

Basi di dati

Sistema Informativo: È l'insieme delle attività umane e dei dispositivi di memorizzazione ed elaborazione che organizza e gestisce l'informazione di interesse per un'organizzazione di dimensioni qualsiasi (NON contiene necessariamente tecnologia informatica).

Base di dati: È una collezione di dati utilizzati per rappresentare con tecnologia informatica le informazioni di interesse per un sistema informativo.

- **Schema:** è la descrizione della struttura e delle proprietà di una specifica base di dati fatta utilizzando i costrutti del modello dei dati (lo schema di una base di dati è invariante nel tempo).
- **Istanza:** è costituita dai valori effettivi che in un certo istante popolano le strutture dati della base di dati (l'istanza di una base di dati varia nel tempo).

DBMS: È un sistema che gestisce su memoria secondaria collezioni di dati (chiamate "Basi di Dati"):

- Grandi, Condivise e Persistenti.

Garantendo:

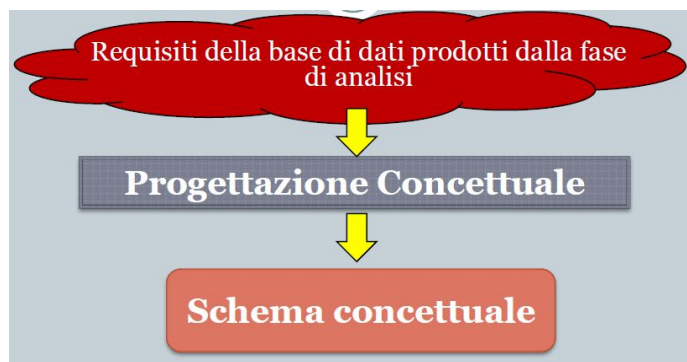
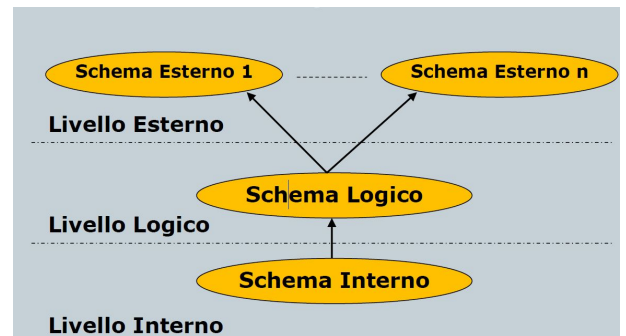
- Affidabilità, Privatezza e Accesso Efficiente.

Modello Dei Dati: È l'insieme dei costrutti forniti dal DBMS per descrivere la struttura e le proprietà dell'informazione contenuta in una base di dati.

- Permettono di definire le strutture dati che conterranno le informazioni della base di dati (analogia con i costruttori di tipo di un linguaggio di programmazione).
- Permettono di specificare le proprietà che dovranno soddisfare le istanze di informazione che saranno contenute nelle strutture dati.

Architettura di un DBMS:

- Schema **Logico**: è la rappresentazione della struttura e delle proprietà della base di dati attraverso i costrutti del modello dei dati del DBMS.
- Schema **Interno**: è la rappresentazione della base di dati per mezzo delle strutture fisiche di memorizzazione (file dati, file indice, etc).
- Schema **Esterno**: descrive una porzione dello schema logico di interesse per uno specifico utente o applicazione (attraverso viste sullo schema logico).



Progettazione concettuale: rappresentare il contenuto informativo della base di dati in modo formale ma indipendente dall'implementazione (quindi indipendente dalla scelta del DBMS) e dalle operazioni.

Progettazione logica: tradurre lo schema concettuale nello schema logico aderente al modello dei dati del DBMS scelto per l'implementazione. Nella traduzione si tiene conto delle operazioni più frequenti che le applicazione eseguiranno sulla base di dati.

Progettazione fisica: completare lo schema logico con i parametri relativi alla memorizzazione fisica dei dati e con gli opportuni metodi di accesso (INDICI) per garantire un accesso efficiente ai dati.

Modello Entità-Relazione (ER)

Entità (o tipo di entità): un'entità E rappresenta una classe di oggetti con le seguenti caratteristiche:

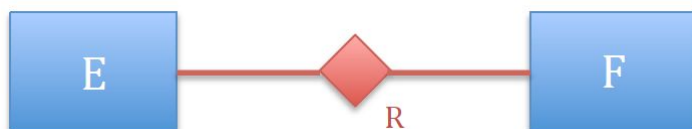
- Hanno proprietà **comuni**.
- Hanno esistenza **autonoma** (rispetto ad altre classi di oggetti).
- Hanno **identificazione univoca** (chiara corrispondenza con i concetti istanziati nel sistema informativo).



Istanza (o occorrenza): un'istanza dell'entità E è un **oggetto appartenente alla classe rappresentata da E**. Si indica con $I(E)$ l'insieme delle istanze di E che esistono nella base di dati in un certo istante.

Relazione (o tipo di relazione): una relazione R rappresenta un **legame logico tra due o più entità**.

Possono esistere **relazioni binarie** (quando due entità diverse vengono coinvolte nella relazione), **ternarie** (tre entità coinvolte), **ennarie** (n entità coinvolte).



Caso particolare: relazione binaria sulla stessa entità (**relazione ricorsiva**).

Vale la seguente proprietà per le istanze di una relazione $R(I(R))$:

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

→ non è possibile rappresentare la stessa ennupla più volte!



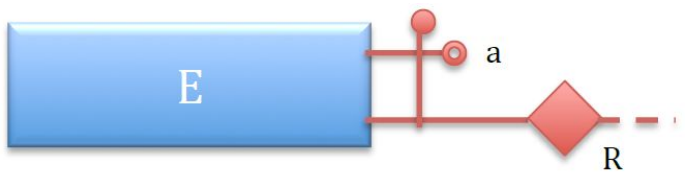
Istanza (o occorrenza): data una **relazione R tra n entità** E_1, \dots, E_n , un'istanza della relazione R è una **ennupla di istanze di entità**: (e_1, \dots, e_n) , dove $e_i \in I(E_i)$.

Attributo (di entità o relazione): rappresenta una proprietà elementare di un'entità (o relazione). Ogni attributo di entità (o relazione) associa ad ogni istanza di entità (o relazione) UNO e UN SOL valore appartenente ad un dominio (insieme di VALORI AMMISSIBILI).

