattern**.

## QUERY SPARQL:

Estrarre i nomi delle opere che hanno Wotan come personaggio:

```
PREFIX o: <http://www.polimi.it/ceri/ope>
SELECT ?t
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
    ?o o:haPersonaggio 'Wotan' .
    ?o o:haTitolo ?t .
}
?t
'Oro del Reno'
'Valchiria'
```

Estrarre i titoli delle opere e i nomi dei personaggi di tutte le coppie di opere che hanno un personaggio in comune:

```
PREFIX o: <http://www.polimi.it/ceri/ope>
SELECT ?t1, ?t2, ?p
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
    ?o1 o:haPersonaggio ?p .
    ?o2 o:haPersonaggio ?p .
    ?o1 != ?o2 .
    ?o1 o:haTitolo ?t1 .
    ?o2 o:haTitolo ?t2 .
}
?t1
?t2
?p
'Oro del Reno'
'Valchiria'
'Wotan'
'Valchiria'
'Oro del Reno'
'Wotan'
'Valchiria'
'Sigfrido'
'Valchiria'
'Brunilde'
'Sigfrido'
'Valchiria'
'Brunilde'
...
...
```

Estrarre le opere che hanno Wotan come personaggio oppure durano più di 350 minuti:

```
PREFIX o: <http://www.polimi.it/ceri/ope>
SELECT ?
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
    ?o o:haTitolo ?t .
    ?o o:haPersonaggio ?p .
}
FILTER (?p = 'Wotan') .
UNION
{
    ?o o:haTitolo ?t .
    ?o o:dura ?m .
}
FILTER (?m > 350) .
}
?t
'Oro del Reno'
'Valchiria'
'Valchiria'
'Crepuscolo degli Dei'
```

Estrarre il titolo di tutte le opere che hanno qualche personaggio diverso da Sigfrido:

```
PREFIX o: <http://www.polimi.it/ceri/ope>
SELECT ?
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
    ?o o:haTitolo ?t
    ?o o:haPersonaggio ?p
}
FILTER {
    ?p != 'Sigfrido'
}
?t
'Oro del Reno'
'Valchiria'
'Sigfrido'
'Crepuscolo degli Dei'
```

## QUESRY SPARQL – CONSTRUCT:

Costruire nuove tuple RDF a partire dall'istanza RDF e costruendo legami per le variabili ?t e ?p, in particolare le coppie il cui primo elemento è il nome Ring e il secondo elemento è un personaggio di una delle quattro opere che lo compongono.

## QUESRY SPARQL – ASK:

Valuta se una query ha un risultato non nullo, cioè se alle variabili della query viene associata almeno una tupla di binding.

```
PREFIX o: <http://www.polimi.it/ceri/ope>
ASK
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
    ?o o:haTitolo 'Walkiria' .
    ?o o:haPrologo ?p .
}
```

La query ASK ha valore **false**.

Estrarre i titoli delle opere e i nomi dei personaggi che hanno gli stessi personaggi che compaiono in Valchiria:

```
PREFIX o: <http://www.polimi.it/ceri/ope>
SELECT ?t, ?p
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
    ?o1 o:haPersonaggio ?p .
    ?o2 o:haPersonaggio ?p .
    ?o1 != ?o2 .
    ?o1 o:haTitolo ?t .
    ?o2 o:haTitolo 'Valchiria' .
}
?t
?p
'Oro del Reno'
'Wotan'
'Sigfrido'
'Brunilde'
'Crepuscolo degli Dei'
'Brunilde'
```

Estrarre i titoli delle opere e i nomi dei personaggi che hanno gli stessi personaggi che compaiono in Valchiria e filtrare le opere la cui durata è superiore a 300 minuti:

```
PREFIX o: <http://www.polimi.it/ceri/ope>
SELECT ?t, ?p
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
    ?o1 o:haPersonaggio ?p .
    ?o2 o:haPersonaggio ?p .
    ?o1 != ?o2 .
    ?o1 o:haTitolo ?t .
    ?o2 o:haTitolo 'Valchiria' .
}
?t
?p
'Sigfrido'
'Brunilde'
'Crepuscolo degli Dei'
'Brunilde'
FILTER (?d > 300).
}
```

Estrarre estrae titolo, durata, numero di atti e presenza di prologo di tutte le opere:

```
PREFIX o: <http://www.polimi.it/ceri/ope>
SELECT ?, ?d, ?, ?
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
    ?o o:haTitolo ?t .
    ?o o:haDurata ?d .
    OPTIONAL {
        ?o o:haAtti ?a .
        ?o o:haPrologo ?p .
    }
}
?t
?d
?a
?p
'Oro del Reno'
150
NULL
NULL
'Valchiria'
310
3
NULL
'Sigfrido'
320
3
NULL
'Crepuscolo degli Dei'
360
3
'si'
```

Estrarre le opere in cui non è presente il personaggio Sigfrido:

```
PREFIX o: <http://www.polimi.it/ceri/ope>
SELECT ?
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
    ?o o:haTitolo ?t
    OPTIONAL {
        ?o o:haPersonaggio ?p .
    }
    FILTER (?p = 'Sigfrido') .
}
?t
'Oro del Reno'
'Valchiria'
```

PREFIX o: <http://www.polimi.it/ceri/ope>

```
CONSTRUCT { ?t haPersonaggio ?p . }
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
    ?r o:haTitolo ?t .
    ?r o:comprende ?o .
    ?o o:haPersonaggio ?p .
}
?Ring haPersonaggio 'Wotan'.
?Ring haPersonaggio 'Brunilde'.
?Ring haPersonaggio 'Wotan'.
?Ring haPersonaggio 'Sieglinde'.
?Ring haPersonaggio 'Brunilde'.
?Ring haPersonaggio 'Sigfrido'.
?Ring haPersonaggio 'Brunilde'.
?Ring haPersonaggio 'Sigfrido'.
?Ring haPersonaggio 'Waltraute'.
```

Valuta se una query ha un risultato non nullo, cioè se alle variabili della query viene associata almeno una tupla di binding.

```
PREFIX o: <http://www.polimi.it/ceri/ope>
ASK
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
    ?o o:haTitolo 'Crepuscolo degli Dei' .
    ?o o:haPrologo ?p .
}
```

Questa query ASK ha valore **true**.

## QUESRY SPARQL – DESCRIBE:

Estrae tutta l'informazione conosciuta relativamente alle risorse che soddisfano una query, in una forma definita dall'endpoint SPARQL.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
DESCRIBE ?o
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
    ?o o:haTitolo 'Valchiria' .
}
```

## SPARQL 1.1:

Essa introduce le clausole GROUP BY e HAVING. Inoltre, introduce l'operatore binario MINUS, che completa la copertura dell'algebra relazionale. Introduce la clausola NOT EXISTS, che ricorda le sottoquery di SQL. E come in SQL, sono disponibili le funzioni aggregate COUNT, SUM, MAX, MIN e AVG, cui si aggiunge la SAMPLE (per estrarre un valore arbitrario da un insieme di valori).

### AGGREGAZIONE IN SPARQL 1.1:

Calcolare la durata totale delle quattro opere che formano il Ring.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT ((SUM ?d) AS ?DurataDelRing)
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
    ?r o:haTitolo 'Ring' .
    ?r o:comprende ?o .
    ?o o:dura ?d .
}
GROUP BY ?r
```

?DurataDelRing

1140

### NEGAZIONE IN SPARQL 1.1:

Estrarre le opere in cui non è presente il personaggio Sigfrido. Due modi alternativi:

```
PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT ?t
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
    ?o o:haPersonaggio ?p .
    ?o o:haTitolo ?t .
} MINUS {
    ?o haPersonaggio 'Sigfrido' .
    ?o haTitolo ?t . }}
```

```
PREFIX o: <http://www.polimi.it/ceri/opera>
```

```
SELECT ?t
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE {
    ?o o:haTitolo ?t .
    FILTER (NOT EXISTS ?o haPersonaggio 'Sigfrido'))
}
```

?t  
'Oro del Reno'  
'Valchiria'

### SOTTOQUERY IN SPARQL 1.1:

Calcolare quanti sono i personaggi che compaiono più di due volte. La soluzione calcola il risultato indicificando due query aggregate.

```
PREFIX o: <http://www.polimi.it/ceri/opera>
SELECT ((sum ?p) AS ?n)
FROM <http://www.polimi.it/ceri/ring.rdf>
WHERE
    { SELECT ?p
    WHERE { ?r o:haTitolo 'Ring' .
            ?r o:comprende ?o .
            ?o o:haPersonaggio ?p . }
    GROUP BY ?p
    HAVING ( COUNT(?o) > 2 ) }
```

?n

3

## SPARQL E INTEROPERABILITÀ:

Tramite istruzioni PREFIX è possibile importare definizioni di standard, vocabolari e ontologie:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbcat: <http://dbpedia.org/resource/Category:>
PREFIX dbpprop: <http://dbpedia.org/property/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX yago: <http://dbpedia.org/class/yago/>
```

## LINKED E OPEN DATA:

I **Linked Data** sono risorse disponibili sul Web, descritte tramite triple RDF e collegate fra loro tramite riferimenti (link). Pubblicare risorse sotto forma di Linked Data vuol dire aderire ad una buona pratica di pubblicazione, usando le IRI e per identificare le risorse, lo standard http per trasferire dati tra diversi nodi del Web, e gli standard RDF, RDFS e OWL per descrivere le triple.

- Risorse interdominio: DBPedia (un dataset periodicamente estratto in modo automatico dai nodi di Wikipedia), Freebase, Yago e OpenCyc.
- Dati di tipo geografico: Geonames (vari milioni di nomi di luoghi) e LinkedGeoData (derivato dal progetto OpenStreetMap).
- Dati relativi ai media: BBC, New York Times e la Reuters.
- Dati relativi all'istruzione e alla ricerca: American Library of Congress, Open Library, DBLP.
- Dati relativi alle scienze della vita: Gene Ontology (che descrive i geni), UniProt (che descrive le proteine), Kegg (l'encyclopedia di Kyoto dei geni e dei genomi) e PubMed (che descrive le pubblicazioni scientifiche relative alla medicina).
- Dati relativi al commercio: GoodRelations (che descrive i vari aspetti del commercio elettronico).

## OPEN DATA:

Sono dati amministrativi resi di dominio pubblico per aumentare la trasparenza delle amministrazioni nei riguardi dei cittadini. Tra i primi a pubblicare open data, gli Stati Uniti ([www.data.gov](http://www.data.gov)) nel 2009, sotto la spunta dell'amministrazione Obama, e la Gran Bretagna ([www.data.gov.uk](http://www.data.gov.uk)).

Il fenomeno della pubblicazione dei dati aperti, talvolta non connessi alla rete dei linked data, è in continua evoluzione, e si citano ad esempio i siti dei comuni di Milano ([www.dati.comune.milano.it](http://www.dati.comune.milano.it)) e di Roma ([www.dati.comune.roma.it](http://www.dati.comune.roma.it)).

## 12. DATABASE NoSQL

**RDBMS** è l'acronimo di *relational database management system* e indica un database management system basato sul modello relazionale. Le motivazioni alla base dell'utilizzo di un RDBMS sono: Schema predefinito per lo storage di dati strutturati, Struttura BCNF già familiare, Strong consistency, Transazioni, Maturi e accuratamente testati, Facile adozione/integrazione, Basati sulle proprietà ACID (atomicity, consistency, isolation, durability), Data retrieval: SQL – versatile e potente, Scalabilità verticale: se volessimo rendere un database SQL scalabile, l'unica alternativa sarebbe quella di potenziare l'hardware sul quale il DBMS è installato.

Cosa ha spinto a database **NoSQL**:

- **BIG USERS**: un gran numero di utenti combinato con pattern di gestione dei dati fortemente dinamici, necessita il bisogno di nuovi database scalabili. Con le tecnologie relazionali, è difficile raggiungere la scalabilità dinamica richiesta dalle applicazioni per garantire il livello di performance richiesto dagli utenti;
- **BIG DATA**: la rapida crescita dell'ammontare dei dati e la natura degli stessi.

Sempre più azienda innovative si affidano ai database NoSQL richiedendo una tecnologia che sia in grado di scalare con i milioni di "cose" (the internet of things) connesse ad Internet. *La chiave è l'accesso globale real-time*. **NoSQL** non è uno specifico linguaggio, ma è il termine che raggruppa un insieme di tecnologie per la persistenza dei dati che funzionano in modo sostanzialmente diverso dai database relazionali, quindi non rispettano una o più caratteristiche dei RDBMS. Possono avere le caratteristiche più disparate: alcuni non utilizzano il modello relazionale, altri usano tabelle e campi ma senza schemi fissi, alcuni non permettono vincoli di integrità referenziale, altri non garantiscono transazioni ACID, oppure ci sono varianti che combinano le precedenti. Alcuni database NoSQL garantiscono solo alcune proprietà ACID. Per esempio, non sempre è garantita la **consistenza** (si parla in questo caso di **eventual consistency**), ossia ci può essere una certa latenza prima che una modifica al database sia visibile. Altri non garantiscono la **durabilità**, ad esempio, in alcuni database distribuiti il malfunzionamento di un nodo dopo una transazione potrebbe impedire la corretta sincronizzazione di tutta la rete. Queste proprietà vengono rilassate per fornire performance migliori.

Caratteristiche dei **NoSQL**:

- **Non relazionali**: i database NoSQL sono **schemaless** e consentono di memorizzare attributi **on the fly**, anche senza averli definiti a priori, questo per consentire la memorizzazione di dati fortemente dinamici;
- **Distribuiti**: la flessibilità nella clusterizzazione e nella replica dei dati permette di distribuire su più nodi lo storage, in modo da realizzare potenti sistemi **fault tolerant**;
- **Scalabili orizzontalmente**: architetture estremamente scalabili che consentono di memorizzare e gestire una grande quantità di informazioni;
- **Open-source**: filosofia alla base del movimento NoSQL;
- Grossi volumi di dati;
- I database NoSQL sono generalmente ideati per richiedere una minore manutenzione rispetto a un sistema RDBMS che, invece, può essere mantenuto solamente con l'assistenza di amministratori esperti e costosi;
- Velocità di risposta alle query;
- Le proprietà ACID non sono richieste;
- CAP theorem.

**TRANSAZIONI BASE:**

I NoSQL si basano sul **modello BASE**:

- **Basically Available**: garantire la disponibilità dei dati anche in presenza di fallimenti multipli. L'obiettivo è raggiunto attraverso un approccio fortemente distribuito;
- **Soft State**: abbandonano il requisito della consistenza dei modelli ACID quasi completamente. La consistenza è un problema dello sviluppatore e non deve essere gestita dal database.
- **Eventually Consistent**: l'unico requisito riguardante la consistenza è garantire che, ad un certo momento, nel futuro, i dati possano convergere ad uno stato consistente.

**BREWER'S CAP THEOREM:**

Un sistema distribuito è in grado di supportare solamente due tra le seguenti caratteristiche:

- **Consistency**: tutti i nodi vedono lo stesso dato nello stesso tempo;
- **Availability**: ogni operazione deve sempre ricevere una risposta;
- **Partition tolerance**: capacità di un sistema di essere tollerante ad una aggiunta o rimozione di un nodo nel sistema distribuito o alla disponibilità di un componente singolo.

**VANTAGGI E SVANTAGGI DI NoSQL:**

Un noto problema di quando si sviluppa con linguaggi orientati ad oggetti è il cosiddetto **O/R impedance mismatch**, ossia i due modelli, relazionale e ad oggetti, sono molto diversi tra loro (può generare problematiche quali il polimorfismo o la conversione dei tipi di dati). Svantaggio più significativo dei database NoSQL è la carenza di tool: essendo una tecnologia recente esistono pochi strumenti di gestione e sviluppo, a contrario del SQL.

### VANTAGGI

- **Strutturazione dei dati**: maggiore centralità alle informazioni e alla loro varietà, supportando l'uso di dati non omogenei pur mantenendo le possibilità di interrogazione, analisi ed elaborazione efficiente;
- **Scalabilità**: i database NoSQL sono generalmente basati su strutture fisiche che si prestano meglio alla distribuzione dei dati su più nodi di una rete (*sharding*), permettendone un'espansibilità maggiore;
- **Prestazioni**: la maggiore distribuzione dei dati sulle reti permette migliori performance;
- **Flessibilità nella progettazione**: la struttura flessibile usata nei database NoSQL non obbliga ad una stereotipazione dei dati durante la progettazione.

### SVANTAGGI

- **Maturità**: i sistemi RDBMS sono in esercizio da tanto tempo;
- **Supporto**: tutte le aziende che sviluppano e vendono RDBMS forniscono un alto livello di supporto alle aziende;
- **Business intelligence**: spesso i tool per la BI non supportano connettività con i database NoSQL;
- **Amministrazione**: al giorno d'oggi sono necessarie grosse competenze per l'installazione e manutenzione dei database NoSQL;
- **Expertise**: è più semplice trovare un esperto amministratore RDBMS che NoSQL.

Conviene utilizzare database NoSQL quando:

- La struttura dati non è definibile a priori;
- I dati disposti nei vari oggetti sono molto collegati tra loro;
- È necessario interagire molto frequentemente con il database;
- Si necessita di prestazioni più elevate.

Non rinunciare al modello relazionale quando:

- DBMS relazionali consolidati da decenni, supportati da una gran varietà di strumenti;
- Si devono modellare dati molto strutturati;
- La duttilità del NoSQL non è un requisito fondamentale.

## CASSIFICARE I DBMS NoSQL:

La rappresentazione dei dati avviene attraverso strutture simili ad oggetti, dette **documenti**, ognuno dei quali possiede delle proprietà che rappresentano le informazioni. I documenti non devono seguire una struttura rigida fissa. Il documento può essere visto come l'equivalente di un record delle tabelle e possono essere messi in relazione tra loro con dei riferimenti.

I DBMS NoSQL possono essere classificati in quattro categorie:

### 1. **Document data store**. Esempio MongoDB:

- Utilizzano dati non strutturati;
- Schema – less;
- Supporto a diversi tipi di documento;
- Ogni documento è identificato da una chiave primaria;
- Scalabilità orizzontale.

### 2. **Key – value data store**. Esempio Redis:

- Utilizza un associative array, chiave – valore, come modello per lo storage;
- Storage, update e ricerca basato sulle chiavi;
- Tipi di dati primitivi familiari ai programmatore;
- Semplice;
- Veloce recupero dei dati;
- Grandi moli di dati.

Vengono costruiti come dizionari o mappe, in cui viene inserita una coppia chiave (identificatore) - valore (informazione). Il loro utilizzo principale è offrire la possibilità di effettuare ricerche rapide di singoli blocchi di informazione. Sono database che puntano tutto sulla velocità.

### 3. **Graph – based data store**. Esempio Neo4j:

- Utilizza nodi (entità), proprietà (attributi) e archi (relazioni);
- Modello logico semplice e intuitivo;
- Ogni elemento contiene un puntatore all'elemento adiacente;
- Attraversamento del grafo per trovare i dati;
- Efficiente per la rappresentazione di reti sociali o dati sparsi;
- Relazioni tra i dati centrali.

Le informazioni possono essere custodite sia nei nodi che negli archi. La forza di questa tipologia è tutto l'insieme del valore informativo che si può estrapolare ricostruendo i percorsi attraverso il grafo.

### 4. **Column – oriented data store**. Esempio Cassandra:

- I dati sono nelle colonne anziché nelle righe;
- Un gruppo di colonne è chiamato famiglia e vi è un'analogia con le tabelle di un database relazionale;
- Le colonne possono essere facilmente distribuite;
- Scalabile;
- Performante;
- Fault – tolerant.

La rapida aggregazione che permettono è stata apprezzata dai grandi produttori di motori di ricerca e Social Network (alcune delle principali soluzioni di questo tipo, Cassandra, Big Table, SimpleDB, sono state realizzate da colossi come Facebook, Google e Amazon).

## MongoDB:

DB NoSQL orientato ai documenti, nato nel 2007 in California come servizio di un progetto più ampio, per poi diventare un prodotto indipendente e open-source. Memorizza i documenti in JSON. I database a documenti seguono un modello che struttura le informazioni in aggregazioni di dati, i **documenti**. I documenti sono disposti in strutture dati lineari dette **collection** o **collezioni**, e possono essere collegati tra loro mediante riferimenti.

Il documento è fondamentalmente un albero che può contenere molti dati, anche annidati. Le collezioni, in cui vengono raggruppati i documenti, possono essere anche eterogenee, non c'è uno schema fisso per i documenti. Tra le collezioni non ci sono relazioni o legami garantiti da MongoDB.

Le caratteristiche chiave di MongoDB sono:

- Consente servizi di alta disponibilità, perché la replicazione di un database (**replica set**) può avvenire in modo molto semplice;
- Garantisce la scalabilità automatica, ossia la possibilità di distribuire (**sharding**) le collezioni in cluster di nodi.

Si adatta a molti contesti, in generale quando si manipolano grandi quantità di dati eterogenei e senza uno schema, non è, invece, opportuno quando si devono gestire molte relazioni tra oggetti.