Attributo a dell'entità E	Attributo b della relazione R

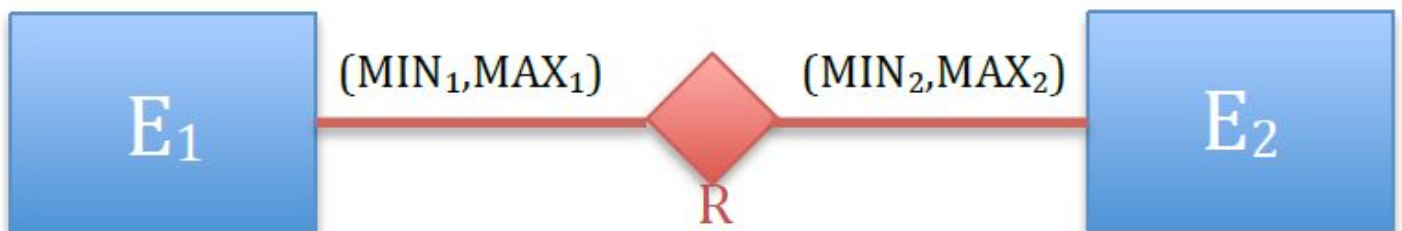
Istanza (o occorrenza): un'istanza dell'attributo **a** è un **valore del dominio di a**.

Identificatore (di una entità E): è un **insieme di proprietà** (attributi e/o relazioni) **che identificano univocamente le istanze dell'entità** (un insieme di proprietà IDENTIFICA UNIVOCAMENTE le istanze di E se non esistono due istanze di E che presentano gli stessi valori nelle proprietà dell'insieme).

Entità E con identificatore a	
Entità E con identificatore a, b	
Entità E con identificatore a, R	

Se l'identificatore contiene **solo attributi** allora si dice **INTERNO**, se contiene **almeno una relazione** si dice allora **ESTERNO**.

Vincoli di cardinalità: data una relazione R i vincoli di cardinalità **vengono specificati per ogni entità E_i coinvolta nella relazione R e specificano il numero minimo e il numero massimo di occorrenze** di a cui una istanza di E_i deve/può partecipare.



Valori possibili per **MIN_i**:

- **0:** indica che la partecipazione alla relazione R delle istanze di E_i è **OPZIONALE**.
- **1:** indica che la partecipazione alla relazione R delle istanze di E_i è **OBBLIGATORIA**.
- **>1:** indica che per ogni istanze di E_i devono essere presenti **almeno <num> occorrenze** della relazione R che la coinvolgono.

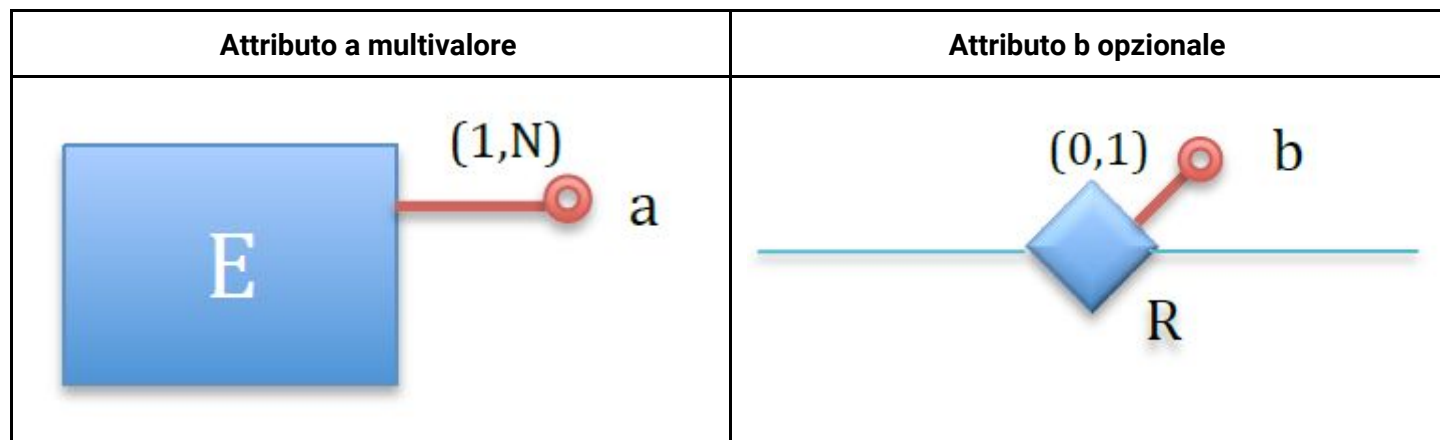
Valori possibili per **MAX_i**:

- **1:** indica che **un'istanza** di E_i può al massimo partecipare a **una sola occorrenza** della relazione R (se R è binaria questo indica che R è una FUNZIONE).
- **N:** indica che **un'istanza** di E_i può partecipare a **più occorrenze** della relazione R senza limite massimo.
- **>1:** indica che per **ogni istanze** di E_i devono essere presenti **al più <num> occorrenze** della relazione R che la coinvolgono.

Attributo opzionale e/o multivalore (di entità o relazione): si ottengono dagli attributi normali specificano un vincolo cardinalità sui **valori che l'attributo può assumere** (il default è (1,1)).

Valori possibili per la cardinalità di un attributo:

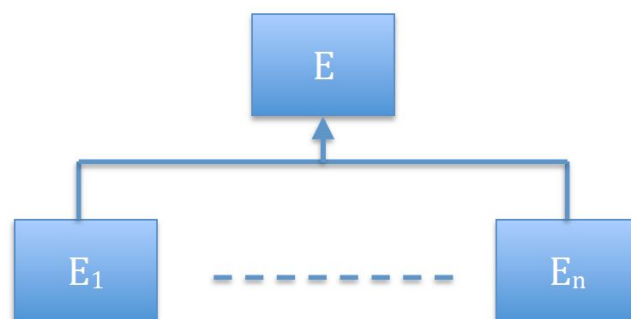
- **(0,1)**: attributo opzionale.
- **(1,N)**: attributo multivalore.
- **(0,N)**: attributo opzionale e multivalore.



Generalizzazione: è un **legame logico** (simile a una ereditarietà tra classi) **tra** un'entità **padre E e n** (con $n > 0$) **entità figlie** E_1, \dots, E_n , dove E rappresenta una classe di oggetti più generale rispetto alle classi di oggetti rappresentate dalle entità E_1, \dots, E_n .

Proprietà delle istanze di entità che partecipano ad una generalizzazione:

- Ogni istanza di un'entità figlia E_i è anche istanza dell'entità padre E.
- Ogni proprietà (attributi, identificatori e relazioni) dell'entità padre E è anche proprietà di ogni istanza delle entità figlie E_1, \dots, E_n .