```
{  
  nome: "Dante",  
  cognome: "Alighieri",  
  nato: 1265,  
  morto: 1321,  
  lingue: ["italiano", "latino" ],  
  opere: [  
    { titolo: "Divina Commedia",  
      iniziata: 1300,  
      lingua: "Italiano",  
      tipo: "poesia",  
      versi: "endecasillabi",  
      libri: ["Inferno", "Purgatorio", "Paradiso"] }  
  ]  
}
```

## PROGETTAZIONE:

1. Individuazione delle possibili entità;
2. Per quanto possibile, selezionare un insieme di proprietà che dovrebbero apparire nei documenti. Queste attività non sono obbligatorie, in quanto molti MongoDB (e altri database NoSQL), non richiedono una definizione apriori della struttura interna dei documenti. Su MongoDB i documenti sono rappresentati in BSON (formato binario derivato e molto simile a JSON). Ogni documento è identificato da un valore univoco, *ObjectID*;
3. Individuazione dei documenti da innestare in altri: innestando documenti in altri, si ha da un lato il vantaggio che abbiamo in un unico documento tutte le informazioni relative ad una entità, d'altro canto, però, potrebbe comportare una ridondanza dei dati.

MongoDB è fortemente orientato all'interazione con le applicazioni e non contempla alcune funzionalità come il controllo sui valori inseriti.

## PROGETTAZIONE DATABASE KEY-VALUE:

Sono ispirati alla struttura dati di dizionario. Tutto ciò che viene archiviato in questi database dovrà essere fornito sotto forma di coppia chiave/valore, dove il valore è l'informazione vera e propria, e la chiave è ciò che ne consente il recupero. Un database key-value può essere considerato come un'unica grande mappa (o dizionario). Il DBMS di questo tipo più diffuso è Redis. I database di questo tipo sono stati spesso criticati perché troppo di nicchia. Vengono usati per operazioni di cache o immagazzinamento di dati di sessione.

Ma si prestano anche ad altri scenari applicativi, ad esempio ogni tabella di un database relazionale vive di una struttura key – value dove la "key" è costituita dalla chiave primaria e il "value" è rappresentato dagli altri campi della stessa riga. In un database key – value, la chiave è l'elemento centrale della memorizzazione, è necessario scegliere correttamente le chiavi. Le chiavi in Redis sono *binary safe*, pertanto possono essere delle sequenze alfanumeriche o il contenuto di un file non testuale, è importante, però, non utilizzare chiavi di dimensioni eccessive per non penalizzare le prestazioni in fase di ricerca. Si consiglia di definire chiavi leggibili, buona norma prevede di anteporre un prefisso seguito da un separatore (spesso si usa ":").

id	nome	cognome	indirizzo	città	CAP
123456	Giulio	Rossi	Via Monte Bianco 78	Torino	10100

Chiave	Valore
Users:123456	HashMap { "nome": "Giulio" "cognome": "Rossi" "indirizzo": "Via Monte Bianco 78" "città": "Torino" "cap": "10100" }

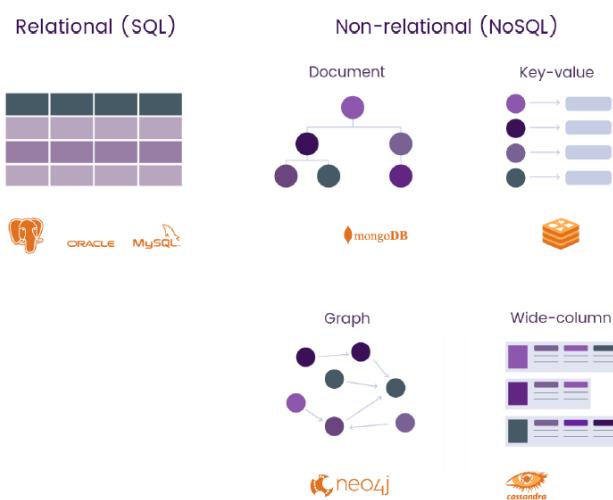
## DATABASE A GRAFO:

Questi si differenziano dai RDBMS per il modello di dati. Neo4j (2003) è il database a grafo più utilizzato, seguito da OrientDB. Caratteristiche principali: modello di dati a grafo orientato con proprietà chiave – valore, supportate sia sui nodi che sulle relazioni (modello detto Property Graph); transazioni ACID; linguaggio di query dichiarativo denominato Cypher; driver per interagire con i linguaggi di programmazione più diffusi.

Modello Property Graph: i nodi e le relazioni che possono essere memorizzati in Neo4j possono avere un elenco di proprietà chiave – valore, solitamente la chiave è una stringa che rappresenta il nome della proprietà, mentre il valore può essere un numero (intero o reale), una stringa, un array di numeri o stringhe. Il modello prevede che i **nodi** abbiano un **ID** univoco assegnato automaticamente da Neo4j al momento della creazione, possano avere una o più **Label** per classificarli e indicizzarli. I nodi sono rappresentati da parentesi tonde, le proprietà si scrivono con la stessa sintassi di JSON, cioè un elenco di attributi **chiave:valore** separati da "," e racchiusi in {}. Ogni **relazione** ha necessariamente un **tipo**, specificato dall'utente in fase di creazione, ha necessariamente una **direzione** (cioè una relazione va da un nodo a un altro) e non è possibile creare relazioni senza verso.

## OrientDB:

Database multi – model, modello ibrido con funzionalità NoSQL, prodotto e sviluppato da Orient Technologies. Supporta: **Object Model** (permette di definire classi di dati, supporta ereditarietà e polimorfismo), **Document Model** (permette di raccogliere i documenti in collezioni legate tra loro da link), **Graph Model** (raccoglie documenti in nodi collegati tra loro da relazioni), **Key – value Model** (immagazzinare i dati in strutture indicizzate tramite chiavi). OrientDB è realmente Multi – Model, nel senso che i vari approcci sono veri modelli di gestione dei dati di cui il motore del DBMS è stato dotato. La varietà di usi cui si presta è uno dei fattori determinanti della sua diffusione. Tra i vari paradigmi NoSQL non ne esiste uno migliore, ma a seconda delle circostanze si possono sfruttare i vantaggi di un approccio piuttosto che di altri.



## 13. DATA WAREHOUSE

La maggior parte delle aziende dispone di enormi basi di dati contenenti dati di tipo operativo per questo c'è bisogno di **sistemi per il supporto alle decisioni** che permettano di analizzare lo stato dell'azienda e prendere decisioni rapide e migliori.

### OLTP (ON LINE TRANSACTION PROCESSING) E OLAP (ON LINE ANALYTICAL PROCESSING):

La tecnologia delle basi di dati è finalizzata prevalentemente alla **gestione di dati "in linea"** (**OLTP**). Con questa tecnologia, le imprese accumulano grandi moli di dati relativi alla gestione operativa, per esempio, le catene di supermercati dispongono dei dati relativi alle vendite dei loro prodotti. Ma questi dati possono rilevarsi utili non solo per la gestione dell'impresa, ma anche per la **pianificazione** e il **supporto alle decisioni**, per esempio, è possibile osservare le variazioni nelle vendite di prodotti di un supermercato in funzione delle varie promozioni per stabilire l'efficacia. In pratica, i dati del presente e del passato possono contenere un'attività di analisi essenziale nella pianificazione e programmazione delle attività future dell'impresa. A fianco dei sistemi OLTP si sono sviluppati sistemi dedicati esclusivamente all'elaborazione e analisi dei dati, ovvero **OLAP (On Line Analytical Processing)**. Questa analisi dei dati avviene attraverso strumenti interattivi che forniscono risposte in tempi brevi, anche a fronte di grandi moli di dati.

OLAP è la principale modalità di fruizione delle informazioni contenute in un DW. Consente agli utenti di esplorare interattivamente i dati sulla base del **modello multidimensionale**. In più, mentre le applicazioni OLTP sono condivise a tutti gli utenti finali, le applicazioni OLAP sono utilizzate da utenti che si occupano esclusivamente dello svolgimento di attività decisionali aziendali, come gli analisti.

	DB operazionali	DW
Utenti	Migliaia	Centinaia
Carico di lavoro	Transazioni predefinite	Interrogazioni di analisi ad hoc
Accesso	Centinaia di record in lettura e scrittura	Milioni di record per lo più in lettura
Scopo	Dipende dall'applicazione	Supporto alle decisioni
Dati	Elementari	Di sintesi
Integrazione dei dati	Per applicazione	Per soggetto
Qualità	In termini di integrità	In termini di consistenza
Copertura temporale	Solo dati correnti	Dati correnti e storici
Aggiornamenti	Continui	Periodici
Modello	Normalizzato	Denormalizzato, Multidimensionale
Ottimizzazione	Per accessi OLTP su una frazione del DB	Per accessi OLAP su gran parte del DB
Sviluppo	A cascata	Iterativo

### BUSINESS INTELLIGENCE:

È una disciplina di supporto alla decisione strategica aziendale il cui obiettivo è quello di trasformare i dati aziendali in informazioni fruibili a diversi livelli di dettaglio e per applicazioni di analisi. Necessita un'adeguata infrastruttura hardware e software di supporto.

### KNOWLEDGE DISCOVERY:

È l'attività che consiste nell'individuare ed estrarre da enormi volumi di dati tutta la conoscenza utile alle attività più strategiche.

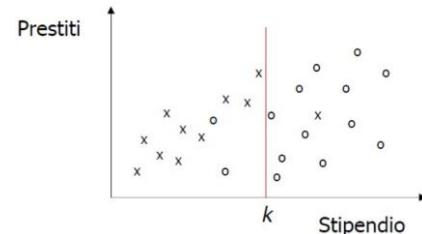
- **Dati**: insieme di informazioni contenute in una base di dati o data warehouse;
- **Pattern**: espressione in un linguaggio opportuno che descrive in modo succinto le informazioni estratte dai dati, ovvero informazione di alto livello.

### Esempio:

Clienti di una banca che hanno contratto un prestito:

- **x**: clienti che hanno mancato la restituzione di rate;
- **o**: clienti che hanno rispettato le scadenze.

If stipendio < k € then mancati pagamenti.



### CARATTERISTICHE DEI PATTERN:

- **Validità**: i pattern scoperti devono essere validi su nuovi dati con un certo grado di certezza. Ad esempio, lo spostamento a destra del valore di  $k$  porta a una riduzione del grado di certezza;
- **Novità**: misurata rispetto a variazione dei dati o della conoscenza estratta;
- **Utilità**: ad esempio un aumento di profitto atteso dalla banca associato alla regola estratta;
- **Comprensibilità**: misure di tipo sintattico (numero di bit del pattern) e semantico.

### DATA WAREHOUSE:

A partire dagli anni 2000, si è imposta un'architettura per l'analisi dei dati basata sulla costruzione di una base di dati chiamata **data warehouse** a supporto delle decisioni, nella quale vengono raccolte tutte le informazioni di ausilio all'analisi. In questa architettura i sistemi OLTP svolgono il ruolo di **"sorgenti di dati"**, che alimentano l'ambiente OLAP a seguito di opportune operazioni di interrogazione, caricamento e pulizia dei dati.

### DIFFERENZA TRA DATA WAREHOUSE E DATA MART:

- **Data Warehouse aziendale**: contiene informazioni sul funzionamento di tutta l'azienda.
- **Data mart**: sottosistemi logici del DW in grado di soddisfare specifiche esigenze di analisi, ad esempio, quelle relative ad un particolare settore dell'azienda. Ha una realizzazione più rapida ma richiede una progettazione attenta per evitare problemi di integrazione in seguito.

### CARATTERISTICHE ARCHITETTURALI DATA WAREHOUSE:

Un **data warehouse** è una base di dati per il supporto alle decisioni, che è mantenuta separata dalle basi di dati operative, e presenta **caratteristiche** che la distinguono rispetto a una base di dati dedicata allo svolgimento di operazioni OLTP:

- È una **base di dati integrata**, i dati di interesse provengono da diverse sorgenti informative preesistenti, richiede la riconciliazione delle eterogeneità;
- Contiene **informazioni di carattere storico/temporale**, le basi di dati operazionali mantengono il valore corrente delle informazioni, mentre nel DW è di interesse l'evoluzione storica delle informazioni;
- Contiene **dati in forma aggregata**, le attività di analisi si basano su informazioni ottenute aggregando dati in base ad un fattore comune;
- Ha un'esistenza autonoma, il DW viene mantenuto fisicamente separato dalle sorgenti informative;
- È una **base di dati fuori linea**, i meccanismi di importazione dei dati sono di tipo asincrono e periodico, in modo da non penalizzare le prestazioni delle sorgenti di dati. In tal caso, il DW non contiene dati perfettamente aggiornati rispetto al flusso di transazioni, ma questo è ritenuto accettabile.

Un'architettura adatta a soddisfare questi requisiti comprende i seguenti **componenti**:

- **Sorgenti dei dati (data source)**: i dati vengono estratti da sistemi preesistenti, includendo una vasta tipologia di sistemi, come i DBMS o legacy;
- **Data warehouse server**: sistema dedicato alla gestione del DW e può basarsi su diverse tecnologie (ROLAP o MOLAP), tale componente memorizza i dati mediante opportune strutture fisiche e realizza interrogazioni complesse (join tra tabelle, ordinamenti e aggregazioni), consente anche speciali operazioni (roll-up, drill-down, ecc...);
- **Sistema di alimentazione**: Esso consiste in una serie di strumenti detti **ETL** che svolgono determinate operazioni di base.
- **Alcuni strumenti di analisi**: consentono di effettuare analisi dei dati usufruendo dei servizi del DW server.

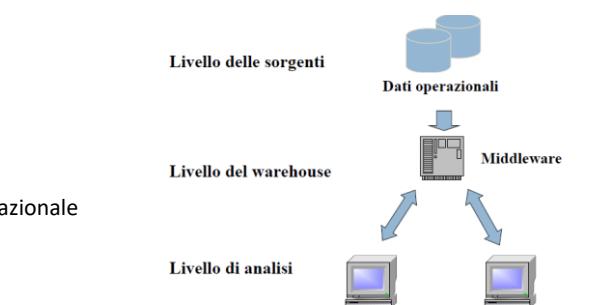
#### ARCHITETTURA DW AD UN LIVELLO:

Caratteristiche:

- DW virtuale
- Minimizzazione dei dati memorizzati

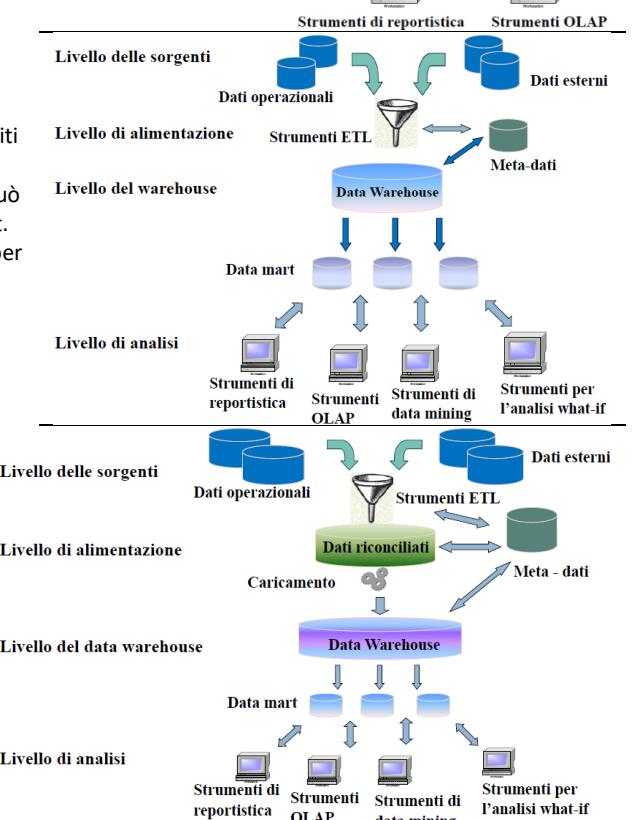
Punti deboli:

- Non rispetta il requisito di separazione tra l'elaborazione analitica OLAP e quella transazionale OLTP.



#### ARCHITETTURA DW A DUE LIVELLI:

- **Livello delle sorgenti**: Fonti di dati eterogenei estratti dall'ambiente di produzione oppure provenienti da sistemi informativi esterni all'azienda.
- **Livello dell'alimentazione**: i dati memorizzati nelle sorgenti vengono estratti e ripuliti tramite strumenti ETL (Extraction, Transformation and Loading)
- **Livello del warehouse**: le informazioni vengono raccolte in un DW centralizzato e può essere consultato direttamente o utilizzato come sorgente per costruire i Data mart.
- **Livello di analisi**: Permette la consultazione efficiente e flessibile dei dati integrati per fini di stesura di report, di analisi e di simulazione.



#### ARCHITETTURA DW A TRE LIVELLI:

Viene introdotto il livello dei dati riconciliati, ovvero il livello che materializza i dati operazionali ottenuti a valle del processo di integrazione e ripulitura dei dati sorgente.

#### STRUMENTI ETL (EXTRACT, TRANSFORM E LOAD):

Il ruolo degli **strumenti ETL** è quello di alimentare una sorgente di dati che possa a sua volta alimentare il DW. Le operazioni di questi strumenti vengono definite con il termine di **riconciliazione**. Svolgono il proprio compito in 4 fasi:

1. Estrazione dei dati da sorgenti esterne;
2. Pulizia dei dati (errori, dati mancanti o duplicati);
3. Trasformazioni e conversioni di formato;
4. Caricamento e refresh periodico;

La riconciliazione avviene periodicamente e quando il DW viene popolato la prima volta.

#### ETL - ESTRAZIONE:

Consiste nell'estrazione di dati rilevanti dalle sorgenti. Tipi di estrazione:

- **Estrazione statica**: effettuata quando il DW viene popolato per la prima volta;
- **Estrazione incrementale**: viene usata per l'aggiornamento periodico nel DW;
- **Guidata dalle sorgenti**: consiste nel riscrivere le applicazioni operazionali per far sì che esse notifichino in modo asincrono le modifiche.

#### ETL - TRASFORMAZIONE:

Converte i dati dal formato operazionale sorgente a quello del DW. Le sue **funzionalità** sono operare sia a livello di formato di memorizzazione sia a livello di unità di misura al fine di uniformare i dati (**conversione e normalizzazione**), stabilire corrispondenze tra campi equivalenti in sorgenti diverse (**Matching**) e ridurre il numero di campi e di record rispetto alle sorgenti (**Selezione**).

#### ETL - PULITURA:

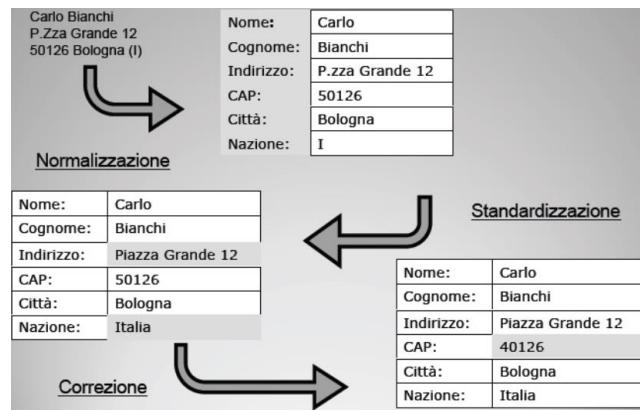
Si occupa di migliorare la qualità dei dati prima di inserirli nel DW, normalmente scarsa nelle sorgenti. Le sue **funzionalità** utilizzano dizionari appositi per correggere gli errori di scrittura (**correzione ed omogeneizzazione**) e applica regole del dominio applicativo per stabilire le corrette corrispondenze tra valori (**pulitura basata su regole**). Alcune tipologie di errori: dati duplicati, mancanti, inconsistenza, valori impossibili.

#### ETL - CARICAMENTO:

Il caricamento dei dati avviene secondo due modalità:

- **Refresh**: i dati nel DW vengono riscritti integralmente, usata in abbinamento all'estrazione statica;
- **Update**: nel DW vengono aggiunti solo i cambiamenti occorsi ai dati sorgente, usata in abbinamento all'estrazione incrementare.

## Esempio pulitura e trasformazione di un dato anagrafico:



## **MODELLO MULTIDIMENSIONALE (STRUTTURA ED ELABORAZIONE DEI DATI):**

I dati presenti in una data warehouse vengono presentati all'utente finale mediante una rappresentazione di alto livello che prescinde dai criteri di memorizzazione dei dati e ne favorisce l'analisi. Questo è il **modello multidimensionale** utilizzato per la rappresentazione e l'interrogazione dei dati nei DW. Tale modello è basato su tre concetti di base:

- Un **fatto**, concetto del sistema informativo aziendale (o della relativa realtà di interesse) sul quale ha senso svolgere un processo di analisi;
- Una **misura** (ogni cella), proprietà atomica di un fatto, ovvero attributo numerico o conteggio delle istanze;
- Una **dimensione** (ogni asse), particolare prospettiva lungo la quale l'analisi di un fatto può essere effettuata. I valori possibili per una dimensione vengono generalmente detti **membri**.

Le dimensioni vengono tipicamente organizzate in **gerarchie** di **livelli di aggregazione**. Ogni dimensione può essere la radice di una **gerarchia** di attributi usati per aggregare i dati memorizzati nei cubi di base.

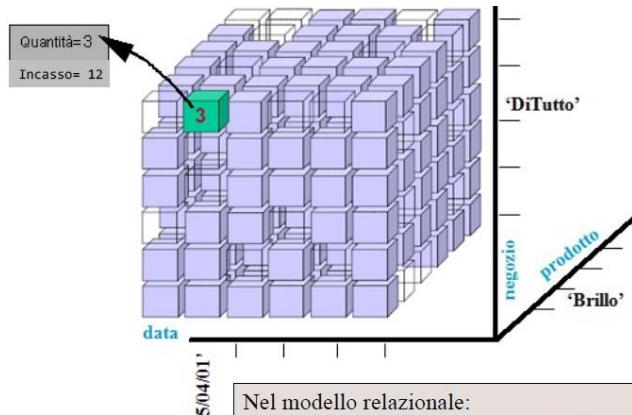
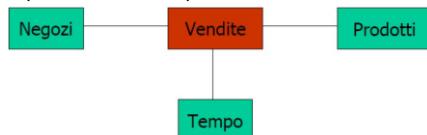
Una volta fissate le dimensioni di un fatto e un livello di aggregazione di interesse per ogni dimensione, un'**istanza** del fatto assegna a ogni combinazione valida di membri dei livelli selezionati un valore per ciascuna misura del fatto.

Esiste una naturale rappresentazione grafica nella quale le istanze di un fatto sono rappresentate da **cubi** multidimensionali costituiti da elementi atomici detti **celle**. In questi cubi, ogni dimensione "fisica", corrisponde ad una dimensione "concettuale" del fatto a un certo livello di aggregazione e le celle del cubo contengono le istanze del fatto. Fissando uno specifico membro per ogni dimensione si determina la coordinata di una particolare cella del cubo. In questa cella viene riportata l'istanza relativa, ovvero la misura del fatto associata ai membri dati.

### Esempio:

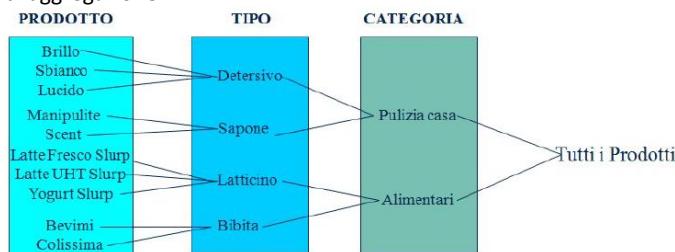
In un'azienda commerciale di vendite che dispone di una catena di supermercati un **fatto** può essere il concetto di *Vendita* e possibili **misure** per questo fatto possono essere la *quantità venduta* di un prodotto in un certo periodo di tempo e l'*incasso* relativo. Possibili **dimensioni** di analisi per una vendita possono essere l'*articolo venduto*, il *periodo di tempo* nel quale la vendita è stata effettuata e il *luogo* in cui si è svolta. Possibili **membri** per il livello città della dimensione *Luogo* possono essere tutti i *comuni italiani* nei quali esistono negozi dei quali vogliamo analizzare le vendite. Una possibile **istanza** di un fatto Vendite sulle dimensioni potrebbe associare il valore 2578 della misura *quantità vendita* ai membri: *Roma* del livello città della dimensione *Luogo*, *televisori* del livello prodotto della dimensione *Articolo* e *maggio 2014* del livello mese della dimensione *Tempo*.

Data warehouse per l'analisi delle vendite di una catena di supermercati (*modello a stella*):



Il cubo a tre dimensioni delle vendite -->

A ciascuna dimensione del cubo è associata una gerarchia di livelli di aggregazione:



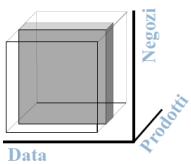
Nel modello relazionale:  
**VENDITE(negozio, prodotto, data, quantità, incasso)**  
con una dipendenza funzionale:  
**negozio, prodotto, data → quantità, incasso**

## **RESTRIZIONE:**

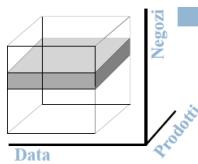
Restringere i dati significa ritagliare una porzione dal cubo circoscrivendo il campo di analisi. La forma più semplice di restrizione è lo **slicing**. Consiste nel ridurre la dimensionalità del cubo fissando un valore per uno o più dimensioni.

### Esempio di restrizione:

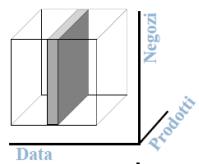
Al manager di prodotto interessa la vendita di **un prodotto** in tutti i periodi e in tutti i negozi:



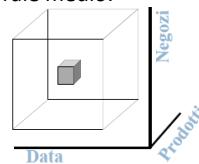
Al manager regionale interessa la vendita dei prodotti in tutti i periodi nei **propri negozi**:



Al manager finanziario interessa la vendita dei prodotti in tutti i negozi relativamente ad **un determinato periodo**:



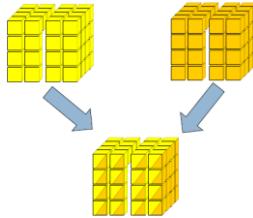
Il manager strategico si concentra su una categoria di prodotti, un'area regionale ed un orizzonte temporale medio:





### OPERATORI OLAP DI AGGREGAZIONE - DRILL-ACROSS:

Il Drill-Across consiste nello stabilire un confronto tra due o più cubi correlati in modo da ottenere una visualizzazione comparata di due diverse misure e per il calcolo di misure derivate dai dati presenti sui cubi. Per fare ciò occorre che i metadati presenti nel DW siano coordinati.



### OPERATORI OLAP DI AGGREGAZIONE - DRILL-THROUGH:

Consiste nel passaggio dai dati aggregati multi dimensionalmente del DW ai dati operazionali presenti nelle sorgenti o nel livello riconciliato.

### REALIZZAZIONE DI UN DATA WAREHOUSE:

Per realizzare una DW esistono due soluzioni alternative:

- **ROLAP: Relational OLAP**, consiste nell'uso della tecnologia relazionale, i dati vengono memorizzati tramite tabelle e le operazioni di analisi vengono tradotte in opportune istruzioni SQL. Viene utilizzato su DBMS relazionali, sono necessarie tipologie specifiche di schemi per traslare il modello multidimensionale su attributi, relazioni e vincoli di integrità. Presenta ridondanza e ha **basse prestazioni** dovute a operazioni di JOIN.
- **MOLAP: Multidimensional OLAP**, memorizza i dati direttamente in forma multidimensionale, tramite speciali strutture dati tipicamente proprietarie. Viene utilizzato su DBMS multidimensionali, è un modello ad hoc ed accesso di tipo posizionale, le operazioni multidimensionali sono realizzabili in modo semplice senza ricorrere a JOIN questo porta ad **ottime prestazioni**.

Quando si realizza l'implementazione di un DW bisogna considerare anche:

- **Qualità**: la qualità di un processo misura la sua aderenza agli obiettivi degli utenti. I fattori che caratterizzano la qualità dei dati sono Accuratezza: conformità tra il valore memorizzato e quello reale, Attualità: il dato memorizzato non è obsoleto, Completezza: non mancano informazioni, Consistenza: la rappresentazione dei dati è uniforme, Disponibilità: i dati sono facilmente disponibili all'utente, Tracciabilità: è possibile risalire alla fonte di ciascun dato, Chiarezza: i dati sono facilmente interpretabili.
- **Sicurezza**: controllo delle autorizzazioni e sulla mole di dati trasferiti dalle sorgenti
- **Evoluzione**: l'evoluzione delle informazioni nel DW nel tempo a livello dei dati, aggiunta di nuove categorie di dati e possibilità di cambiare la categoria di un dato, e a livello di schema, se mutano i requisiti dell'utente e se variano le sorgenti di dati.

### CICLO DI VITA DEI SISTEMI DI DATA WAREHOUSE:

All'interno di un DW sono diversi i **fattori di rischio**:

- Legati alla gestione del progetto: scarsa disponibilità a condividere informazioni tra reparti;
- Legati alle tecnologie: rapida evoluzione delle tecnologie, mancanza di standard;
- Legati ai dati ed alla progettazione: risultati di scarso valore, causati da sorgenti instabili ed inaffidabili, specifica inaccurata dei requisiti;
- Legati all'organizzazione: incapacità di coinvolgere attivamente l'utente finale nel progetto.

Esistono diversi approcci per diminuire questi fattori di rischio:

- **Approccio Top-Down**: con questo tipo di approccio è necessario analizzare i bisogni globali dell'intera azienda, pianificare lo sviluppo del DW e progettarlo e realizzarlo nella sua interezza.
  - **Vantaggi**: visione globale dell'obiettivo, DW consistente e ben integrato.
  - **Svantaggi**: Tempi lunghi di realizzazione, complessità nell'analisi e nella riconciliazione di tutte le fonti, impossibilità di prevedere le esigenze particolari delle diverse aree aziendali
- **Approccio Bottom-Up**: con questo tipo di approccio il DW è costruito in modo incrementale, i Data mart sono concentrati su una specifica area di interesse, il primo Data mart deve essere quello più strategico per l'azienda e deve ricoprire un ruolo centrale per l'intero DW.  
Questo approccio ha un particolare ciclo di vita:
  - **Definizione degli obiettivi e pianificazione**: individuazione obiettivi e confini del sistema, stima delle dimensioni, valutazione dei costi e del valore aggiunto, scelta dell'approccio per la costruzione, analisi dei rischi e delle aspettative;
  - **Progettazione dell'infrastruttura**: scelte architettoniche e degli strumenti;
  - **Progettazione e sviluppo dei Data mart**.

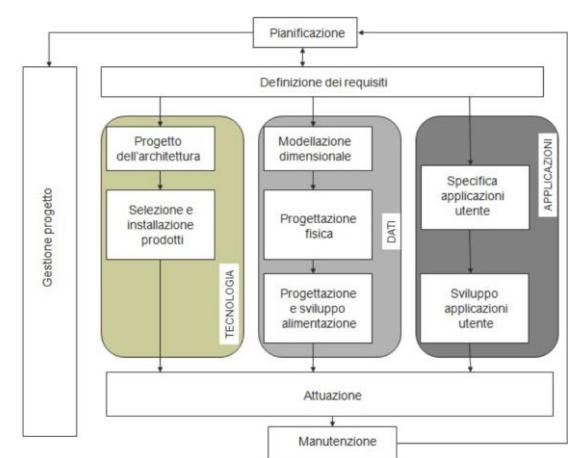
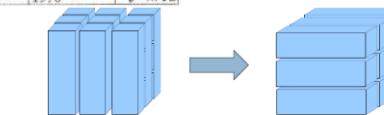
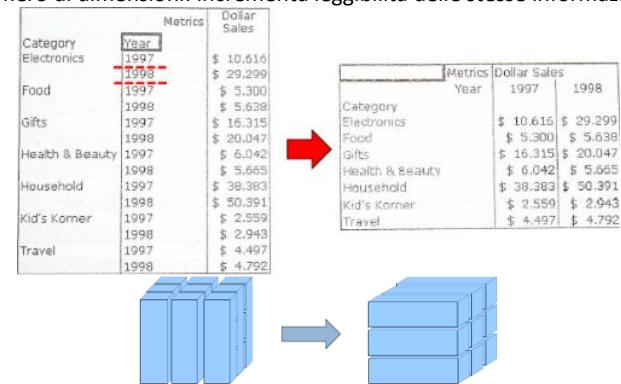
### BUSINESS DIMENSIONAL LIFECYCLE (BDL):

È il ciclo di vita per la progettazione, lo sviluppo e l'attuazione di DW.

- **Pianificazione**: scopi e confini del sistema, valutazione impatti organizzativi, stima costi e benefici, allocazione delle risorse;
- **Definizione dei requisiti**: massima utilità e redditività del sistema, catturare i fattori chiave e trasformarli in specifiche (diviso in fase dei dati, tecnologia e applicazioni);
- **Attuazione**: comporta l'effettivo avviamento del sistema sviluppato;
- **Manutenzione**: assicura il supporto e la formazione degli utenti;
- **Gestione del progetto**: occupa tutte le fasi del ciclo di vita, permette di mantenere le diverse attività sincronizzate.

### OPERATORI OLAP DI PIVOTING:

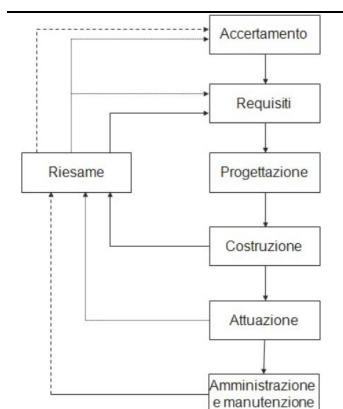
L'operatore di pivoting consiste nel ruotare gli assi di visualizzazione del cubo dei fatti mantenendo invariato il livello di aggregazione ed il numero di dimensioni: incrementa leggibilità delle stesse informazioni.



## RAPID WAREHOUSING METHODOLOGY (RVM):

È un'altra metodologia di sviluppo di DW. È una metodologia iterativa ed evolutiva che suddivide grossi progetti in sotto progetti meno rischiosi chiamati **build**. Ogni build riprende l'ambiente di quello precedente, estendendolo con nuove funzionalità.

- **Accertamento:** fattibilità del progetto da parte dell'azienda, scopi, rischi e benefici;
- **Requisiti:** specifiche di analisi, del progetto e di architettura;
- **Progettazione:** progetto logico e fisico dei dati, dell'alimentazione e selezione degli strumenti d'implementazione;
- **Costruzione:** implementazione e popolazione del DW, sviluppo e collaudo front-end;
- **Attuazione:** il sistema viene consegnato e avviato, gli utenti vengono addestrati;
- **Amministrazione e manutenzione:** presente durante tutta la vita del sistema, estensione delle funzionalità, ridimensionamento dell'architettura;
- **Riesame:** verifica dell'implementazione, accertamento che il sistema sia ben accettato dall'organizzazione.

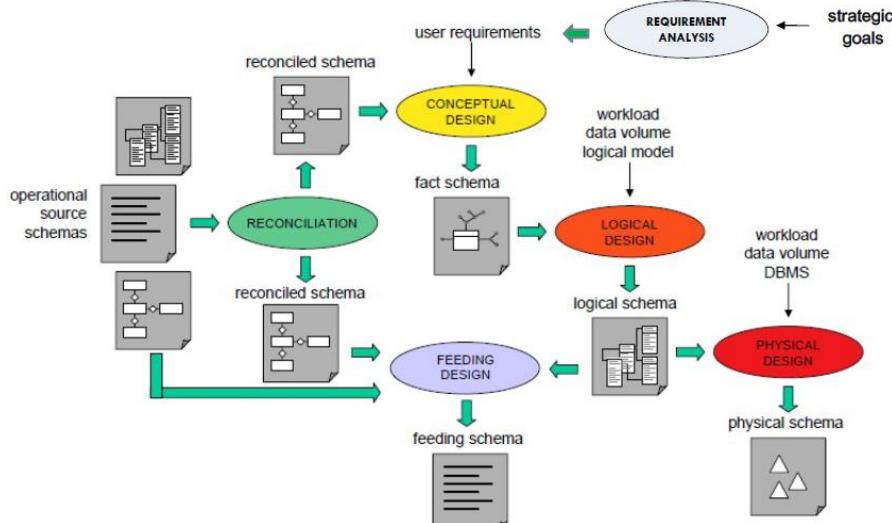


## PROGETTAZIONE DI UN DATA MART:

- **Analisi e riconciliazione delle fonti dati:** analizzare e comprendere gli scemi delle sorgenti, normalizzazione, valutare la qualità dei dati;
- **Analisi dei requisiti:** raccolta, filtro e documentazione dei requisiti, scelta dei fatti e granularità dei fatti (compromesso tra velocità e dettaglio);
- **Progettazione concettuale:** può essere usato il DFM (Dimensional Fact Model), creazione degli schemi di fatto;
- **Raffinamento del carico di lavoro e validazione dello schema concettuale:** formulazione delle interrogazioni direttamente sullo schema concettuale, verifica che le interrogazioni siano effettivamente esprimibili;
- **Progettazione logica:** scelta dell'implementazione (ROLAP o MOLAP), schemi logici, materializzazione delle viste;
- **Progettazione dell'alimentazione:** decisioni riguardanti il processo di alimentazione del livello riconciliato e del Data mart;
- **Progettazione fisica:** scelta degli indici per ottimizzare le prestazioni, riferimento ad un particolare DBMS, carico di lavoro e volume dei dati.

Fase	Ingresso	Uscita	Figure Coinvolte
Analisi e riconciliazione delle fonti	Schemi delle sorgenti	Schema riconciliato	Progettista; amministratore db operazionale;
Analisi dei requisiti	Obiettivi strategici	Specifiche dei requisiti; carico di lavoro preliminare	Progettista; utenti finali
Progettazione concettuale	Schema riconciliato; specifica dei requisiti	Schemi di fatto	Progettista; utenti finali
Raffinamento carico di lavoro, validazione schema concettuale	Schemi di fatto; carico di lavoro preliminare	Carico di lavoro; schemi di fatto validati	Progettista; utenti finali
Progettazione Logica	Schemi di fatto; modello logico target; carico di lavoro	Schema logico del Data mart	Progettista
Progettazione dell'alimentazione	Schemi delle sorgenti; schema riconciliato; schema logico del Data mart.	Procedure di alimentazione	Progettista; amministratori db operazionale
Progettazione fisica	Schema logico del Data mart; DBMS target; carico di lavoro	Schema fisico del Data mart	Progettista

## LE SETTE FASI DELLA PROGETTAZIONE:



## ANALISI E RICONCILIAZIONE DELLE FONTI DATI:

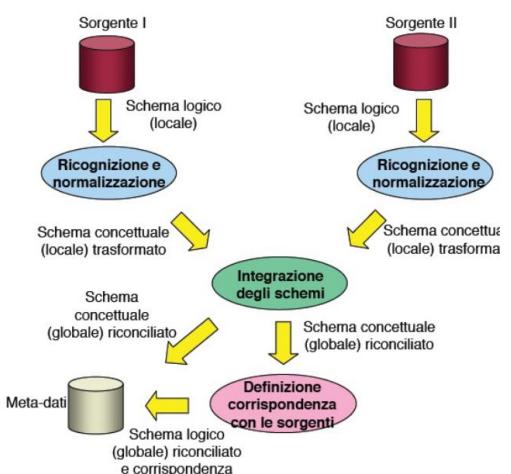
Questa fase richiede di definire e documentare lo schema del livello dei dati operazionali, a partire da quale verrà alimentato il Data mart. Riceve in ingresso gli schemi delle sorgenti e produce un insieme di metadati che modellano lo schema riconciliato e le corrispondenze tra gli elementi di quest'ultimo e quelli del sistema operazionale.

Le **figure coinvolte** sono: il progettista e l'amministratore dei database operazionali in quanto la sua conoscenza del dominio applicativo è indispensabile per normalizzare gli schemi.

A questo livello il modello architettonurale di riferimento è quello **a tre livelli**, nel quale si suppone che il livello di dati riconciliato esiste, la scelta ricade su questo modello perché l'alimentazione diretta del DW è un compito troppo complesso per essere eseguito in modo atomico.

La figura dettaglia la fase di analisi e riconciliazione in caso di più sorgenti, di cui è noto il solo schema logico:

- **Riconoscione e normalizzazione** dei diversi schemi locali che produce un insieme di schemi concettuali, localmente consistenti e completi (va svolta anche in presenza di una sola sorgente dati, con più sorgenti viene ripetuta per ogni singolo schema locale)
- **Integrazione** che produce uno schema concettuale globalmente consistente
- Dallo schema ottenuto si effettua la progettazione logica dello schema riconciliato, per poi **definirne la corrispondenza** con schemi logici delle sorgenti.

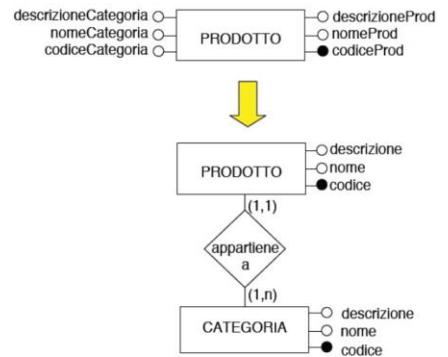


## RICONOSCIMENTO E NORMALIZZAZIONE:

Prima di procedere alla fase di progettazione concettuale il progettista deve acquisire conoscenza delle sorgenti operazionali attraverso attività di:

- **Riconoscimento:** esame approfondito degli schemi locali mirato alla piena comprensione del dominio applicativo;
- **Normalizzazione:** mira a correggere gli schemi locali al fine di modellare in modo più accurato il dominio applicativo;

Il progettista deve verificare la completezza degli schemi locali sforzandosi di individuare eventuali correlazioni involontariamente omesse. Le trasformazioni apportate allo schema non devono introdurre nuovi concetti, bensì rendere esplicativi tutti quelli ricavabili, oltre alle trasformazioni necessarie il progettista deve anche individuare eventuali porzioni degli schemi locali non utili al Data mart.



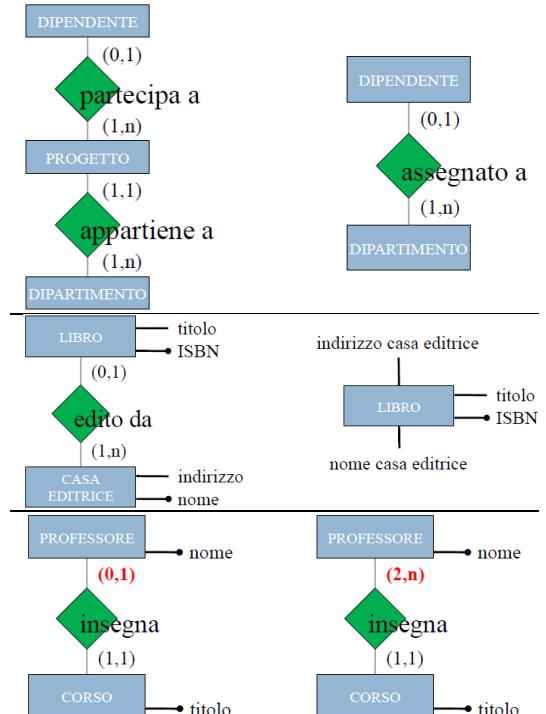
## INTEGRAZIONE:

Consiste nella individuazione di corrispondenze tra i concetti degli schemi locali e nella risoluzione dei conflitti evidenziati. Lo scopo è di creare un unico schema globale i cui elementi sono correlati con i corrispondenti elementi degli schemi locali (**mapping**). In questa fase vanno anche identificati concetti distinti di schemi differenti che sono correlati attraverso proprietà semantiche (**proprietà inter-schema**). Per poter ragionare sui concetti degli schemi sorgente è necessario usare un unico formalismo (ER, UML, DTD ecc.) in modo da fissare i costrutti utilizzabili e la potenza espressiva.

Sono 5 i **problem**i individuabili durante l'integrazione:

1. **Diversità di prospettiva:** il punto di vista rispetto al quale diversi gruppi di utenti vedono uno stesso oggetto del dominio applicativo può differenziarsi in base agli aspetti per la funzione a cui essi sono preposti.
2. **Equivalenza dei costrutti del modello:** i formalismi di modellazione permettono di rappresentare uno stesso concetto utilizzando combinazioni diverse dei costrutti a disposizione.
3. **Incompatibilità delle specifiche:** schemi diversi che modellano una stessa porzione del dominio applicativo racchiudono concetti diversi, in contrasto tra loro. Tali diversità possono coinvolgere ad esempio la scelta dei nomi, dei tipi di dati e dei vincoli di integrità.
4. **Concetti comuni:** quattro sono le possibili relazioni esistenti tra due distinte rappresentazioni R1 e R2 di uno stesso concetto:
  - **Identità:** si verifica quando vengono utilizzati gli stessi costrutti, il concetto è modellato dallo stesso punto di vista, quindi R1 e R2 coincidono.
  - **Equivalenza:** si verifica quando R1 e R2 non sono le stesse poiché sono stati utilizzati costrutti diversi ma equivalenti (sono equivalenti se le loro istanze possono essere messe in corrispondenza 1 a 1).
5. **Concetti correlati:** a seguito dell'integrazione, molti concetti diversi, ma correlati, verranno a trovarsi nello stesso schema dando vita a nuove relazioni che non erano percepibili in precedenza. Tali relazioni sono dette **proprietà inter-schema**.

Esempio: in un caso un professore non può tenere più di un corso, nell'altro deve tenerne almeno 2.

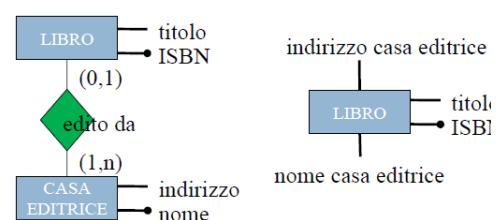


4. **Concetti comuni:** quattro sono le possibili relazioni esistenti tra due distinte rappresentazioni R1 e R2 di uno stesso concetto:
  - **Identità:** si verifica quando vengono utilizzati gli stessi costrutti, il concetto è modellato dallo stesso punto di vista, quindi R1 e R2 coincidono.
  - **Equivalenza:** si verifica quando R1 e R2 non sono le stesse poiché sono stati utilizzati costrutti diversi ma equivalenti (sono equivalenti se le loro istanze possono essere messe in corrispondenza 1 a 1).

LIBRO		CASA EDITRICE	
ISBN	titolo	nome	indirizzo
123445	Il DFM	McGraw-Hill	Via Ripamonti, 89

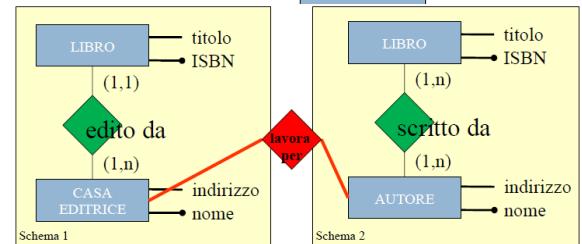
LIBRO		CASA EDITRICE	
ISBN	titolo	nome c.e.	Indirizzo c.e.
123445	Il DFM	McGraw-Hill	Via Ripamonti, 89



- **Comparabilità:** questa situazione si verifica quando R1 e R2 non sono né identici né equivalenti ma, i costrutti utilizzati e i punti di vista dei progettisti non sono in contrasto tra loro.
- **Incompatibilità:** questa situazione si verifica quando R1 e R2 sono in contrasto a causa dell'incoerenza nelle specifiche, ovvero quando la realtà modellata da R1 nega quella modellata da R2.

Ad esclusione della situazione di identità, i casi precedenti determinano dei conflitti la cui soluzione rappresenta la componente principale nella fase di integrazione. Pertanto, si verifica un **confitto** tra due rappresentazioni R1 e R2 di uno stesso concetto ogni qual volta le due rappresentazioni non sono identiche.

5. **Concetti correlati:** a seguito dell'integrazione, molti concetti diversi, ma correlati, verranno a trovarsi nello stesso schema dando vita a nuove relazioni che non erano percepibili in precedenza. Tali relazioni sono dette **proprietà inter-schema**.

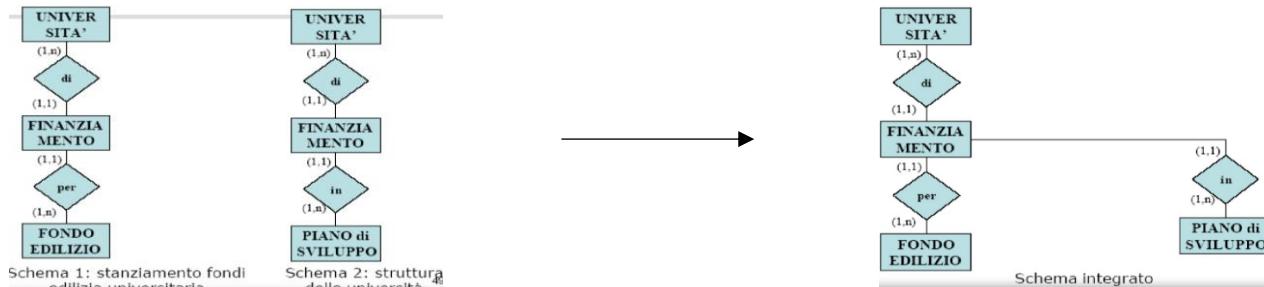


## FASI DELL'INTEGRAZIONE:

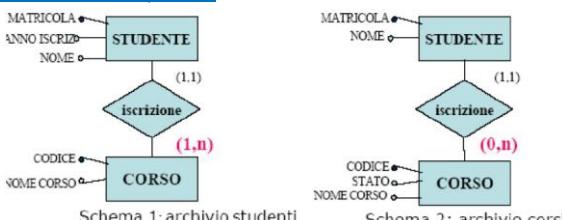
Per risolvere i problemi fin qui elencati, la sequenza dei passi da svolgere possono essere così sintetizzati:

- Preintegrazione:** Durante questa fase viene svolta l'analisi della sorgente dati, che porta a definire la politica generale dell'integrazione. Bisogna prendere decisioni sulle porzioni degli schemi che dovranno essere integrate, non tutti i dati operazionali sono utili ai fini decisionali e quindi alcuni di essi potranno essere scartati a priori, e sulle strategie di integrazione, è necessario decidere in che ordine si dovranno integrare gli schemi.
- Comparazione degli schemi:** Questa fase consiste in un'analisi comparativa dei diversi schemi e mira a identificare correlazioni e conflitti tra concetti in essi espressi. I tipi di conflitti che possono essere evidenziati sono di diverso tipo:
  - Di eterogeneità:** diversità dovute all'utilizzo di formalismi con diverso potere espressivo negli schemi sorgenti;
  - Semantici:** due schemi modellano la stessa porzione di mondo reale ma a un livello diverso di astrazione;
  - Sui nomi:** differenze nelle terminologie utilizzate nei diversi schemi sorgenti (omonimie e sinonimie);
  - Strutturali:** scelte diverse nella modellazione di uno stesso concetto, questi conflitti possono a loro volta essere di vario tipo:
    - Di tipo:** uno stesso concetto è modellato utilizzando due costrutti diversi;
    - Di dipendenza:** due o più concetti sono correlati con dipendenze diverse in schemi diversi;
    - Di chiave:** per uno stesso concetto vengono utilizzati identificatori diversi in schemi diversi;
    - Di comportamento:** diverse politiche di cancellazione/modifica dei dati vengono adottate per uno stesso concetto in schemi diversi.
- Allineamento degli schemi:** Lo scopo di questa fase è la risoluzione dei conflitti evidenziati al passo precedente, mediante primitive di trasformazione degli schemi sorgenti o dello schema riconciliato temporaneo. Alcune primitive tipiche riguardano il cambio dei nomi e dei tipi degli attributi, la modifica delle dipendenze funzionali e dei vincoli sugli schemi. È qui che il progettista definisce il mapping tra gli elementi degli schemi sorgenti e quelli dello schema riconciliato.
- Fusione e ristrutturazione degli schemi:** Nell'ultima fase gli schemi allenati vengono fusi per formare un unico schema riconciliato, solitamente si sovrappongono i concetti comuni a cui saranno collegati i rimanenti concetti degli schemi locali. Dopo questa operazione ulteriori trasformazioni permetteranno di migliorare lo schema rispetto a:
  - Leggibilità:** facilita e velocizza le successive fasi di progettazione;
  - Completezza:** ricerca di proprietà inter-schema non evidenziate in precedenza;
  - Minimalità:** eliminare ridondanza di concetti duplicati o comunque derivabili.

### Esempio di schemi compatibili in seguito all'integrazione:

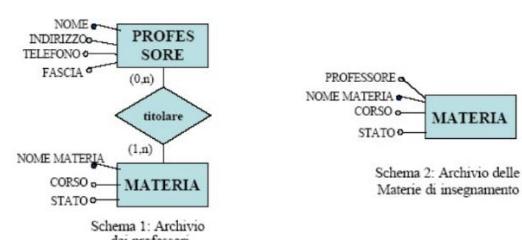


### Esempi di schemi incompatibili:



Nello schema 1 sono archiviati tutti gli studenti iscritti ad un corso universitario, mentre lo schema 2 include tutti i corsi attivi e quindi anche quelli a cui non è iscritto alcuno studente.

Nello schema integrato si sceglie la seconda soluzione, perché meno restrittiva.



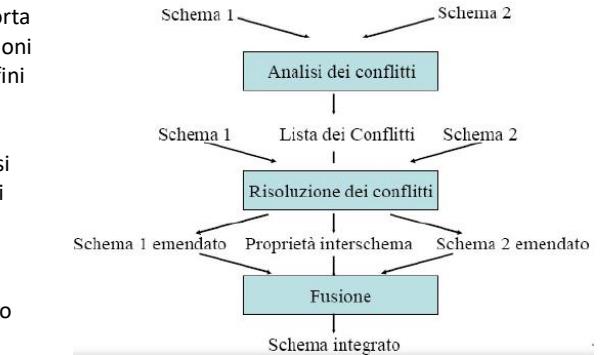
Nello schema 1 sono archiviati tutti i professori che possono essere o meno titolari di materie di insegnamento, mentre nello schema 2 sono memorizzati tutte le materie. Nello schema integrato si sceglie la prima soluzione, perché più completa e meno restrittiva.

## DEFINIZIONE DELLE CORRISPONDENZE:

Il risultato dell'analisi delle sorgenti operazionali è composto da due elementi:

- Schema riconciliato**, in cui sono stati risolti i conflitti presenti tra gli schemi locali;
- Insieme di corrispondenze** tra gli elementi presenti negli schemi sorgenti e quello dello schema destinazione (necessarie per la fase di progettazione degli ETL).

L'approccio per stabilire la corrispondenza tra i due livelli dell'architettura prevede che lo schema globale sia espresso in termini degli schemi sorgente detto **GAV (Global as view)**. Con questo approccio ad ogni concetto dello schema globale è associata una vista definita in base a concetti degli schemi sorgenti. Per fare ciò con il GAV si sostituisce ad ogni concetto dello schema globale la vista che lo definisce in termini di concetti degli schemi locali (**unfolding**).



### Esempio mapping di tipo GAV:

```
// DB1 Magazzino
ORDINI2001(chiaveO, chiaveC, data ordine, impiegato)
CLIENTE(chiaveC, nome, indirizzo, città, regione, stato)
.....
```

```
// DB2 Amministrazione
CLIENTE(chiaveC, partitalva, nome, telefono, fatturato)
FATTURE(chiaveF, data, chiaveC, importo, iva)
STORICO_ORDINI2000(chiaveO, chiaveC, data ordine, impiegato)
.....
```

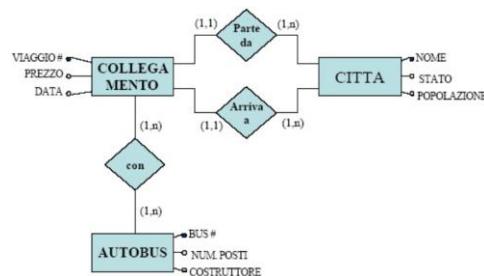
```
CREATE VIEW CLIENTE AS
SELECT CL1.chiaveC, CL1.nome, CL1.indirizzo, CL1.città, CL1.region, CL1.estado, CL2.partitalva, CL2.telefono, CL2.fatturato
FROM DB1.CLIENTE AS CL1, DB2.CLIENTE AS CL2
WHERE CL1.chiaveC = CL2.chiaveC;