Classificazione delle generalizzazioni:

- Una generalizzazione si dice **totale** se ogni istanza dell'entità padre E è anche istanza di almeno un'entità figlia E_i ; altrimenti si dice **parziale**.
- Una generalizzazione si dice **esclusiva** se ogni istanza dell'entità padre E è istanza al più di un'entità figlia E_i ; altrimenti si dice **sovrapposta**.

Il **tipo di generalizzazione** si indica con la stenografia seguente:

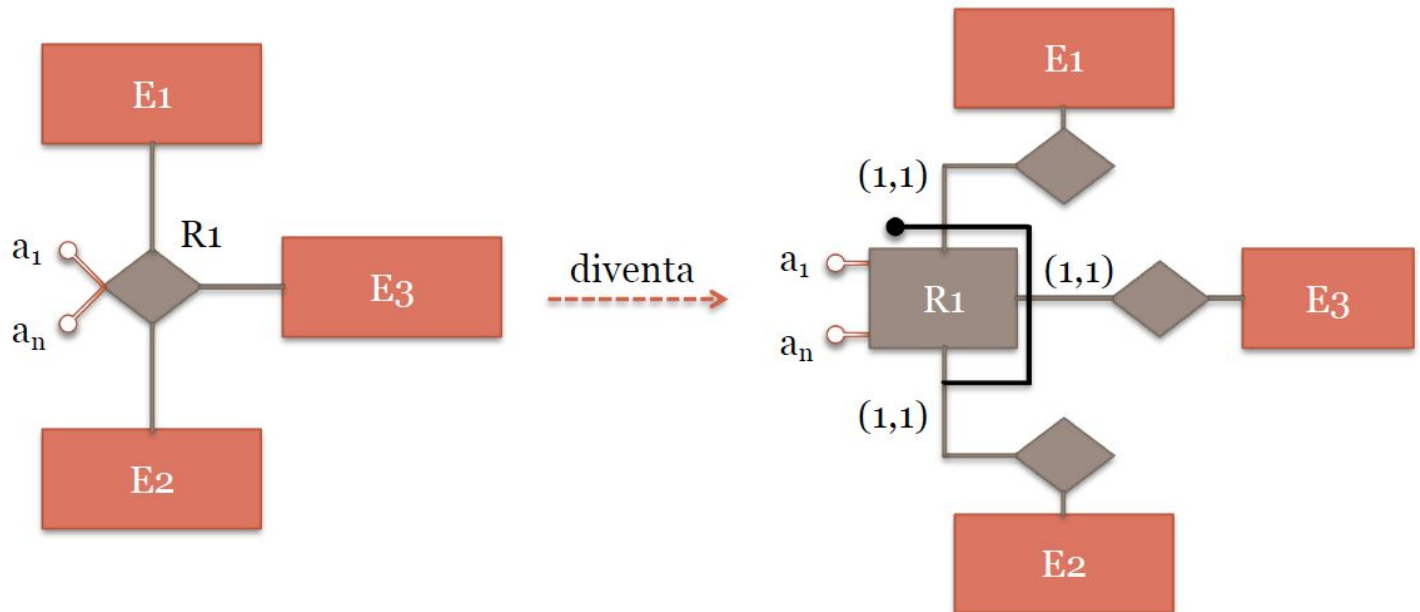
- totale ed esclusiva: **(t,e)**.
- totale e sovrapposta: **(t,s)**.
- parziale e esclusiva: **(p,e)**.
- parziale e sovrapposta: **(p,s)**.

posta **vicino alla freccia** che indica la generalizzazione.

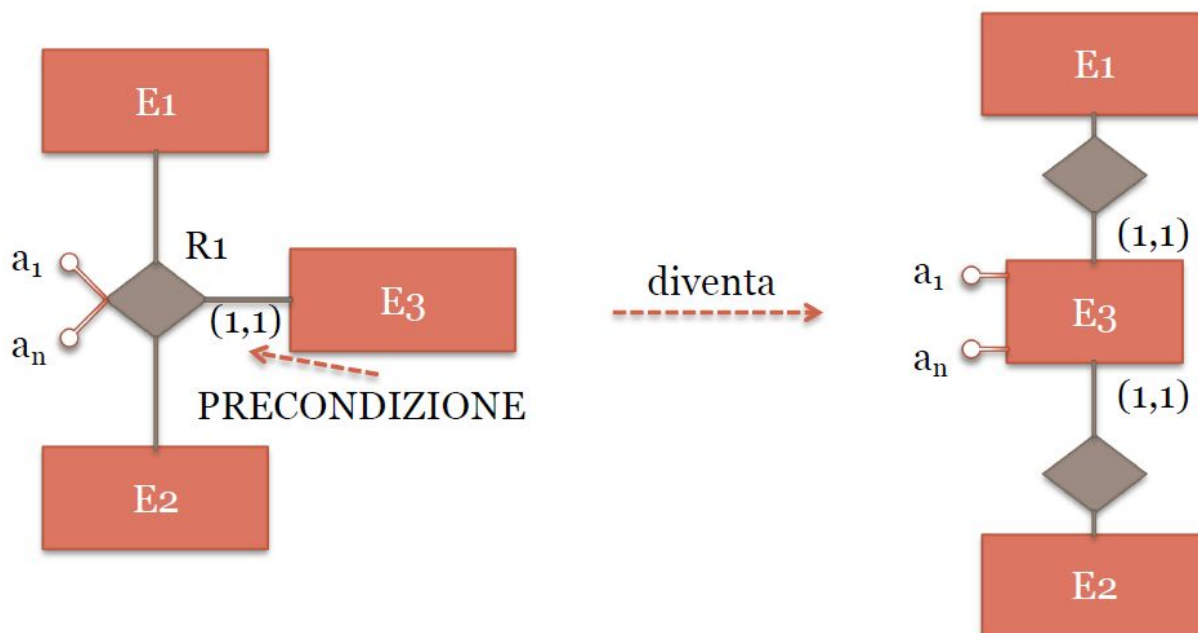
Semplificazioni dello schema

Eliminazione delle relazioni ternarie:

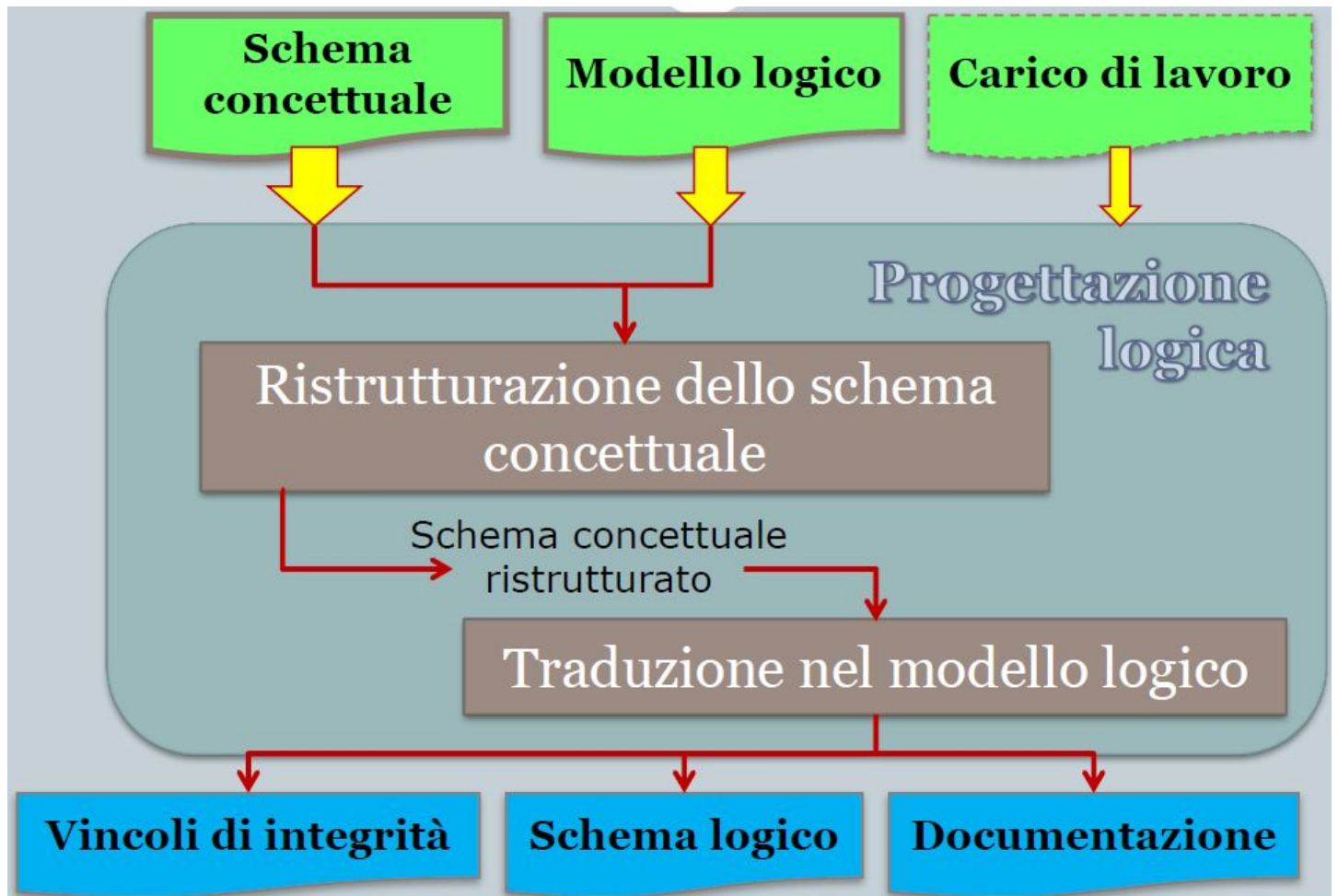
- Trasformazione di una relazione ternaria in una entità:



- Riduzione di una relazione ternaria a due relazioni binarie:



Progettazione logica



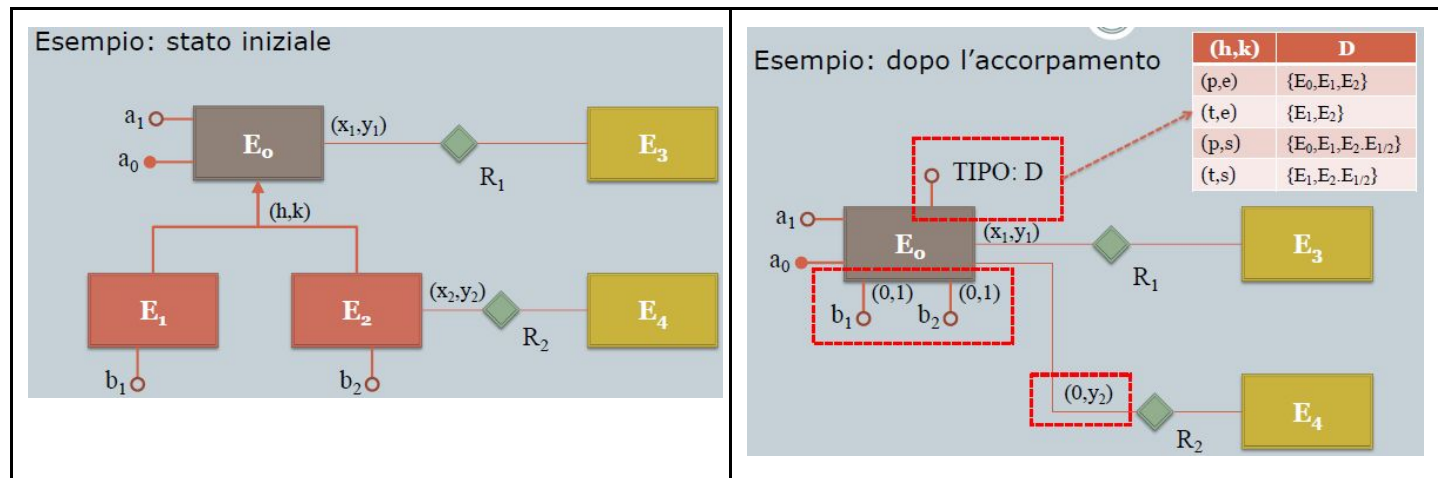
Motivazioni della **ristrutturazione**:

- **Semplificare** la fase successiva di **traduzione**.
- **Ottimizzare** le strutture rispetto al **carico di lavoro**.
- Sottofasi:
 - a. Analisi delle ridondanze dovute a presenza di dati derivabili.
 - Presenza di dati derivabili da altri dati.
 - Analizzare i dati derivabili presenti nello schema, e decidere se mantenere il dato derivabile, memorizzandolo ($COSTO_Calcolo > COSTO_Aggiornamento$), o calcolarlo quando serve ($COSTO_Calcolo < COSTO_Aggiornamento$);
 - b. Eliminazione delle generalizzazioni.
 - c. Accorpamento/partizionamento di entità e relazioni.
 - d. Scelta degli identificatori principali.

Eliminazione delle generalizzazioni:

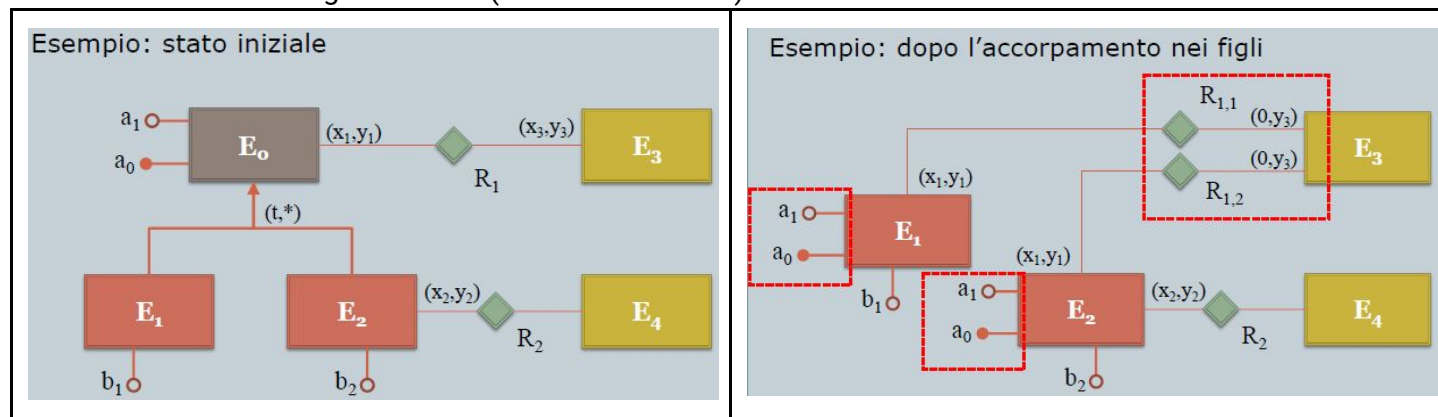
1. Accorpamento delle entità figlie nel padre:

- Eliminazione delle entità figlie.
- Accorpamento nel padre di tutti gli attributi specifici delle entità figlie come attributi opzionali.
- Accorpamento nel padre delle relazioni che coinvolgono le entità figlie assegnando cardinalità minima uguale a zero (lato entità padre).
- Aggiunta di un attributo TIPO per distinguere nel padre le occorrenze delle entità figlie eliminate.



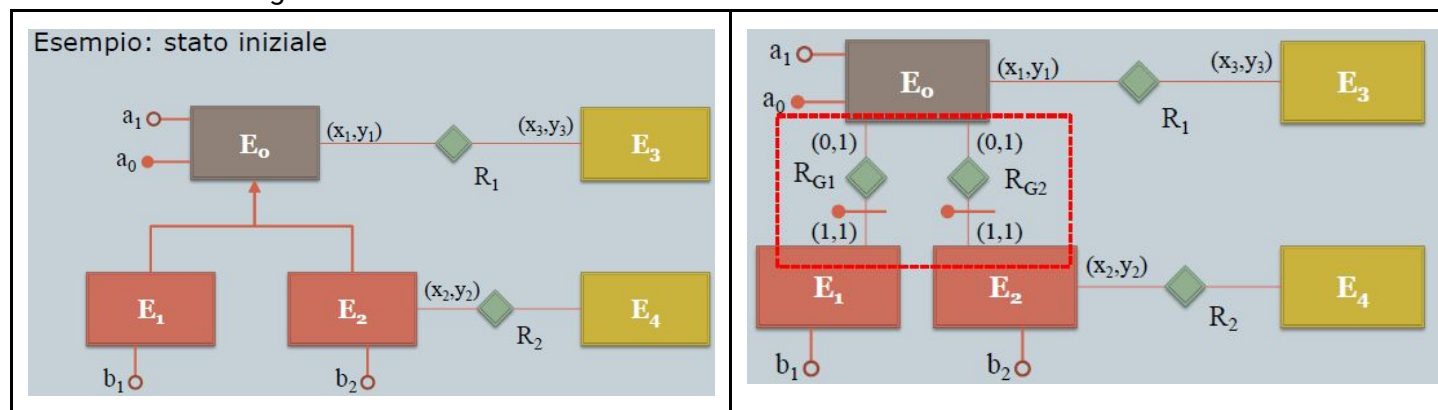
2. Accorpamento nei figli:

- Eliminazione dell'entità padre.
- Replicazione degli attributi dell'entità padre su ogni entità figlia.
- Partizionamento sui figli delle relazioni che coinvolgono l'entità padre assegnando cardinalità minima uguale a zero (lato entità esterne).



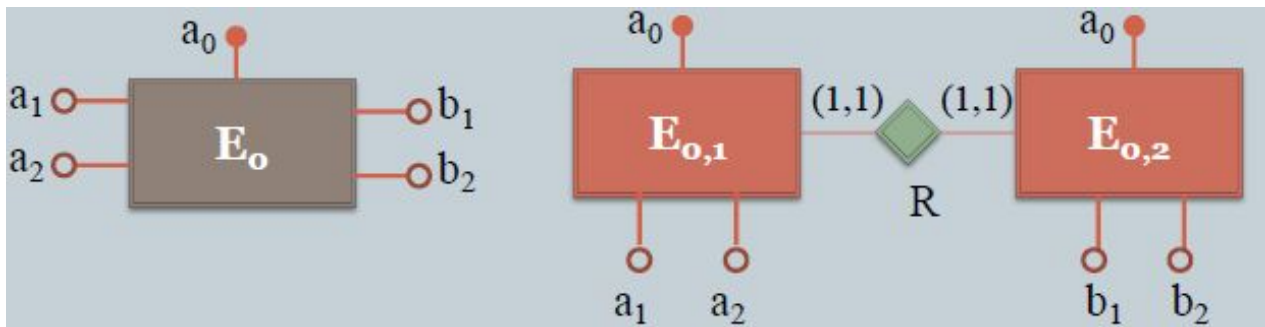
3. Sostituzione con relazioni:

- Eliminazione della generalizzazione.
- Inserire n relazioni tra il padre e ciascuna delle n entità figlie.
- Tutti gli attributi conservano la loro collocazione.