CREATE VIEW ORDINI AS
SELECT * FROM DB1.ORDINI2001
UNION
SELECT * FROM DB2.STORICO_ORDINI2000;
```

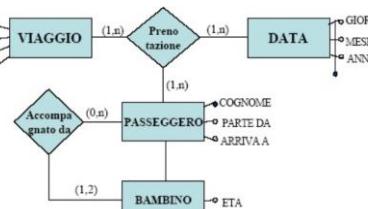
## Esercizio – Compagnia di viaggi:

Integrazione tre schemi:

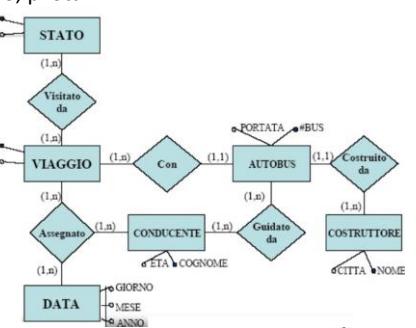
**Collegamenti:** descrive i collegamenti tra le diverse città e il bus che effettua il viaggio.



**Prenotazione:** descrive le prenotazioni ai viaggi organizzati dalla compagnia



**Viaggi quotidiani:** descrive l'utilizzo giornaliero degli autobus: costruttore, percorso, pilota.



### Integrazione Collegamento – Prenotazione:

Conflitti di nome:

- VIAGGIO#: nel primo schema l'identificatore diventa COLLEGAM#
- PARTE DA e ARRIVA A: nel primo schema relazione tra COLLEGAMENTO e CITTA; nel secondo attributi che specificano partenza e arrivo del passeggero.
- La DATA del COLLEGAMENTO non è necessariamente quella del viaggio;

Conflitti di struttura:

- CITTA: nel secondo schema le città di partenza, arrivo e quelle intermedie sono rappresentate da attributi, nel primo c'è l'entità CITTA in doppia relazione con COLLEGAMENTO.

Proprietà interschema:

- VIAGGIO è in relazione con COLLEGAMENTO.

Eliminazione delle ridondanze:

- Il concetto di CITTA è espresso completamente in relazione con COLLEGAMENTO;

### Fusione schema integrato Collegamenti - Prenotazioni con schema Viaggi quotidiani:

Conflitti di nome:

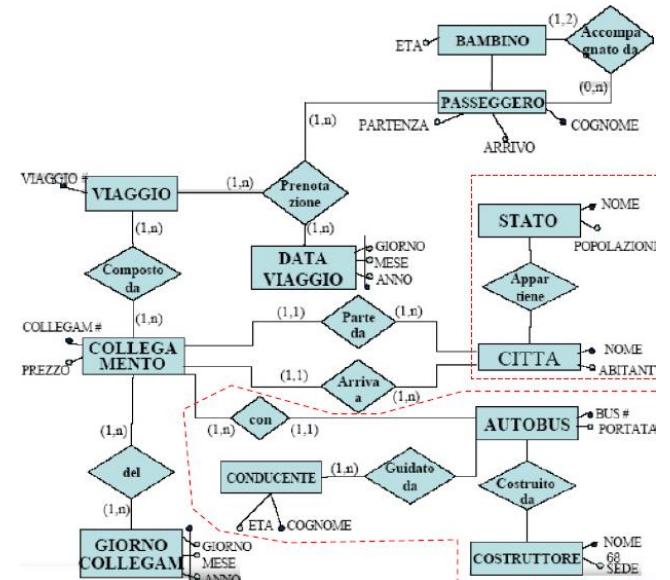
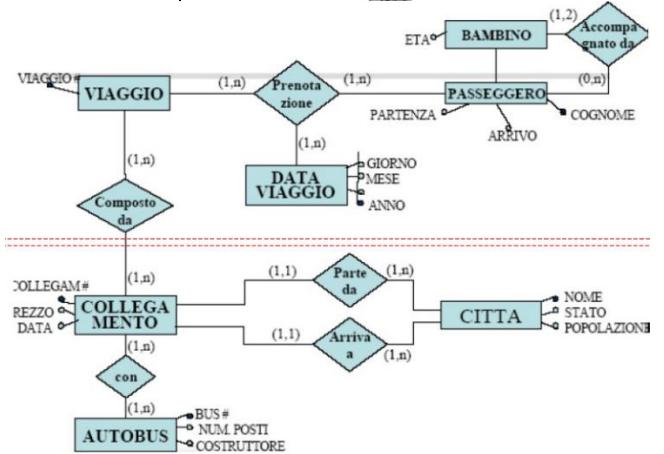
- VIAGGIO: nello schema Viaggi quotidiani rappresenta il concetto di COLLEGAMENTO.
- POPOLAZIONE: in uno schema rappresenta il numero di abitanti di una CITTA, nell'altro quello di uno STATO.
- L'attributo CITTA dell'entità COSTRUTTORE rappresenta la sede e non il concetto di CITTA di partenza e arrivo COLLEGAMENTO;
- Il numero di posti in un AUTOBUS è espresso in uno schema con l'attributo PORTATA e nell'altro con NUM\_POSTI.
- DATA: in Collegamenti-Prenotazioni è la data del VIAGGIO, in Viaggi quotidiani è il giorno in cui un CONDUCENTE guida un AUTOBUS per un COLLEGAMENTO.

Conflitti di struttura:

- COSTRUTTORE: attributo in Collegamenti-Prenotazioni e entità in Viaggi quotidiani.
- In Viaggi quotidiani si assume che un AUTOBUS possa effettuare solo un COLLEGAMENTO (card.Rel. (1,n)), mentre in Collegamenti-Prenotazioni la cardinalità della relazione è (n,n).
- In Viaggi quotidiani la DATA è in relazione (n,n) con COLLEGAMENTO mentre nell'altro schema è un attributo singolo.

Eliminazione delle ridondanze:

- La relazione Assegnato tra VIAGGIO e CONDUCENTE è ridondante.



## ANALISI DEI REQUISITI UTENTE:

### Obiettivi dell'analisi dei requisiti utente:

- Raccolta delle esigenze di utilizzo del Data mart espresse dagli utenti finali.
- Importanza strategica perché influenza tutte le decisioni da prendere durante le diverse attività, con un ruolo primario nel determinare:
  - Schema concettuale dei dati del DW;
  - Progetto dell'alimentazione;
  - Specifiche delle applicazioni per l'analisi dei dati;
  - Architettura del sistema;
  - Piano di avviamento e formazione;
  - Linee guida per la manutenzione e l'evoluzione del sistema;

**Fonti dell'analisi dei requisiti utente:** le fonti da cui attingere per i requisiti sono i **business users**, i futuri utenti del data warehouse.

- Dialogo spesso infruttuoso a causa del differente linguaggio usato;
- Porre grande cura nella fase di analisi, la soddisfazione degli utenti dipende dall'accuratezza con la quale le loro richieste ottengono un'efficace risposta nel sistema;
- Sono previste sia **Interviste** che **Riunioni coordinate**.

Gli aspetti tecnici vengono individuati mediante l'interazione con i gestori del sistema operazionale:

- I requisiti riguardano vincoli imposti al sistema e mirano a garantire livelli ottimali di prestazioni ed una facile integrazione con il sistema.

**I fatti:** sono concetti su cui gli utenti finali del Data mart baseranno il processo decisionale. Ogni fatto descrive una categoria di eventi che si verificano in azienda. Caratteristiche che guidano il progettista verso la determinazione dell'insieme dei fatti per il Data mart sono:

- Aspetti dinamici: gli eventi che vengono descritti dal fatto devono avere una componente temporale;
- Dominio applicativo;
- Tipo di analisi che l'utente vuole eseguire (un fatto di interesse per un Data mart potrebbe non esserlo per un altro).

Individuare i fatti non è sufficiente. Per ognuno di essi è necessario disporre di informazioni di contorno, definite con l'aiuto della documentazione del livello riconciliato: Possibili dimensioni (granularità), Possibili misure e Intervallo di storicità.

**Granularità:** focalizzare le dimensioni di un fatto è importante per determinare la granularità, ovvero il più fine livello di dettagli a cui i dati saranno rappresentati nel Data mart. La scelta della granularità di rappresentazione di un fatto nasce da un compromesso tra due esigenze contrapposte:

- Raggiungere un'elevata flessibilità di utilizzo, che richiederebbe di mantenere la stessa granularità del livello operazionale;
- Conseguire buone prestazioni con la necessità di avere un consistente grado di sintesi dei dati.

**Misure e intervallo di storicità:** la valutazione delle misure con cui quantificare ciascun fatto ha un ruolo preliminare e orientativo, in quanto la definizione dettagliata delle misure da abbinare al fatto è rimandata alla successiva fase di progettazione concettuale.

È l'arco temporale che gli eventi memorizzati nel Data mart devono coprire (valori tipici variano da 3 a 5 anni).

#### Caso di studio:

Data mart per la gestione di approvvigionamenti e vendite in una catena di supermercati.

#### Requisiti utente -->

**Carico di lavoro preliminare:** Raccolta delle specifiche relative alle interrogazioni di analisi più frequenti sul Data mart -->

Fatto	Possibili dimensioni	Possibili misure	Storicità
inventario di magazzino	prodotto, data, magazzino	quantità in magazzino	1 anno
vendite	prodotto, data, negozio	quantità venduta, importo, sconto	5 anni
linee d'ordine	prodotto, data, fornitore	quantità ordinata, importo, sconto	3 anni

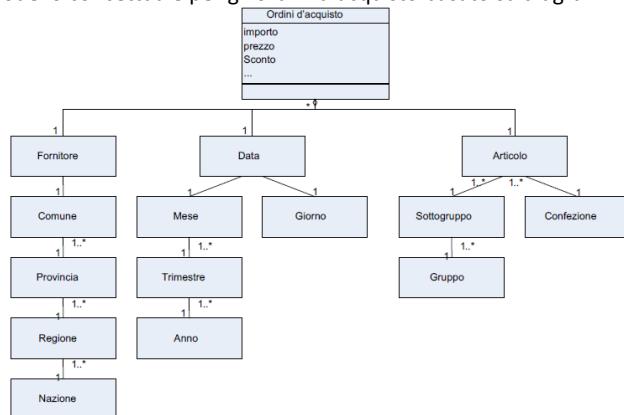
Fatto	Interrogazione
Inventario di magazzino	<ul style="list-style-type: none"> <li>• quantità media di prodotto presente mensilmente in tutti i magazzini</li> <li>• andamento giornaliero delle scorte complessive per ogni tipo di prodotto</li> <li>• prodotti per i quali è stata esaurita la scorta di magazzino contemporaneamente in almeno un'occasione durante la settimana scorsa.</li> </ul>
Vendite	<ul style="list-style-type: none"> <li>• incasso totale giornaliero di ciascun negozio</li> <li>• per un negozio, incassi relativi alle diverse categorie di prodotti durante un certo giorno</li> <li>• riepilogo annuale degli incassi per regione relativamente ad un dato prodotto</li> <li>• quantità totali di ciascun tipo di prodotto venduto durante l'ultimo mese</li> </ul>
Linee d'ordine	<ul style="list-style-type: none"> <li>• quantità totale ordinata annualmente presso un certo fornitore</li> <li>• importo giornaliero ordinato nell'ultimo mese per un certo tipo di prodotto</li> <li>• sconto massimo applicato da ciascun fornitore durante l'ultimo anno per ciascuna categoria di prodotto</li> </ul>

#### MODELLO CONCETTUALE:

Affronta il problema della traduzione dei requisiti in termini di un modello astratto, indipendente dal DBMS. Non esistono standard di modello o di processo. Il modello Entity-relationship (ER) è diffuso come strumento di modellizzazione concettuale dei Data mart. Esso è però orientato alle associazioni tra i dati e non alla sintesi, è sufficientemente espressivo per rappresentare la maggior parte dei concetti, ma non è in grado di mettere in luce il modello multidimensionale.

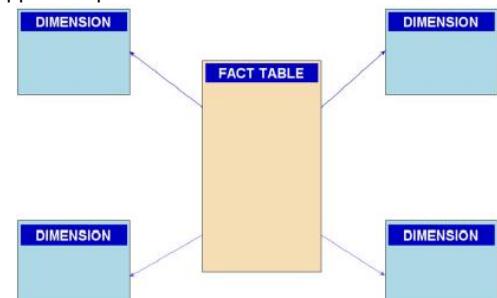
#### Modellazione concettuale in UML:

Un modello concettuale per gli ordini d'acquisto basato su diagramma UML.



#### Schema a Stella (Star Schema):

Lo schema a stella è un modello logico che può essere usato per la modellazione concettuale. Usare lo schema a stella per i Data mart equivale a modellare uno schema logico per un DB relazionale. Questo approccio porta a schemi fortemente denormalizzati.



#### DIMENSIONAL FACT MODEL (DFM):

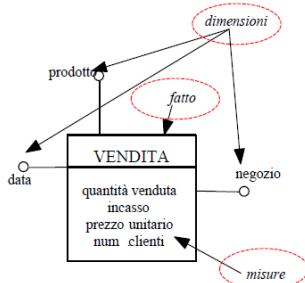
È un **modello concettuale** concepito per il supporto allo sviluppo di **Data mart**. È una specializzazione del modello multidimensionale per applicazioni di Data warehousing. È un modello concettuale grafico per Data mart, pensato per:

- Supportare efficacemente il progetto concettuale;
- Creare un ambiente su cui formulare in modo intuitivo le interrogazioni dell'utente;
- Permettere il dialogo tra progettista e utente finale per raffinare le specifiche dei requisiti;
- Creare una piattaforma stabile da cui partire per il progetto logico (indipendentemente dal modello logico target);
- Restituire una documentazione a posteriori espressiva e non ambigua.

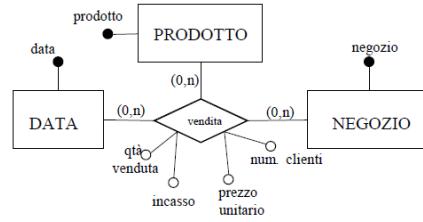
La rappresentazione concettuale generata dal DFM consiste in un insieme di **schemi di fatto**. Gli elementi di base modellati dagli schemi di fatto sono i fatti, le misure, le dimensioni e le gerarchie.

## Esempio DFM:

Un semplice schema di fatto per le vendite:



Schema ER corrispondente:



## DFM – COSTRUTTI DI BASE:

Un **fatto** è un concetto di interesse per il processo decisionale. Tipicamente modella un insieme di eventi che accadono nell’impresa (ES: vendite, spedizioni). È essenziale che un fatto abbia aspetti dinamici, ovvero evolva nel tempo.

Una **misura** è una proprietà numerica di un fatto e ne descrive un aspetto quantitativo di interesse per l’analisi (ES: ogni vendita è misurata dal suo incasso). Le misure vengono in genere usate per effettuare calcoli.

Una **dimensione** è una proprietà con dominio finito di un fatto e ne descrive una coordinata di analisi (dimensioni, tipiche per il fatto vendite sono prodotto, negozio, data). Un fatto esprime una associazione molti-a-molti tra le dimensioni ed hanno natura dinamica, rappresentata da una dimensione temporale.

Un **evento primario** è una particolare occorrenza di un fatto, individuata da una ennupla costituita da un valore per ciascuna dimensione. A ciascun evento primario è associato un valore per ciascuna misura. Esempio: il giorno 10/10/2001 il negozio ‘DiTutto’ ha venduto 10 confezioni di detersivo Brillo per un incasso complessivo di 25€.

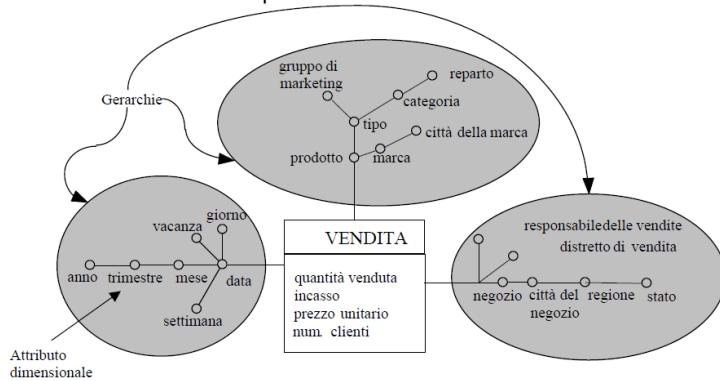
## DFM – ATTRIBUTI DIMENSIONALI:

Gli **attributi dimensionali** sono le dimensioni e gli attributi che le descrivono. Esempio: un prodotto è descritto da tipo, categoria, marca. Le relazioni tra gli attributi dimensionali sono espresse dalle gerarchie. Una **gerarchia** è un albero direzionale i cui nodi sono attributi dimensionali e i cui archi modellano associazioni molti-a-uno tra coppie di attributi dimensionali. Racchiude una dimensione, posta alla radice dell’albero, e tutti gli attributi dimensionali che la descrivono.

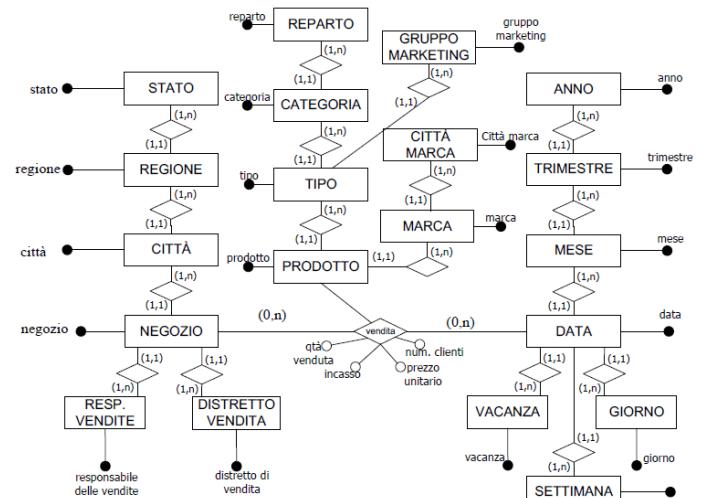
**ATTENZIONE: Gerarchie ER != Gerarchie DFM**

## DFM ARRICCHITO:

Schema di fatto arricchito per la Vendita:



Schema ER del DFM Vendita:



## NAMING CONVENTIONS:

Tutti gli attributi dimensionali in ciascuno schema di fatto devono avere nomi diversi. Eventuali nomi uguali dicono essere differenziati qualificandoli con il nome di un attributo dimensionale che li precede nella gerarchia. Ad esempio, **warehouse city** è la città in cui si trova un magazzino, mentre **store city** è la città in cui si trova un negozio. I nomi degli attributi non dovrebbero riferirsi esplicitamente al fatto a cui appartengono. Es: si evitino shipped product e shipment date. Attributi con lo stesso significato in schemi diversi devono avere lo stesso nome.

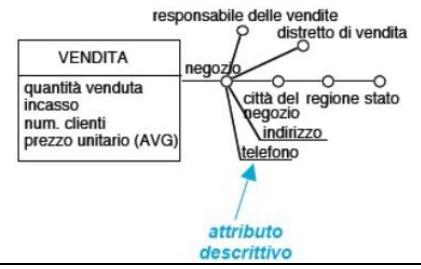
## EVENTO SECONDARIO:

Dato un insieme di attributi dimensionali, ciascuna ennupla di valori individua un **evento secondario** che aggrega tutti gli eventi primari corrispondenti. A ciascun evento secondario è associato un valore per ciascuna misura, che riassume in sé tutti i valori della stessa misura negli eventi primari corrispondenti. Es: Le vendite possono essere raggruppate a seconda della categoria dei prodotti venduti, oppure a seconda del mese in cui sono effettuate le vendite, oppure a seconda della città in cui si trova in negozio, ecc...

## ATTRIBUTI DESCRIPTIVI:

Specificano le proprietà degli attributi dimensionali di una gerarchia, e sono determinati tramite dipendenze funzionali. Non possono essere usati per l’aggregazione poiché hanno spesso domini con valori continui. Un attributo descrittivo non può essere usato per identificare singoli eventi né per effettuare calcoli.

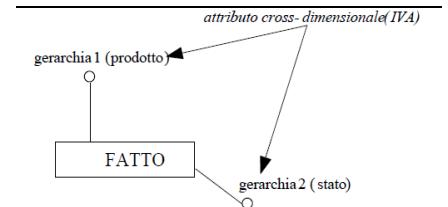
Es: numero di telefono di un negozio, indirizzo di un negozio.



## ATTRIBUTI CROSS-DIMENSIONALI:

Sono attributi dimensionali o descrittivi il cui valore è determinato mediante la combinazione di due o più attributi dimensionali. Gli attributi dimensionali possono appartenere anche a gerarchie diverse.

Ese: l'IVA su un prodotto dipende sia dalla categoria del prodotto che dallo stesso in cui esso è venduto.



## Esempio di gerarchia temporale completa:

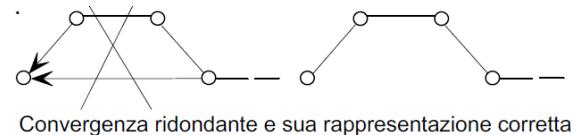
L'attributo booleano **feriale?** È determinato congiuntamente dal **giorno** e dal booleano **vacanza?**



## CONVERGENZA:

La **convergenza** riguarda la struttura delle gerarchie. Sullo schema di fatto le convergenze sono denotate da due o più archi, in genere appartenenti alla stessa gerarchia, che terminano nello stesso attributo dimensionale. In presenza di una gerarchia che non ha una struttura ad albero non è più possibile determinare univocamente il verso degli archi e, per fare ciò, gli archi convergenti devono essere orientati.

Attributi apparentemente uguali non determinano sempre una convergenza. Se uno dei percorsi alternativi non comprende attributi intermedi, non ha ragione di esistere, la convergenza è infatti del tutto ovvia grazie alla transitività delle dipendenze funzionali.

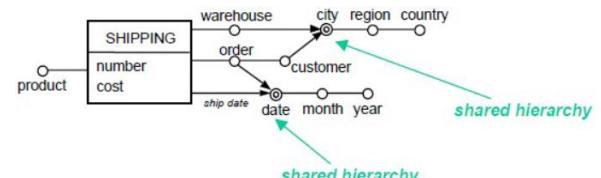
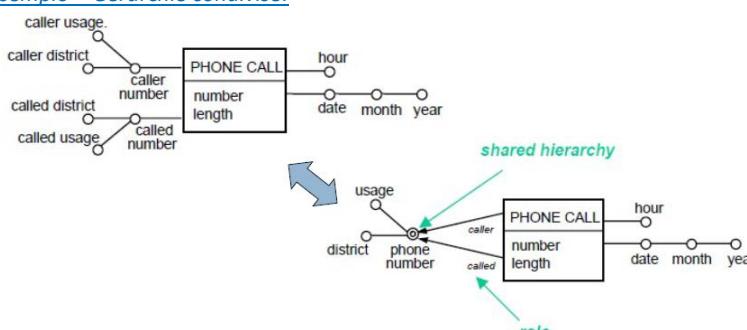


## GERARCHIE CONDIVISE:

Negli schemi di fatto spesso si rende necessario duplicare intere porzioni di gerarchie e ciò comporta l'uso di diversi nomi per evitare ambiguità.

Tramite le **gerarchie condivise** si introduce una notazione grafica abbreviata che migliora la leggibilità dello schema. Si introducono quando si hanno significati diversi per lo stesso tipo di dati e il significato viene espresso inserendo il ruolo sull'arco entrante della gerarchia.

## Esempio – Gerarchie condivise:



## ARCHI MULTIPLI:

Un **arco multiplo** tra due attributi a e b indica che ad ogni singolo valore di a possono corrispondere valori di b.

Ese: Schema di fatto per le vendite dei libri -->

Il significato di un arco multiplo che va da un attributo autore ad un attributo libro sta nel fatto che tra autore e libro esiste una **associazione M-N**.

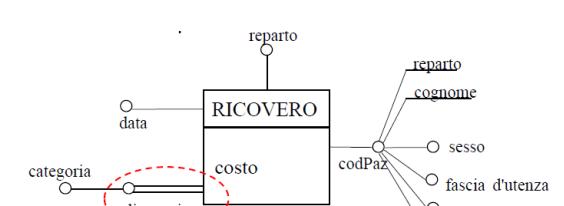
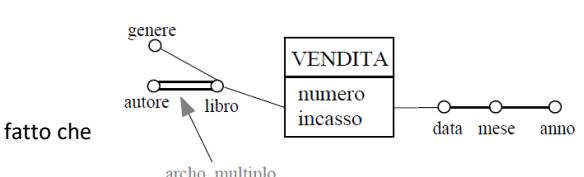
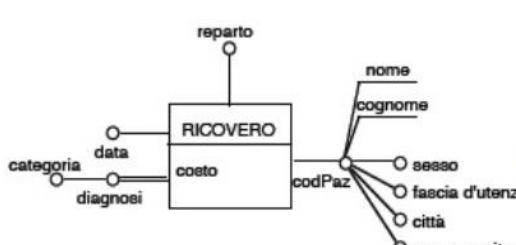
## AGGREGAZIONE SU DIMENSIONI:

Ese: Schema di fatto per ricoveri -->

Nel momento in cui un attributo entra in una dimensione piuttosto che in un attributo qualsiasi, il caso diventa più complesso.

Infatti, è possibile aggregare i ricoveri in base alle diagnosi in uscita, ma anche selezionare le diagnosi in base ai ricoveri.

Alternativamente:



## ARCHI OPZIONALI:

Vengono impiegati per modellare associazioni dello schema di fatto non definite per un particolare sottoinsieme di eventi. Si rappresenta con un trattino sull'arco. Se r è l'**arco opzionale**, bisogna distinguere se esso determina un attributo o una dimensione. Se r determina la dimensione d allora essa è opzionale, ossia esistono alcuni eventi primari identificati solo dalle altre dimensioni.

Esempio: la promozione su un prodotto, identificato da una dimensione, vale solo per alcune combinazioni di **prodotto-negozi.data**



## COPERTURA TRA ARCHI OPZIONALI:

Se esistono più archi opzionali uscenti da uno stesso attributo è possibile definire la **copertura**, ossia stabilire una relazione tra le diverse opzionalità. Sia  $a$  un attributo dimensionale con archi opzionali verso i propri figli  $b_1, \dots, b_m$ . Allora la copertura si dice:

- Totale** se per ogni valore di  $a$  è sempre associato almeno un valore dei figli o **parziale** se esistono valori di  $a$  per i quali tutti i figli sono indefiniti;
- Esclusivo** se per ogni valore di  $a$  si ha al massimo un valore per uno dei figli o **sovraposta** se invece esistono valori di  $a$  abbinati a due o più figli

I tipi di copertura sono identificati da T-E, T-S, P-E, P-S.

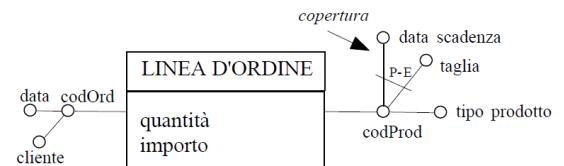
### Esempio di copertura:

Copertura per un insieme di archi opzionali.

Prodotti di tre tipi: alimentari, abbigliamento, casalinghi.

Quindi **data, scadenza e taglia** sono definiti solo per alimentari

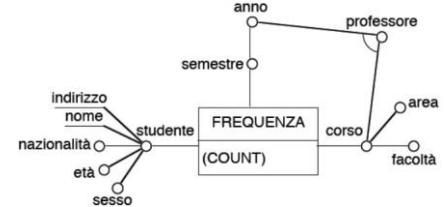
E casalinghi rispettivamente: la copertura è P-E.



## SCHEMI DI FATTO VUOTI:

Uno schema di fatto si dice vuoto se non ha misure:

- In questo caso, il fatto registra solo il verificarsi di un evento.



## ADDITIVITÀ:

L'**aggregazione** richiede di definire un operatore adatto per comporre i valori delle misure che caratterizzano gli eventi primari in valori da abbinare a ciascun evento secondario.

## MISURE ADDITIVE:

Una misura è detta **additiva** su una dimensione se i suoi valori possono essere aggregati lungo la corrispondente gerarchia tramite l'operatore di somma. Una misura è **non-additiva** se non può essere aggregata lungo una data gerarchia tramite l'operatore di somma. Una misura è **non-aggregabile** se non può essere aggregata lungo una qualsiasi gerarchia tramite l'uso di qualsiasi operatore di aggregazione.

Una misura **non-additiva** è **non-aggregabile** se nessun operatore di aggregazione può essere usato su di essa.

### Esempio:

Il livello di inventario non è additivo sul tempo, ma lo è sulle altre dimensioni:

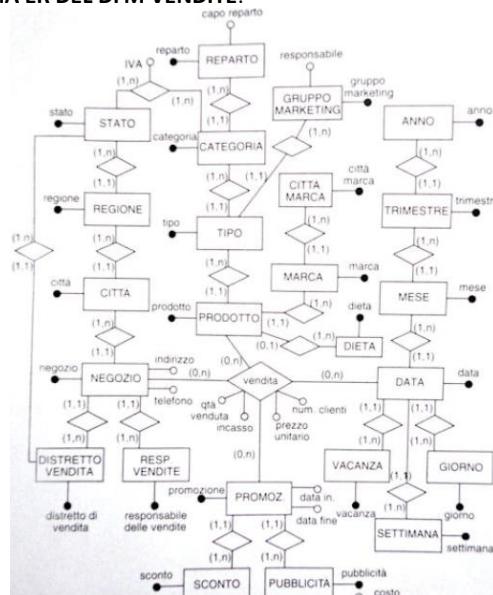


## TIPI DI MISURE ADDITIVE:

Esistono tre tipi di misure additive:

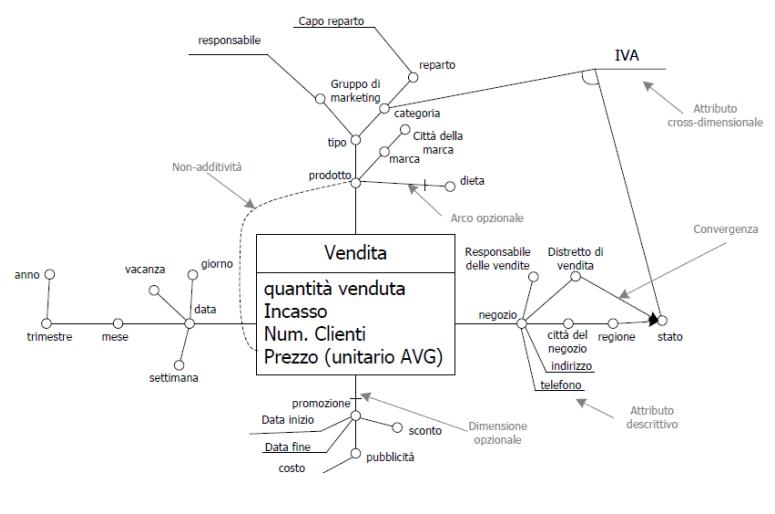
- di flusso** (periodo temporale, come incasso mensile, num\_prodotti)
  - Può essere valutata cumulativamente alla fine di un periodo di tempo;
  - Può essere aggregata tramite tutti gli operatori standard;
- di livello** (particolari instanti di tempo, come gli abitanti di una città)
  - Valutata in un preciso istante (snapshot);
  - Non additiva lungo la dimensione temporale;
- unitarie** (in particolari istanti di tempo, ma in termini relativi come il prezzo unitario di un prodotto o una percentuale di sconto)
  - Valutata ad un certo istante ed espressa con termini relativi;
  - Non additiva lungo qualsiasi dimensione;

## SCHEMA ER DEL DFM VENDITE:



## DFM COMPLETO DELLE VENDITE:

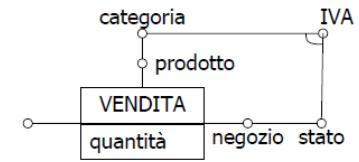
Gli attributi descrittivi sono sempre foglie delle gerarchie e sono rappresentati nel DFM da linee orizzontali.



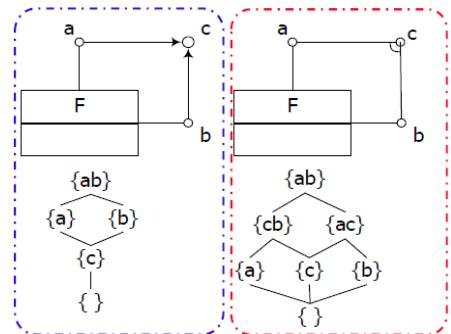


## AGGREGAZIONE IN PRESENZA DI CONVERGENZE E ATTRIBUTI CROSS-DIMENSIONALI:

Una convergenza nello schema di fatto è del tutto trasparente ai fini dell'aggregazione. Per verificare la semantica dell'aggregazione in presenza di attributi cross-dimensional dobbiamo risalire agli eventi primari che includono l'attributo cross-dimensionale. L'attributo cross-dimensionale IVA determinato da categoria e stato. Ciascun evento primario è associato ad un prodotto e ad un negozio (quindi ad una categoria e ad un posto). Essendo definito univocamente un valore di IVA per ogni evento primario, gli eventi secondari sui pattern che includono IVA risultano immediatamente determinati.



Differenza tra convergenza e attributo cross-dimensionale:



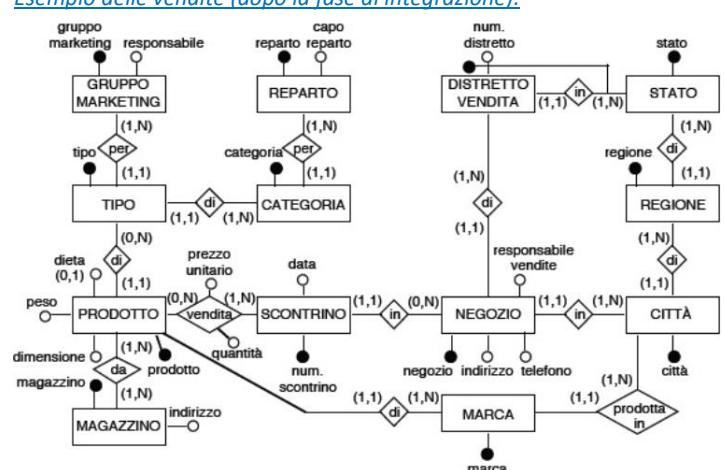
## PROGETTAZIONE CONCETTUALE:

La progettazione concettuale è guidata dai dati.

La tecnica per la progettazione concettuale di un Data mart a partire dalle sorgenti operazionali, secondo il **DFM**, consiste nei seguenti passi:

- 1) Definizione dei fatti;
- 2) Per ciascun fatto:
  - a. Costruzione dell'albero degli attributi
  - b. Potatura e innesto dell'albero degli attributi
  - c. Definizione delle dimensioni/misure.
  - d. Creazione dello schema di fatto

## Esempio delle vendite (dopo la fase di integrazione):



## DEFINIZIONE DEI FATTI:

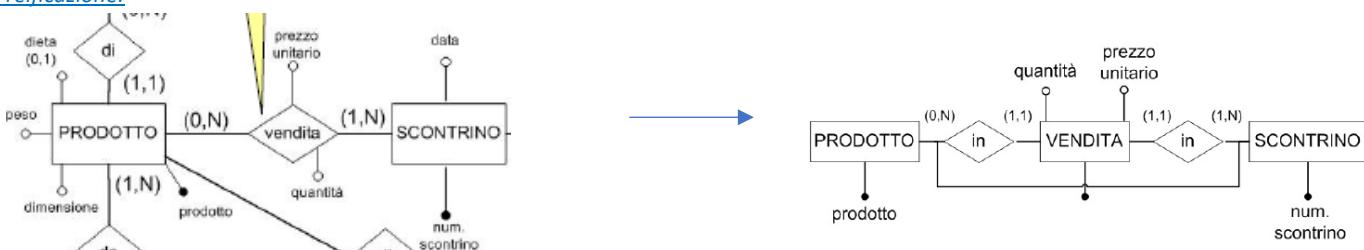
I **fatti** sono concetti di interesse primario per il processo decisionale. In uno schema ER un fatto può corrispondere ad un'**entità** F, oppure ad un'**associazione** n-aria R tra le entità  $E_1, E_2, \dots, E_n$ .

## PROCESSO DI REIFICAZIONE:

Se il fatto corrisponde ad un'**entità** del modello ER, non è richiesta alcuna modifica. Se il fatto corrisponde ad una **relazione** n-aria R, è preferibile trasformare R in un'entità F che possieda tutti gli attributi della relazione e la cui chiave è data dall'unione delle chiavi delle singole entità coinvolte. Ciascuno dei rami di R viene sostituito con un'associazione binaria  $R_i$  tra F ed  $E_i$  tale che :

- $\min(F, R_i) = \max(F, R_i) = 1$
- $\min(E_i, R_i) = \min(E_i, R)$
- $\max(E_i, R_i) = \max(E_i, R)$

## Esempio reificazione:



## LINEE GUIDA NELLA SCELTA DEI FATTI:

Le entità che rappresentano archivi aggiornati frequentemente (es. VENDITA) sono buoni candidati per la definizione dei fatti.

Le entità che rappresentano proprietà strutturali del dominio, corrispondenti ad archivi quasi statici (es. NEGOZIO e CITTA'), non lo sono.

In realtà, tale regola non è sempre valida in quanto la scelta del fatto dipende in maniera significativa sia dal dominio applicativo che dal tipo di analisi che l'utente intende eseguire. Ciascun fatto identificato sullo schema sorgente diviene la radice di un differente schema di fatto. Nel caso in cui diverse entità siano candidate ad esprimere lo stesso fatto, conviene sempre scegliere come fatto F l'entità, a partire dalla quale, è possibile costruire l'albero che include il maggior numero di attributi.

## COSTRUZIONE ALBERO DEGLI ATTRIBUTI:

**Albero degli attributi:** data un'entità F designata come fatto, si definisce albero degli attributi quello che soddisfa i seguenti requisiti:

- Ogni vertice corrisponde ad un attributo semplice o composto dello schema sorgente;
- La radice corrisponde all'identificativo di F;
- Per ogni vertice v, l'attributo corrispondente determina funzionalmente tutti gli attributi che corrispondono ai discendenti di v.

## ALGORITMO PER LA COSTRUZIONE DELL'ALBERO DEGLI ATTRIBUTI:

La procedura proposta naviga ricorsivamente le dipendenze funzionali espresse dagli identificatori e dalle associazioni ...-a-1 dello schema ER. L'entità a partire dalla quale viene innescato il processo è quella scelta come fatto.

Quando si esamina un'entità E si crea nell'albero un vertice v corrispondente all'identificatore di E, e gli si aggiunge un vertice per ogni altro attributo di E.

Per ogni associazione R da E verso un'entità G, con cardinalità massima 1, si aggiungono a v tanti figli quanti sono gli attributi di R, per poi ripetere il procedimento per G.