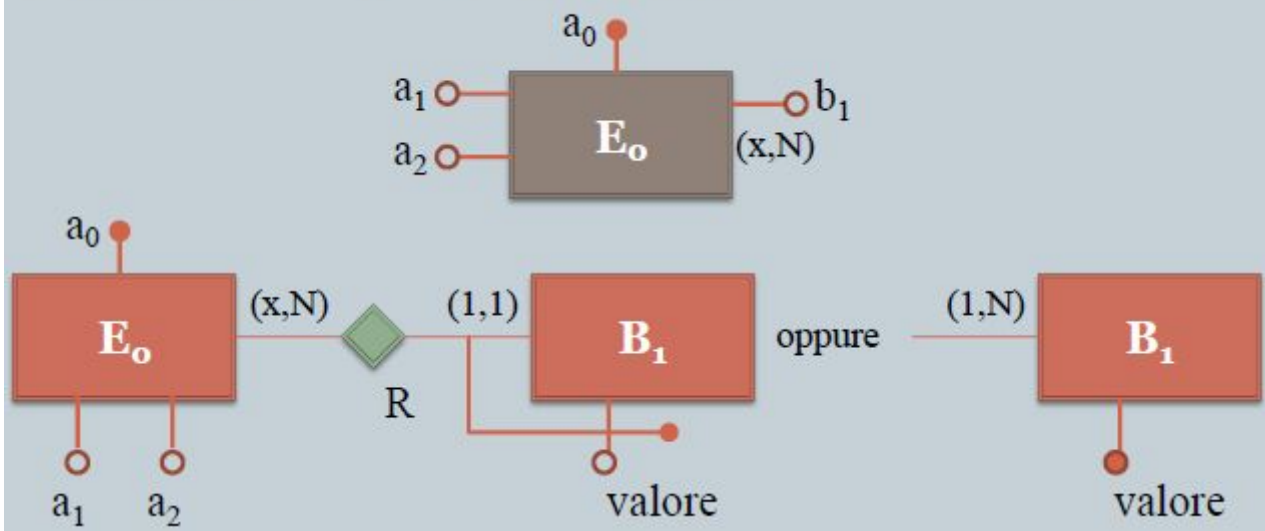


Partizionamento di entità

Sostituire un'entità dello schema con una coppia di entità con lo stesso identificatore e una relazione uno a uno che le lega tra loro. Si esegue solo nel caso in cui esistano operazioni frequenti che trattano solo un sottoinsieme degli attributi dell'entità da partizionare.



Eliminazione degli attributi multivalore



Partizionamento di relazioni

Si realizza **sostituendo una relazione R con due relazioni distinte R1 e R2**, dove le **occorrenze di R si partizionano tra R1 e R2**. E' conveniente quando esistono operazioni che si riferiscono ad un sottoinsieme delle occorrenze di R e altre operazioni che si riferiscono alle altre occorrenze di R.

Scelta degli identificatori principali

Va scelto un **identificatore principale** per **ogni entità** in quanto tale identificatore verrà utilizzato nel modello relazionale per identificare le tuple che rappresentano istanze di entità.

Criteri:

- Gli **identificatori** che includono **attributi opzionali NON possono essere chiavi primarie**.
- **Meglio identificatori con pochi attributi**.
- **Meglio identificatori utilizzati nelle operazioni**.

Traduzione verso il modello relazionale

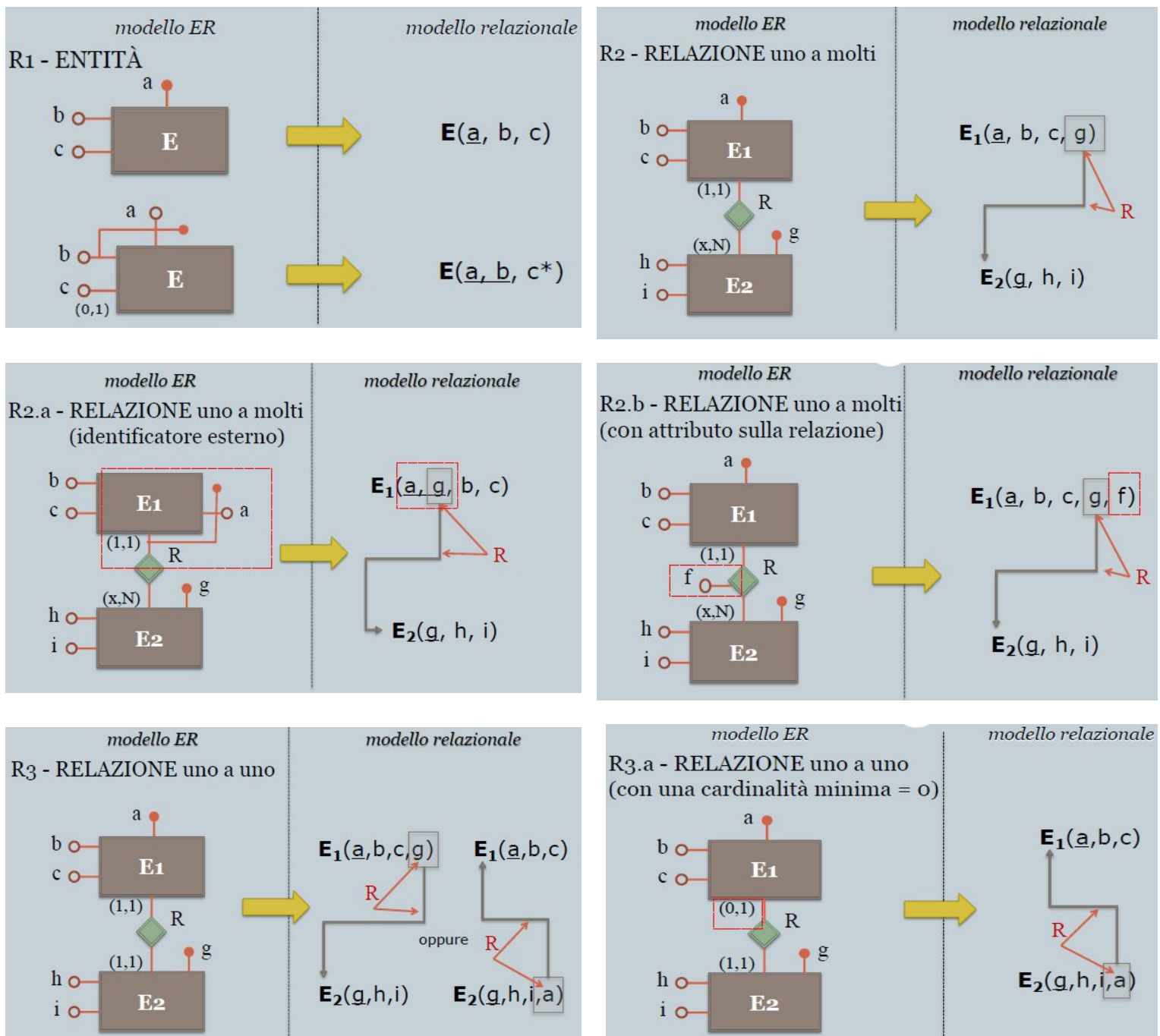
La traduzione è un **processo automatico** che si realizza **applicando** allo schema concettuale ristrutturato un insieme di **regole di traduzione**. Ogni regola si applica ad un costrutto del modello concettuale ER e produce una o più strutture del modello relazionale.