```

root = nuovoVertice(ident(F)); /* ident(F) e' l'identificatore di F, la radice dell'albero e' etichettata con l'identificatore dell'entita' scelta come fatto */

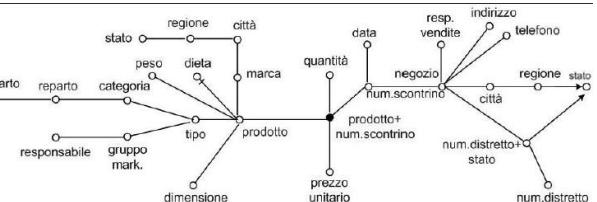
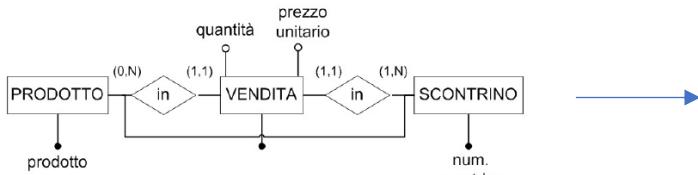
traduci(F,root);

procedura traduci(E,v);
//E e' l'entita' corrente dello schema sorgente, v il vertice corrente dell'albero
{
    per ogni attributo a di E tale che a!=ident(E)
        aggiungiFiglio(v, nuovoVertice(a))
        // aggiunge al vertice v un figlio a

    per ogni entita' G connessa ad E da una associazione R tale che max(E,R)=1
    {
        per ogni attributo b di R
            aggiungiFiglio(v, nuovoVertice(b));
            // aggiunge al vertice v un figlio b
        prossimo = nuovoVertice(ident(G));
        //crea un nuovo vertice con il nome dell'identificatore di G...
        aggiungiFiglio(v, prossimo);
        // ... lo aggiunge a v come figlio ...
        traduci(G, prossimo);
        //... e innesca la ricorsione
    }
}
}