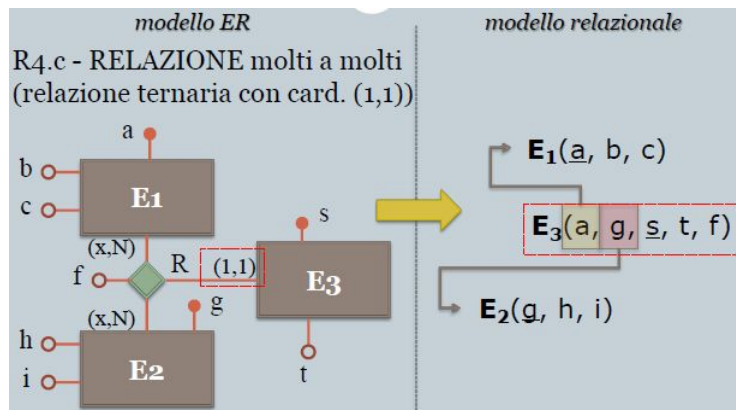
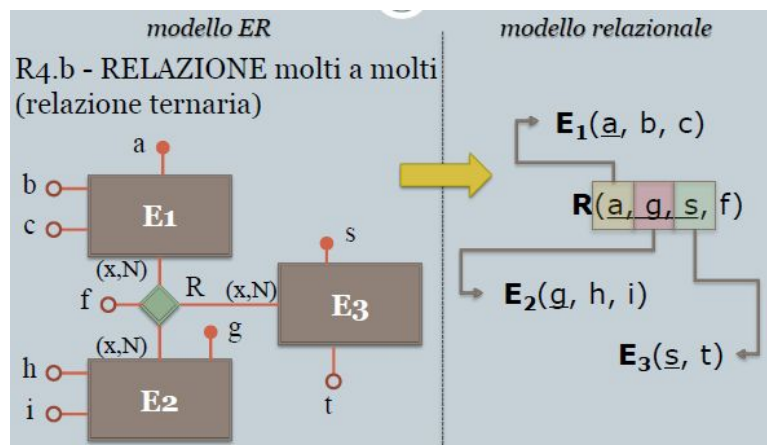
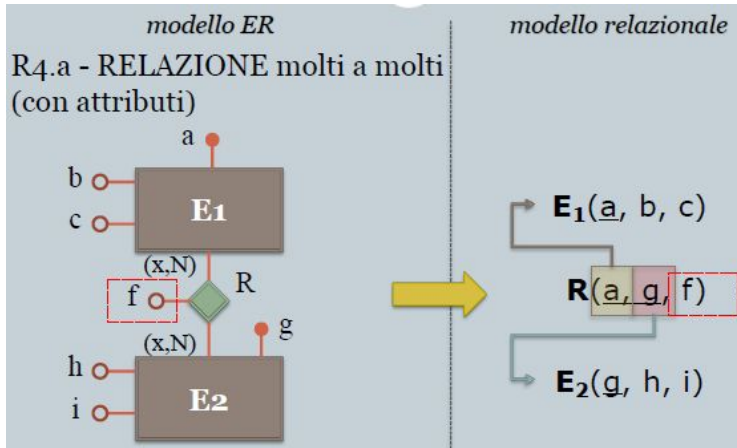
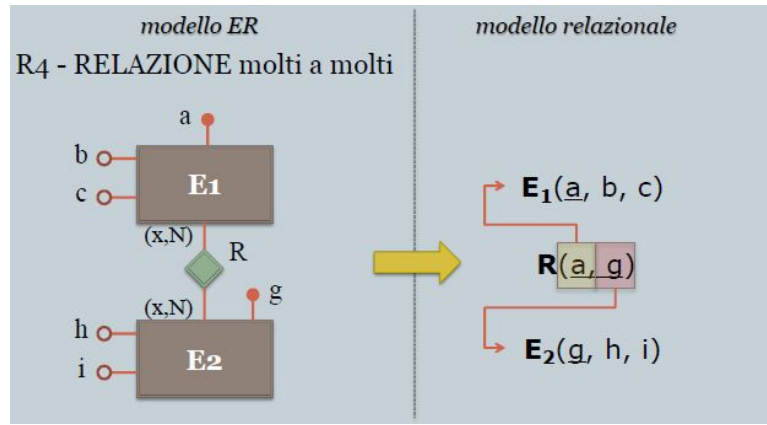
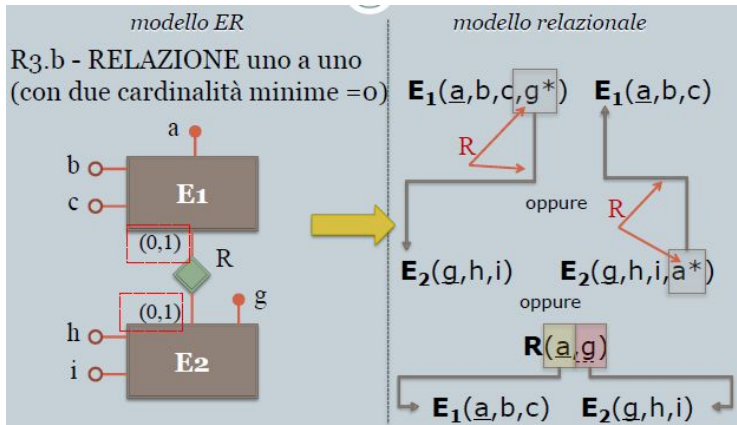
- Si rappresenta un'istanza di entità con una **tupla**.
- Si rappresenta un'istanza di relazione con un **legame tra tuple** o con una **tupla esplicita**.