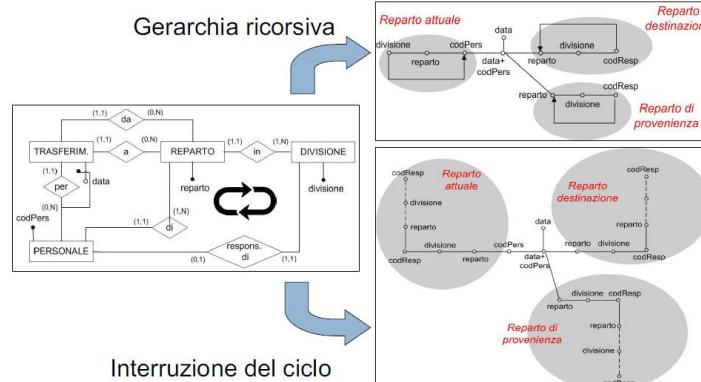
```

Albero degli attributi ottenuto a partire dallo schema reificato (*include tutto lo schema ER*):



**Eccezioni:** l'applicazione dell'algoritmo è condizionata da specifiche caratteristiche strutturali dello schema sorgente:

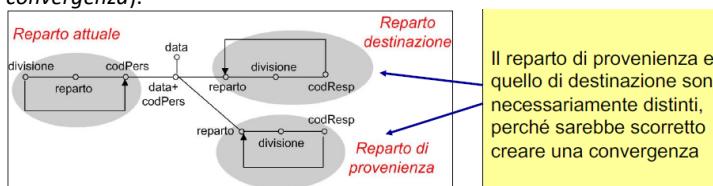
### 1) Presenza di un ciclo di associazioni molti a uno/uno a uno:



### 2) Raggiungimento della stessa entità E attraverso cammini differenti

Se si raggiunge due volte la stessa entità E attraverso cammini differenti, vengono generati nell'albero due vertici omologhi v' e v''.

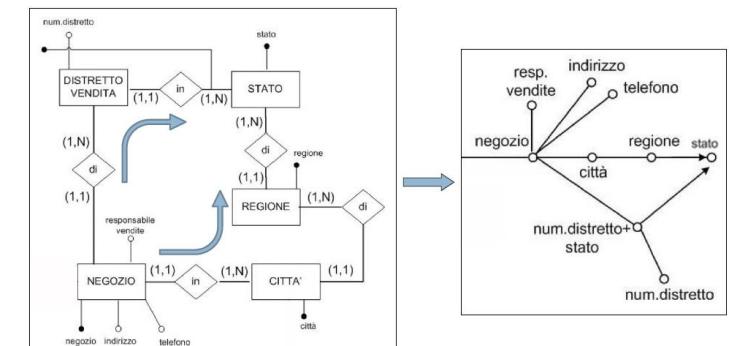
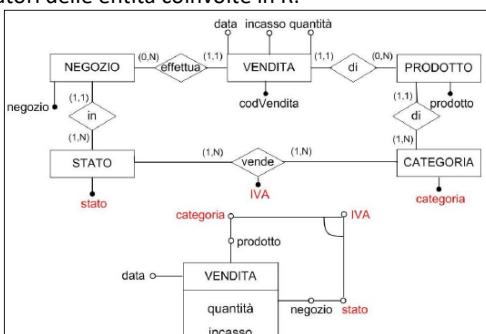
Se ogni istanza di F determina E (i.e. F -> E) indipendentemente dal cammino seguito, allora v' e v'' possono coincidere (*si genera una convergenza*).



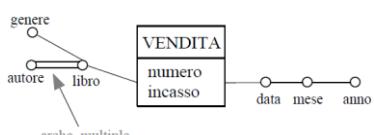
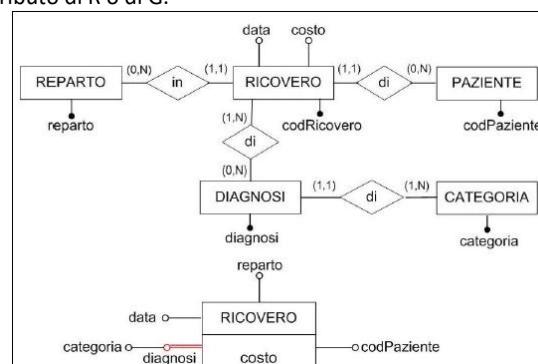
### 3) Presenza di associazioni ...-a-molti e di attributi multipli

Eventuali associazioni ...-a-molti ( $\max(E,R) > 1$ ) e attributi multipli presenti nello schema ER non vengono inseriti automaticamente nell'albero degli attributi. Esse possono generare attributi cross-dimensionali o archi multipli disegnati manualmente dal progettista sullo schema di fatto al termine della progettazione concettuale.

Un **attributo cross-dimensional** corrisponde in genere a un attributo posto su un'associazione molti-a-molti R dello schema ER; i suoi padri nello schema di fatto corrispondono allora agli identificatori delle entità coinvolte in R.



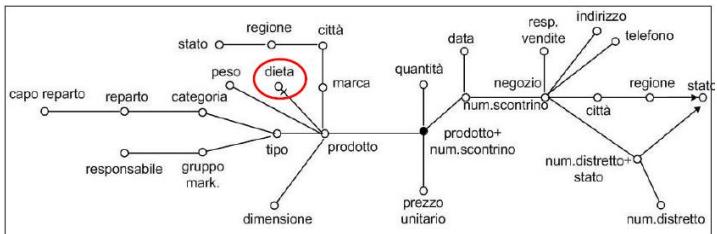
Un **arco multiplo** corrisponde ad un'associazione R...-a-molti da un'entità E ad un'entità G. Nello schema di fatto esso potrà connettere l'identificatore di E con un attributo di R o di G.



#### 4) Presenza di associazioni o attributi opzionali

Gli **attributi opzionali** ( $\text{min}(E,R)=0$ ) portano a collegamenti opzionali.

L'esistenza di un collegamento opzionale deve essere sottolineata nell'albero degli attributi con un trattino sugli archi corrispondenti ad associazioni o attributi opzionali nello schema ER:

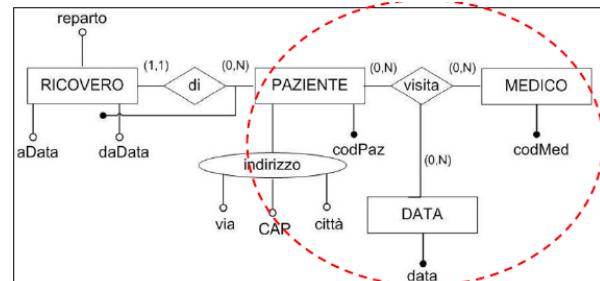


#### 5) Presenza di associazioni n-arie

Eventuali associazioni n-arie saranno trasformate in n associazioni binarie attraverso il processo di reificazione.

Molte delle associazioni n-arie hanno molteplicità massima maggiore di 1 sui rami (dunque sono inserite nell'algoritmo):

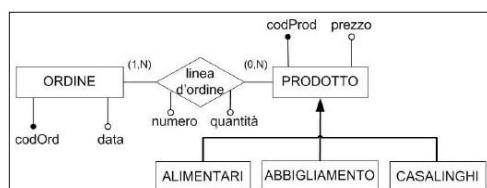
- questa situazione porta ad n-associazioni binarie 1-a-molti che non possono essere inserite automaticamente nell'albero degli attributi.



#### 6) Presenza di gerarchie di specializzazione

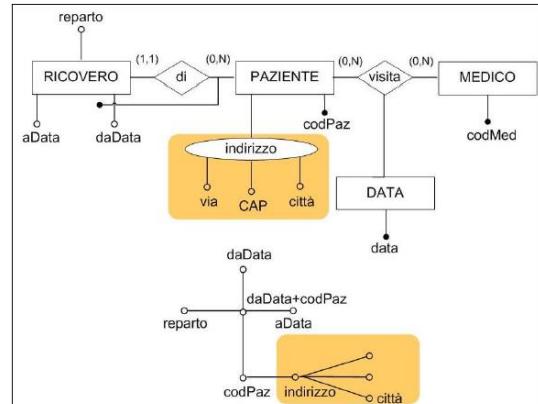
Le gerarchie dell'ER possono essere trattate dall'algoritmo come delle semplici **associazioni 1-a-1 opzionali** tra le super entità e la sotto entità.

In alternativa è possibile limitarsi ad aggiungere al noto corrispondente alla chiave della super entità un figlio che funga da discriminatore tra le diverse sotto entità (*sotto entità unite con la super entità*).



#### 7) Presenza di attributi composti

In presenza di un attributo composto c, che consiste degli attributi semplici  $a_1, \dots, a_n$  tale attributo viene inserito nell'albero degli attributi come un vertice c con figli  $a_1, \dots, a_n$ .



#### POTATURA E INNESTO DELL'ALBERO DEGLI ATTRIBUTI:

In genere, non tutti gli attributi dell'albero sono di interesse per il Data mart.

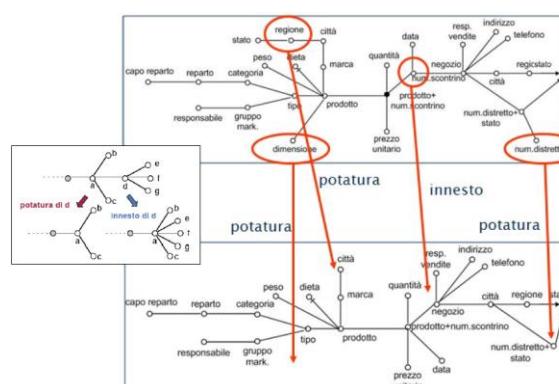
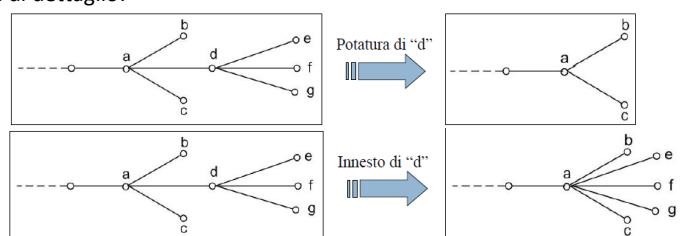
**ES:** Il numero di fax di un cliente difficilmente potrà rivelarsi utile ai fini decisionali per cui il Data mart è progettato.

È dunque possibile manipolare l'albero al fine di eliminare e/o aggiungere livelli di dettaglio.

**Potatura di un vertice v:** si effettua eliminando l'intero sottoalbero radicato in v. Gli attributi eliminati non verranno inclusi nello schema di fatto. Non potranno essere usati per aggregare dati.

**Innesto di un vertice v:** viene utilizzato quando, sebbene un vertice esprima un'informazione non interessante, è necessario mantenere nell'albero i suoi discendenti (che verranno collegati direttamente al padre del vertice da innestare).

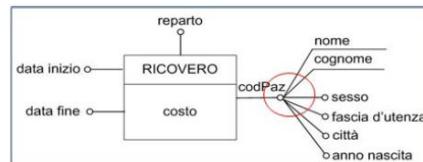
Esempio di potatura e innesto:



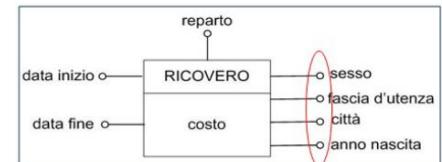
## DEFINIZIONE DELLE DIMENSIONI:

Le dimensioni devono essere scelte nell'albero degli attributi tra i vertici figli della radice, possono corrispondere ad attributi discreti o a intervalli di valori di attributi discreti o continui.

ES: in un DW in ambito sanitario, un classico problema riguarda il mantenimento o meno della granularità del paziente.



Nel primo schema la dimensione codPaz è mantenuta (**grana transazionale**)



Nel secondo schema abbiamo rinunciato alla granularità del singolo paziente innestando codPaz e introducendo le dimensioni sesso, fascia d'utenza, città e anno di nascita (**grana temporale**)

In un DW non siamo interessati, di solito, ad interrogazioni di natura operazionale (che sono prerogativa dei DB relazionali). Si preferisce in generale una grana temporale. Tuttavia, se è necessario mantenere una granularità massima, si procede nel seguente modo:

- Duplicare il vertice radice nell'albero degli attributi, il nuovo vertice sarà collegato alla vecchia radice tramite un'associazione 1-a-1 e la radice non avrà altri archi uscenti.
- Scegliere le dimensioni:
  - 1) Marcare come dimensione l'unico figlio diretto della radice (schema di fatto monodimensionale), oppure
  - 2) Trasformare in figli diretti della radice alcuni attributi dell'albero (schema di fatto multidimensionale).

## IL TEMPO:

Il tempo è un fattore chiave nella progettazione di un DW. Gli schemi sorgente possono essere classificati rispetto al tempo come:

- **Snapshot:** descrivono lo stato corrente del dominio applicativo, vengono mantenute solo le versioni dei dati correnti che rimpiazzano continuamente le precedenti. Il tempo viene aggiunto manualmente come dimensione nello schema di fatto.
- **Storici:** descrivono l'evoluzione del dominio applicativo durante un intervallo di tempo, anche le vecchie versioni dei dati continuano ad essere mantenute. Pertanto, in tal caso il tempo diventa un ovvio candidato alla definizione di una dimensione nello schema di fatto.

## VALID TIME E TRANSACTION TIME:

- **Valid time:** istante in cui l'evento si verifica nel mondo aziendale.
- **Transaction time:** istante in cui l'evento è memorizzato nel database.

Non necessariamente entrambe le coordinate temporali (**valid** e **transaction**) devono essere mantenute. La scelta di quale debba essere mantenuta dipende dal tipo di interrogazioni, che possono essere:

- 1) *Interrogazioni che richiedono il tempo di validità.* ES: in quali mesi gli studenti preferiscono iscriversi ad un certo corso;
- 2) *Interrogazioni che richiedono il tempo di transazione.* ES: confrontare il numero totale degli iscritti con quello degli anni precedenti.
- 3) *Interrogazioni che richiedono entrambi i tempi.* ES: stabilire qual è il ritardo medio nella trasmissione di pagamenti.

## MODELLAZIONE DEL TEMPO NEL DFM:

A seconda del tipo di interrogazione il tempo viene modellato concettualmente in maniera diversa:

- 1) **Modellazione del solo tempo di validità:** soluzione che riflette la semantica del tempo comunemente adottata negli schemi di fatto. Consente solo interrogazioni del primo tipo non essendo disponibili, prima dell'aggiornamento retrospettivo, i valori che riflettono la situazione reale.
- 2) **Modellazione del solo tempo di transazione:** soluzione sconsigliata se non per i casi in cui il tempo di transazione ha una semantica rilevante all'interno del dominio applicativo.
- 3) **Modellazione di entrambi i tempi:** è la soluzione più generale e consente la formulazione di tutti e tre i tipi di interrogazione.

## DEFINIZIONE DELLE MISURE:

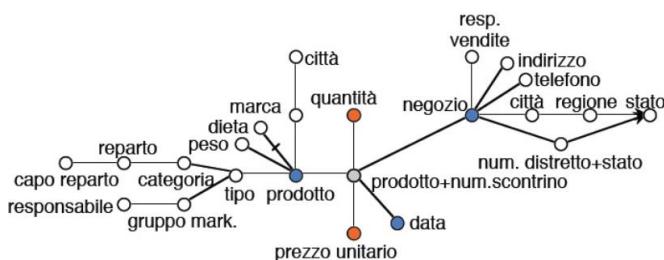
Se tra le dimensioni compaiono tutti gli attributi che costituiscono un'entità fatto (schema a grana transazionale), allora le misure corrispondono ad attributi numerici che siano figli della radice. Se lo schema è a grana temporale, le misure devono essere definite applicando, ad attributi numerici dell'albero, funzioni di aggregazione (SUM, AVG, MAX, MIN) che operano su tutte le istanze di F corrispondenti a ciascun evento primario. Scegliere come misura un attributo che non è figlio diretto della radice significa rinunciare ad una dipendenza funzionale.

## GLOSSARIO PER IL CALCOLO DELLE MISURE:

Occorre definire un glossario che associa ciascuna misura ed un'espressione che descrive come essa possa essere calcolata. Se la granularità del fatto è differente da quella dello schema sorgente, può essere utile definire più misure che aggregano lo stesso attributo tramite operatori diversi.

### Esempio delle vendite:

Definizione delle **misure**:



Il **glossario** delle vendite potrebbero essere:

quantità venduta = SUM(VENDITA.quantità)  
incasso = SUM(VENDITA.quantità \* VENDITA.prezzoUnitario)  
num.clienti = COUNT(\*)

## CREAZIONE DELLO SCHEMA DI FATTO:

L'albero degli attributi può ora essere tradotto in uno schema di fatto che include le dimensioni e le misure definite:

- Le gerarchie corrispondono ai sottoalberi dell'albero degli attributi con radice nelle diverse dimensioni.
- Il nome del fatto corrisponde al nome dell'entità scelta come fatto.
- È possibile potare e innestare l'albero per eliminare dettagli inutili.
- È possibile aggiungere attributi dimensionali definendo opportuni intervalli per attributi numerici (ES. sulla dimensione tempo)

- Gli attributi che non verranno usati per l'aggregazione possono essere contrassegnati come descrittivi. Tra questi compariranno in genere anche gli attributi determinati da associazioni 1-a-1 e privi di discendenti.
- Per quanto riguarda eventuali **attributi alfanumerici** figli della radice ma non prescelti né come dimensioni né come misure:
  - Se la granularità degli eventi primari coincide con quella entità F, essi possono essere rappresentati come attributi descrittivi associati direttamente al fatto, di cui descriveranno ciascuna occorrenza;
  - Se invece le due granularità sono differenti, essi devono necessariamente essere potati.

#### Esempi di generazione finale dello schema di fatto:



#### MODELLAZIONE LOGICA:

Esistono due distinti modelli logici per rappresentare la struttura multidimensionale dei dati:

- Quello relazionale, che dà luogo ai sistemi **ROLAP** (Relation On-Line Analytical Processing);
- Quello multidimensionale, che dà luogo ai sistemi **MOLAP** (Multidimensional On-Line Analytical Processing)

La maggior parte del mercato è orientata ai sistemi ROLAP, a causa di un insieme di problemi relativi al sistema MOLAP.

#### SISTEMI MOLAP:

I sistemi MOLAP memorizzano i dati usando strutture dati multidimensionali, ad esempio vettori multidimensionali in cui ogni elemento è associato ad un insieme di coordinate nello spazio dei valori.

#### Vantaggi:

- Il tipo di struttura dati utilizzata è la rappresentazione più naturale per i dati di un DW;
- Fornisce ottime prestazioni poiché si presta bene alle esecuzioni delle operazioni OLAP, che sono esprimibili direttamente sulla struttura dati e non hanno bisogno di essere simulate attraverso interrogazioni SQL.

#### Svantaggi:

- Sparsità dei dati;
- Mancanza di standard, i diversi sistemi hanno in comune solo principi di base (ES. strutture dati), ma non ci conoscono i dettagli implementativi;
- Non esistono standard di interrogazione che svolgono un ruolo simile a quello di SQL nei sistemi relazionali.

#### SISTEMI MOLAP – PROBLEMA DELLA SPARSITÀ:

- Causa:** solo una piccola porzione delle celle di un cubo contiene effettivamente informazioni, le rimanenti corrispondono ad eventi non accaduti. Questa comporta un forte spreco in termini di spazio su disco e di tempo per caricare i dati nel cubo.
- Questo problema non incide sui sistemi ROLAP, poiché essi consentono di memorizzare solo le celle di interesse.

#### SISTEMI ROLAP:

Utilizzano il modello relazionale per la rappresentazione dei dati multidimensionali. I motivi che spingono all'adozione di un modello bidimensionale per modellare concetti multidimensionali sono i seguenti:

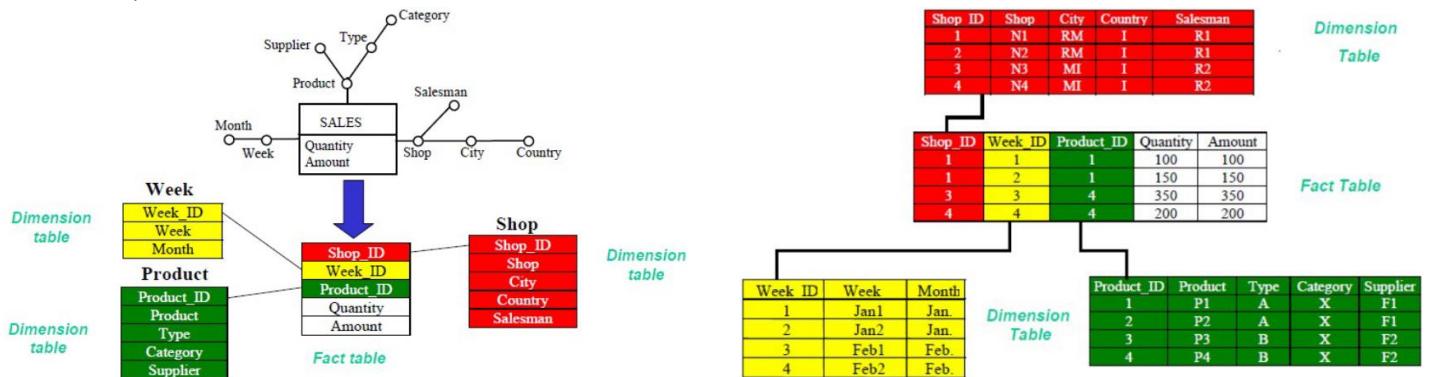
- Il modello relazionale è lo standard *“de facto”* dei database, ed è conosciuto dai professionisti del settore.
- L’evoluzione subita dai DBMS relazionali, da trent’anni sul mercato, li rende strumenti raffinati ed ottimizzati.
- L’assenza di sparsità dei dati garantisce maggiore scalabilità, fondamentale per database in continua crescita quali i DW.

Si usa per la modellazione multidimensionale su sistemi ROLAP. È composto da:

- Un insieme di relazioni DT<sub>1</sub>... DT<sub>n</sub>, chiamate dimension table, ciascuna associata ad una dimensione e caratterizzata da una chiave primaria ed un insieme di attributi che descrivono le dimensioni a vari livelli di aggregazione.
- Una relazione FT, chiamata fact table, la cui chiave primaria è data dall’insieme delle chiavi primarie delle dimension table.
- Inoltre, FT contiene un attributo per ogni misura.

#### Esempio Schema a Stella:

Schema a stella per il fatto delle vendite. La chiave della fact table SALES è costituita dalla combinazione delle chiavi esterne sulle tre dimension table.



## SISTEMA ROLAP – SCHEMA A STELLA:

In una relazione ROLAP i dati di un fatto multidimensionale sono organizzati con uno **schema a stella**, che è composta da:

- Relazione principale, detta *tavella dei fatti*, che memorizza le istanze di un fatto;
- Relazioni ausiliarie, dette *tabelle dimensione*, che memorizza i membri delle dimensioni associate al fatto;
- *Insieme di vincoli* di integrità referenziale, ognuna dei quali collega un attributo della tabella dei fatti a una tabella dimensione.

Lo schema a stella corrisponde all'implementazione relazionale di un fatto rappresentato da un cubo multidimensionale.

La visione multidimensionale dei dati si ottiene eseguendo il **join** tra le *fact table* e le diverse dimension table.

Per il fatto delle vendite riportato in precedenza, l'interrogazione SQL che ricostruisce le celle associando i valori delle misure ai corrispondenti valori degli attributi presenti nelle gerarchie è:

```
SELECT * FROM VENDITE AS FT, PRODOTTO AS DT1, NEGOZIO AS DT2, DATA AS DT3
WHERE FT.chiaveP=DT1.chiaveP AND FT.chiaveN=DT2.chiaveN AND FT.chiaveD=DT3.chiaveD;
```

Alla chiave di un **dimension table** si possono riferire più fact table, se le gerarchie sono conformi. Le dimension table non sono in 3NF, a causa della presenza di dipendenze funzionali transitive generate dalla presenza contemporanea di tutti gli attributi della gerarchia. La sparsità non rappresenta un problema, poiché nella fact table vengono memorizzate solo le combinazioni di chiavi per le quali effettivamente esiste l'informazione.

## SISTEMA ROLAP – SCHEMA SNOWFLAKE:

Uno **schema snowflake** è ottenibile da uno schema a stella decomponendo (o normalizzando) una o più **dimension table**  $DT_i$  in più tabelle  $DT_{i,1}, \dots, DT_{i,n}$ , al fine di eliminare alcune delle dipendenze funzionali transitive presenti. Ogni dimension table è caratterizzata da:

- Una chiave primaria  $d_{i,j}$  (di solito surrogata);
- Un sottoinsieme degli attributi di  $DT_i$  che dipendono funzionalmente da  $d_{i,j}$ ;
- Zero o più chiavi esterne riferite ad altre  $DT_{t,k}$  necessarie a garantire la ricostruibilità del contenuto informativo di  $DT_i$ .

Le dimension table primarie sono quelle in cui le chiavi sono importate nella fact table, secondarie le altre.

Un possibile snowflake per lo schema a stella delle vendite prevede l'inserimento delle tabelle CITTA' (City) e CATEGORIA (Type), ottenendo una parziale normalizzazione dei dati contenuti nelle dimension table.

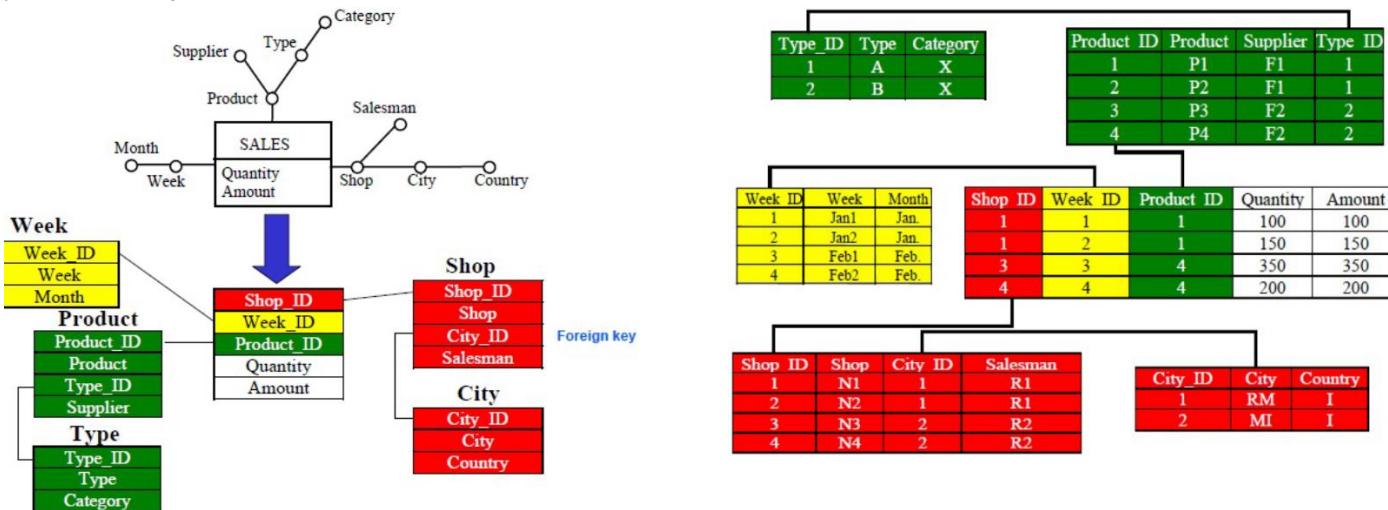
Vengono spezzate le dipendenze transitive tra negozio (Shop) e città (City), e tra prodotto (Product) e categoria (Type):

- Lo spazio richiesto per la memorizzazione dei dati si riduce. ES: le corrispondenze tra valori degli attributi città e regione vengono memorizzate una sola volta;
- Il tempo di esecuzione delle interrogazioni che coinvolgono attributi delle dimension table secondarie aumenta poiché è necessario un maggior numero di join.

## SISTEMI ROLAP – VANTAGGI SCHEMA SNOWFLAKE:

È necessario inserire nuove chiavi surrogate per determinare le corrispondenze tra dimension table primarie e secondarie. ES: l'importanza di Type\_ID nella tabella PRODOTTO permette di associare ad ogni prodotto il relativo tipo. L'esecuzione di interrogazioni che coinvolgono solo gli attributi contenuti nella fact table e nelle dimension table primarie è avvantaggiata poiché i join coinvolgono tabelle di dimensioni inferiori.

Esempio schema Snowflake:



Lo schema snowflake normalmente non è raccomandato:

- La diminuzione dello spazio di memorizzazione raramente è benefico. Maggiore spazio è consumato dalla fact table;
- Il costo della esecuzione del join potrebbe essere significativo;

Lo schema snowflake è utile:

- Quando parte di una gerarchia è condivisa tra le dimensioni (ES: gerarchie geografiche);
- Per le viste materializzate che richiedono una rappresentazione aggregata delle dimensioni corrispondenti.

## LE VISTE:

Le interrogazioni di un DW richiedono ripetutamente aggregazioni e sintesi laboriose, può essere conveniente valutare viste che esprimano i dati aggregati una volta per tutte, e memorizzarle. Questa tecnica prende il nome di **materializzazione delle viste**. Per esempio, nel data mart relativo alla gestione delle vendite, una vista può contenere i dati relativi alle vendite mensili di ciascun negozio. Tutte le interrogazioni interessa alle queste aggregazioni, verrebbero eseguite direttamente sulle viste, invece che sul data mart originario.

**Problema:** Analisi utente rese difficili dalla quantità di dati memorizzati nel DW.

**Soluzione:** Ridurre la porzione da esaminare attraverso operazioni di:

- **Selezione:** restringono la porzione di dati di interesse individuano quelli effettivamente interessanti per la specifica analisi;
- **Aggregazione:** riducono i dati collassando più elementi non aggregati in un unico elemento aggregato.

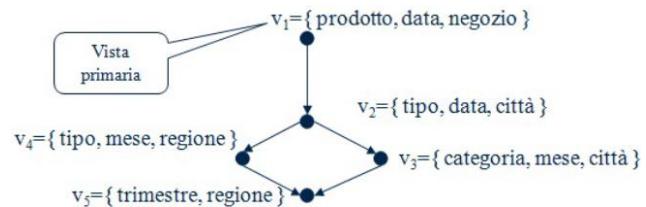
L'aumento delle prestazioni è ottenuto precalcolando i dati aggregati di uso comune.

Le fact table contenenti dati aggregati sono dette **viste**:

- **Viste primarie**: corrispondono a pattern primari (definito dall'insieme delle dimensioni);
- **Viste secondarie**: corrispondono a pattern secondari o sono individuate verificando se possono essere alimentate a partire da viste nel DW (aggregati).

Sono rappresentate alcune viste materializzabili per lo schema a stella delle vendite:

- Una freccia da  $v_i$  a  $v_j$  indica che  $P_j \leq P_i$  essendo  $P_i$  e  $P_j$  rispettivamente i pattern di  $v_i$  e  $v_j$ . Quindi i dati contenuti in  $v_j$  possono essere calcolati aggregando quelli di  $v_i$ ;
- Un'interrogazione relativa alle vendite che richieda i dati aggregati per il tipo del prodotto, data di vendita e città (i.e., {tipo, data, città}) in cui la vendita è stata effettuata risulterà meno costosa se eseguita su  $v_2$  poiché essa insisterà su una fact table piccola e non richiederà ulteriori operazioni di aggregazione.



### PROBLEMI CON LE VISTE:

Ciascuna vista dipende da un insieme di tabelle di base, quando queste subiscono modifiche è necessario aggiornare le viste, propagando gli effetti delle modifiche sulle tabelle di base. Ne consegue che la materializzazione è conveniente in un ambiente nel quale le tabelle di base non subiscono modifiche frequenti.

**Problema:** calcolando l'incasso delle vendite a partire dalla tabella aggregata si ottiene un dato diverso da quello ottenuto dalla tabella non aggregata.

**Causa:** Applicando l'operatore di media si perde l'informazione relativa ai singoli prezzi praticati.

**Soluzione:** Memorizzare nella tabella aggregata anche i valori degli incassi.

tipo	prodotto	quantità	prezzo	incasso
latticino	Latte Slurp	5	1,0	5,0
latticino	Latte Gnam	7	1,5	10,5
bibita	Colissima	9	0,8	7,2

totale: 22,7

tipo	quantità	prezzo	Quantità x prezzo
latticino	12	1.25	15,0
bibita	9	0,8	7,2

totale: 22,2

### PROGETTAZIONE LOGICA:

La progettazione logica definisce l'insieme dei passi per trasformare lo schema concettuale in uno schema logico. La progettazione logica dei Data mart è profondamente diversa dai sistemi operazionali. Infatti, nei DW l'obiettivo è quello di massimizzare la velocità del reperimento dei dati mentre nei sistemi operazionali si mira a minimizzare la quantità di informazione da memorizzare. I principali passi di questo processo sono:

- **Traduzione degli schemi di fatto in schemi logici;**
- **Materializzazione delle viste;**
- **Frammentazione verticale ed orizzontale delle fact table.**

### TRADUZIONE DEGLI SCHEMI DI FATTO IN SCHEMI LOGICI:

Uno schema di fatto può essere modellato in ambito relazionale mediante uno schema a stella in cui la fact table contiene tutte le misure e gli attributi descrittivi e, per ogni gerarchia, viene creata una dimension table che ne contiene tutti gli attributi. La traduzione dal DFM al modello logico non è del tutto automatica, richiedendo in alcuni momento l'intervento del progettista. Una corretta traduzione di uno schema di fatto richiede una trattazione più approfondita per i costrutti avanzati del DFM.

### COSTRUTTI AVANZATI - ATTRIBUTI DESCRIPTIVI:

Sappiamo che un attributo descrittivo contiene informazioni non utilizzabili per effettuare aggregazioni che si ritiene utile mantenere.

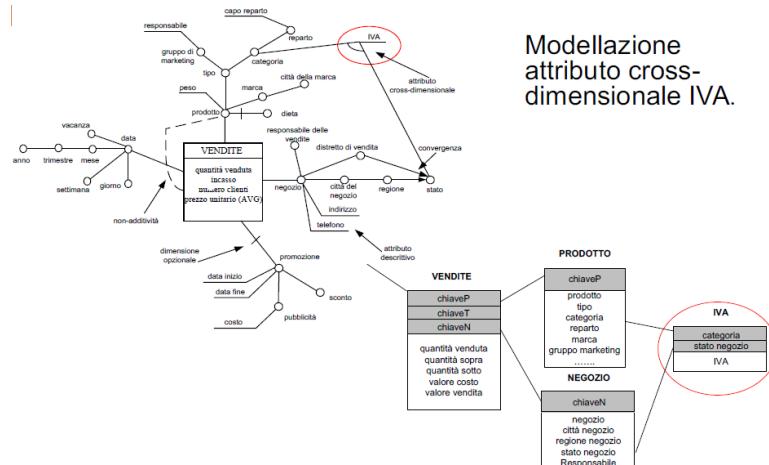
Durante la modellazione un attributo descrittivo:

- Viene incluso nella dimension table relativa alla gerarchia che lo contiene se collegato ad un attributo dimensionale da cui dipende funzionalmente.
- Viene incluso nella fact table assieme alle misure collegate direttamente al fatto.

### COSTRUTTI AVANZATI – ATTRIBUTI CROSS-DIMENSIONALI:

Sappiamo che un attributo cross-dimensionale è tale se il suo valore è determinato dalla combinazione di due o più attributi dimensionali eventualmente appartenenti a gerarchie diverse.

Se un attributo cross-dimensionale b definisce un'associazione multi-a-molti tra due o più attributi dimensionali  $a_1, \dots, a_n$ , esso richiede l'inserimento di una nuova tabella che includa b ed abbia come chiave gli attributi  $a_1, \dots, a_n$ .



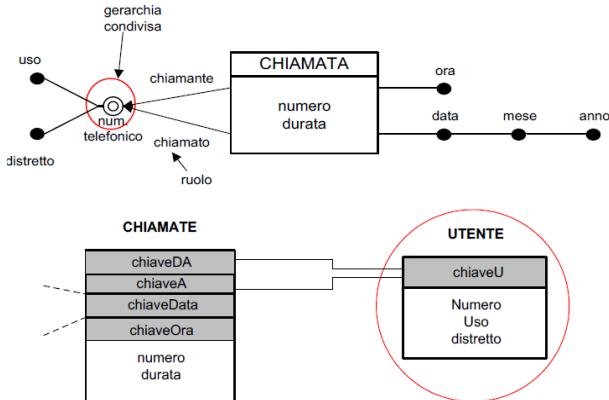
### COSTRUTTI AVANZATI – GERARCHIE CONDIVISE:

Sappiamo che una gerarchia condivisa è una porzione di gerarchia che viene ripetuta due o più volte. A livello logico non è consigliabile introdurre più dimension table che contengano gli stessi dati. A livello progettuale esistono due soluzioni per due situazioni distinte:

- Se due gerarchie contengono esattamente gli stessi attributi è sufficiente importare due valori diversi dell'unica chiave della dimension table nella fact table.
- Se due gerarchie condividono una parte degli attributi si può scegliere di replicare le informazioni comuni o di introdurre una nuova dimension table comune ad entrambe le gerarchie.

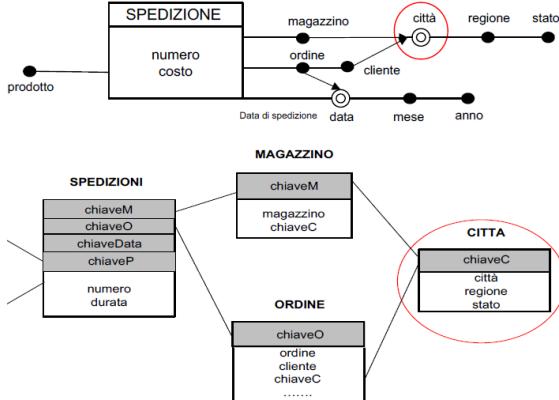
## CONDIVISIONE TOTALE

Modellazione numero telefonico chiamante e chiamato.



## CONDIVISIONE PARZIALE

Modellazione città del magazzino e del cliente.



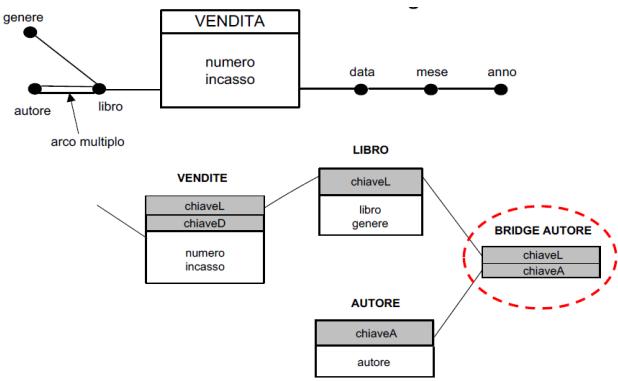
## COSTRUTTI AVANZATI – ARCHI MULTIPLI:

Ricordiamo che una gerarchia che codifica associazioni molti-a-molti viene modellata con archi multipli. A livello logico esistono diverse soluzioni:

- **Soluzione bridge table:** utilizzo di una nuova tabella la cui chiave è composta dalla combinazione degli attributi collegati dall'arco multiplo (schema snowflake);
- **Soluzione push-down:** l'associazione molti-a-molti viene modellata direttamente all'interno della fact table. Viene poi aggiunta una nuova dimensione corrispondente all'attributo terminale a dell'arco multiplo, ed eventuali figli di a verranno memorizzati nella nuova dimension table (schema a stella).

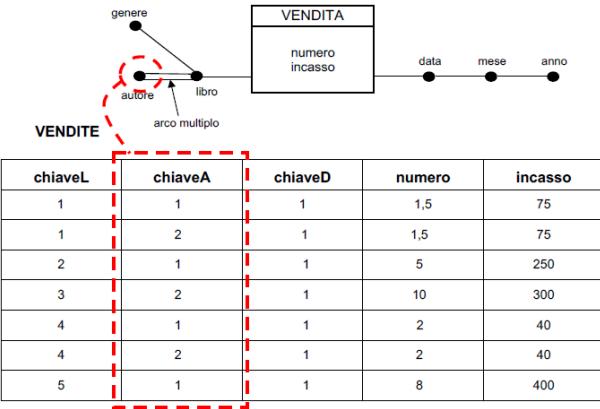
### BRIDGE TABLE

Modellazioni di Autore tramite bridge table.



### PUSH-DOWN

Una possibile istanza della fact table Vendite avendo effettuato il push-down di Autore.



La soluzione push-down introduce una forte ridondanza nella fact table le cui righe devono essere replicate tante volte quante sono le corrispondenze dell'arco multiplo. La forte ridondanza causa operazioni di aggiornamento molto costose che non si verificano nella soluzione bridge table. Operazioni di interrogazioni nelle soluzioni push-down prevedono un singolo join mentre con bridge table risultano più complesse. Il calcolo degli eventi primari avviene durante l'alimentazione nella soluzione push-down mentre nella soluzione con bridge table avviene durante l'interrogazione.

## COSTRUTTI AVANZATI – ARCHI OPZIONALI:

Ricordiamo che un arco opzionale si riferisce ad una gerarchia opzionale, in cui un'associazione dello schema di fatto non è definita per un sottoinsieme di eventi. La presenza di archi opzionali non incide sulla struttura della corrispondente dimension table:

- L'attributo continua a comparire anche se per alcune istanze non risulterà valorizzato;
- Per le istanze in cui non è definito tale valore si introduce un valore fittizio.

L'opzionalità non può essere gestita direttamente nella fact table introducendo un valore fittizio per la chiave. Bisognerà introdurre un'intera tupla fittizia all'interno della dimension table.

## COSTRUTTI AVANZATI – GERARCHIE RICORSIVE:

Ricordiamo che una gerarchia ricorsiva è una gerarchia in cui le relazioni padre-figlio tra i livelli sono consistenti ma possono avere istanze di lunghezza differenti. La modellazione delle gerarchie ricorsive può essere effettuata in due modi:

- Nelle dimension table la ricorsione è modellata con un auto anello che rappresenta un numero variabile, e potenzialmente illimitato, di livelli. Tale modellazione non è supportata dalla maggior parte dei DBMS commerciali.
- Con una **tavola di navigazione** che modella un'associazione molti a molti tra le fact table e la dimension table. La dimensione di tale tabella cresce in maniera esponenziale rispetto alla profondità della gerarchia ma tale modellazione ha un maggiore potere espressivo in fase di interrogazione.

## LO SCHEMA LOGICO RELAZIONALE:

Schema logico relazionale di vendita:

PRODOTTO (prodotto, peso, dieta, marca: MARCA, tipo: TIPO)

MARCA (marca, prodottaIn:CITTA)

CITTA (città, regione:REGIONE)

REGIONE (regione, stato:STATO)

STATO (stato)

TIPO (tipo, gruppoMarketing:GRUPPOMARK, categoria:CATEGORIA)

GRUPPOMARK (gruppoMarketing, responsabile)

CATEGORIA (categoria, reparto:REPARTO)

REPARTO (reparto, capoReparto)

IVA (categoria:CATEGORIA, stato:STATO, iva)

NEGOZIO (negozi, indirizzo, telefono, respVendite, (numDistr, stato):DISTRETTO, inCitta:CITTA)

DISTRETTO (numDistr, stato:STATO)

DATA (data, giorno, vacanza, settimana, mese:MESE)

MESE (mese, trimestre:TRIMESTRE)

TRIMESTRE (trimestre, anno:ANNO)

ANNO (anno)

PROMOZIONE (promozione, dataInizio, dataFine, sconto, pubblicità:PUBBLICITA)

PUBBLICITA (pubblicità, costo)

VENDITA (prodotto:PRODOTTO, negozi:NEGOZIO, data:DATA, promozione:PROMOZIONE, quantità, prezzoUnitario)

Gli attributi facenti parte di chiavi esterne composte sono indicati tra parentesi

## MATERIALIZZAZIONE DELLE VISTE:

Con il termine materializzazione delle viste si intende il processo di selezione di un insieme di viste secondarie ottenute a partire dai dati contenuti nelle viste primarie. La scelta delle viste da materializzare deve essere fatta sulla base di un insieme di obiettivi di progetto.

Due sono gli elementi principali nel processo di materializzazione:

- La definizione degli obiettivi della materializzazione, che possono essere funzioni di minimizzazione di costo o vincoli.
- La tecnica di selezione da utilizzare.

## FUNZIONI DI COSTO DA MINIMIZZARE:

Tipicamente le funzioni di costo che possono essere minimizzate sono:

- **Costo del carico di lavoro:** rappresenta il costo totale del carico di lavoro che può essere calcolato come somma pesata del costo delle diverse interrogazioni, dove il peso di ogni singola interrogazione può essere la frequenza e/o la sua importanza per l'utente.
- **Costo di manutenzione delle viste:** rappresenta il costo delle interrogazioni necessarie a propagare gli aggiornamenti delle sorgenti operazionali alle viste.

## VINCOLI:

**Vincoli di sistema:** sono dettati dalla limitatezza delle risorse disponibili e riguardano:

- **Spazio di memorizzazione:** per calcolare questo vincolo è necessaria una funzione per la stima della dimensione delle viste;
- **Tempo di aggiornamento:** il tempo di aggiornamento delle viste.

**Vincoli utente:** sono legati a particolari requisiti espressi dagli utilizzatori del sistema:

- **Tempo di risposta alle interrogazioni:** tempo di massimo di risposta richiesto dall'utente per le diverse interrogazioni;
- **Data di aggiornamento delle risposte:** limite massimo imposto dall'utente per il tempo intercorso dall'ultimo aggiornamento di una vista impiegata per l'esecuzione di un'interrogazione.

## TECNICHE DI SELEZIONE:

Le tecniche di selezione operano su due diverse fasi:

- Tra tutte le possibili viste materializzabili, vengono individuate quelle effettivamente utili per il carico di lavoro.
- Successivamente, tramite tecniche euristiche, se ne determina un sottoinsieme che minimizza la funzione di costo nel rispetto dei vincoli di sistema.

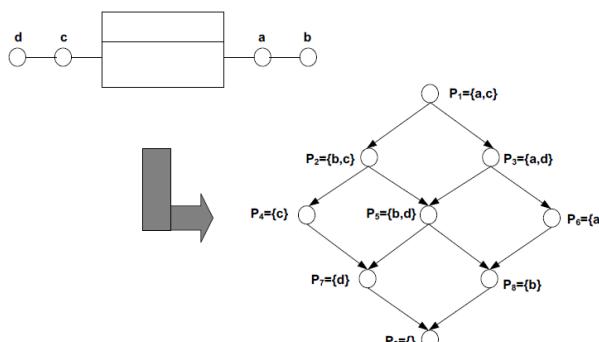
## RETIKOLO MULTIDIMENSIONALE:

Il **reticolo multidimensionale** o MD-lattice viene utilizzato per individuare tutte le possibili viste materializzabili a partire da uno schema di fatto, definendo tutti i possibili pattern di aggregazione validi. Un pattern è valido se non esistono dipendenze funzionali tra i suoi elementi.

**Nota:** Un arco del reticolo da un pattern  $P_i$  ad un pattern  $P_j$  indica che  $P_j$  è meno fine di  $P_i$  ( $P_j \leq P_i$ ).

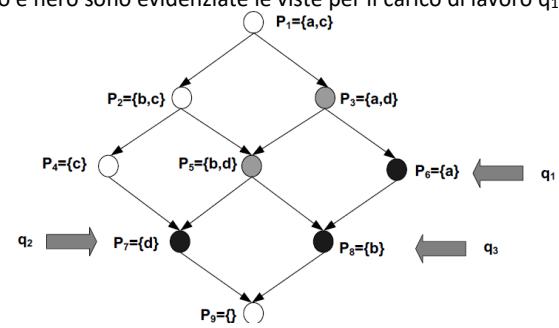
### RETIKOLO MULTIDIMENSIONALE

Reticolo corrispondente al cubo multidimensionale con dimensioni  $a, c$  e gerarchie  $a \rightarrow b \wedge c \rightarrow d$ .



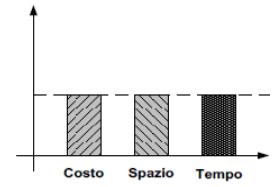
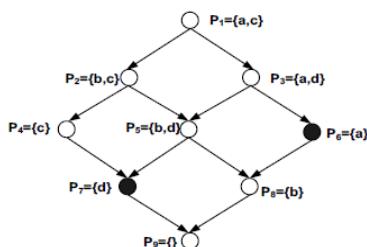
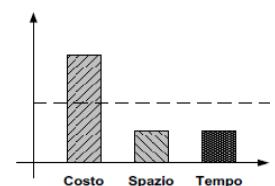
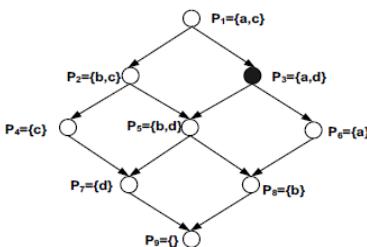
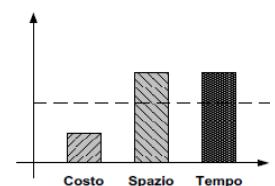
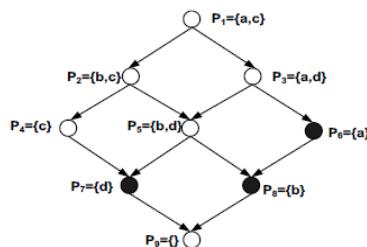
### VISTE CANDIDATE

In grigio e nero sono evidenziate le viste per il carico di lavoro  $q_1, q_2, q_3$ .



## SCELTA DELLE VISTE DA MATERIALIZZARE

Tre possibili soluzioni al problema della materializzazione delle viste.



### LINEE GUIDA PER SELEZIONARE LE VISTE:

È consigliabile materializzare una vista quando:

- Risolve direttamente un'interrogazione molto frequente;
- Permette di risolvere molte interrogazioni.

Non è consigliabile materializzare una vista quando:

- Il suo pattern è molto simile ad una vista già materializzata;
- Il suo pattern è molto fine;
- La materializzazione non riduce di almeno un ordine di grandezza il carico di lavoro.