Classificazione delle relazioni binarie del modello ER:

- **UNO a UNO**: se entrambe le entità coinvolte nella relazione hanno cardinalità massima uguale a UNO.
- **UNO a MOLTI**: se una delle entità coinvolte nella relazione ha cardinalità massima uguale a UNO e l'altra ha cardinalità massima > 1.
- **MOLTI a MOLTI**: se entrambe le entità coinvolte nella relazione hanno cardinalità massima > 1.

Regole di traduzione





Operazioni su una base di dati relazionale

- **Data Definition Language (DDL):** linguaggio per la definizione dei dati.
 - **Creare e modificare lo schema** della base di dati.
- **Data Manipulation Language (DML):** linguaggio per la manipolazione dei dati.
 - **Inserire/cancellare tuple** nelle relazioni.
 - **Aggiornare** i valori nelle tuple delle relazioni.
 - **Interrogare** le relazioni per estrarre informazione.



Linguaggi di interrogazione:

- Linguaggi di tipo **procedurale**: specificano il procedimento per ottenere il risultato (es. algebra relazionale).
- Linguaggi di tipo **dichiarativo**: specificano le proprietà del risultato (es. calcolo relazionale e SQL).

Problemi legati alla presenza di ridondanza

Se in una relazione (tabella) si uniscono concetti disomogenei e con esistenza autonoma si presenta la seguente situazione: ogni tupla rappresenta un'istanza della associazione che lega i concetti autonomi, per le istanze di informazione che sono coinvolte in più associazioni si produce una ridondanza inutile di valori in diverse tuple.

- Anomalia di aggiornamento: per aggiornare il valore in un attributo si è obbligati a modificare lo stesso valore su più tuple.
- Anomalia di inserimento: per inserire una nuova istanza di un concetto è necessario inserire valori nulli per gli attributi non disponibili.
- Anomalia di cancellazione: per cancellare un'istanza è necessario cancellare valori ancora validi o sostituire con valori nulli quelli da cancellare.

Lo schema di una relazione è costituito dal nome della relazione e da un insieme di nomi per i suoi attributi: $R(A_1, \dots, A_n)$ oppure $R(A_1:D_1, \dots, A_n:D_n)$ dove $DOM(A_i) = D_i$

Lo schema di una base di dati è un insieme di schemi di relazione:

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

L'istanza di una relazione di schema $R(A_1, \dots, A_n)$ con $X = \{A_1, \dots, A_n\}$ è un insieme r di tuple su X ;

L'istanza di una base di dati di schema $S = \{R_1(A_{1,1}, \dots, A_{1,n_1}), \dots, R_m(A_{m,1}, \dots, A_{m,n_m})\}$ è un insieme di istanze di relazioni $db = \{r_1, \dots, r_m\}$ dove ogni r_i è un'istanza della relazione di schema $R_i(A_{i,1}, \dots, A_{i,n_i})$.

Valori nulli e vincoli di integrità

Nel caso di una base di dati reale, non sempre esistono tutti i valori per gli attributi di una tupla. Possono presentarsi i seguenti casi:

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

Per poter gestire queste situazioni, gli attributi di una tupla possono assumere un valore del dominio oppure un valore speciale detto NULLO. Definizione:

Una tupla su X è una funzione

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

dove:

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

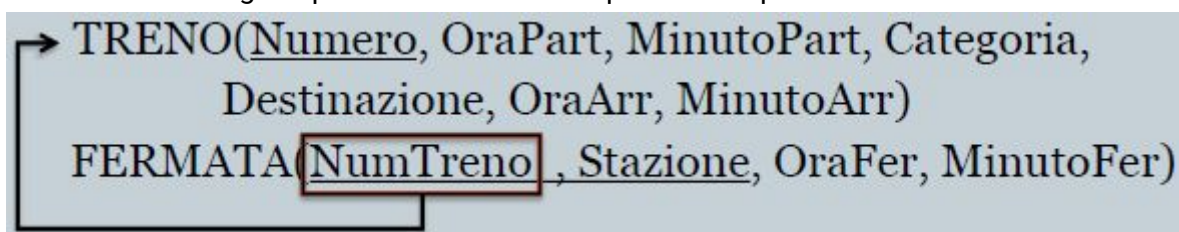
Nota: La presenza di valori nulli è accettabile solo su alcuni attributi, nel caso di attributi replicati per rappresentare legami tra tuple la presenza di valori nulli può rendere inutilizzabile l'informazione.

Vincoli di integrità

Quando è necessario introdurre vincoli sulla popolazione di una base di dati in quanto non tutte le possibili istanze sono corrette rispetto al sistema informativo considerato, a questo scopo si introduce il concetto di:

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

- Vincoli di dominio: impongono una restrizione sul dominio dell'attributo di una relazione.
- Vincoli di tupla: impongono una restrizione alla combinazione di valori che una tupla della relazione può assumere indipendentemente da altre tuple.
- Vincoli intrarelazionali: impongono una restrizione al contenuto di una relazione e specificano una condizione che ogni tupla della relazione deve soddisfare rispetto alle altre tuple della medesima relazione.
 - Superchiave: una relazione di schema $R(X)$, un insieme di attributi K , sottoinsieme di X , è superchiave per $R(X)$, se per ogni istanza r di $R(X)$ vale la condizione:
 - $\forall t, t' \in r : t \neq t' \Rightarrow t[K] \neq t'[K]$, dove $t[K] \neq t'[K] \equiv \exists A_i \in K : t[A_i] \neq t'[A_i]$
 - Chiave Candidata: una relazione di schema $R(X)$, un insieme di attributi K , sottoinsieme di X , è chiave candidata, o chiave, per $R(X)$, se K è superchiave per $R(X)$ e vale la condizione:
 - $\neg \exists K' \subset K : K' \text{ è superchiave per } R(X)$
 - Esiste sempre una chiave candidata K per una relazione $R(X)$.
 - Chiave Primaria: una relazione di schema $R(X)$ la sua chiave primaria è la chiave candidata scelta per identificare le tuple della relazione.
 - Non contiene mai valori nulli.
 - Su K il sistema genera una struttura di accesso ai dati (o indice) per supportare le interrogazioni.
 - Vincolo di integrità referenziale: Un vincolo di integrità referenziale tra un insieme di attributi $Y=\{A_1, \dots, A_p\}$ di R_1 e un insieme di attributi $K=\{K_1, \dots, K_p\}$, chiave primaria di un'altra relazione R_2 , è soddisfatto se, per ogni istanza r_1 di R_1 e per ogni istanza r_2 di R_2 vale la condizione:
 - $\forall t \in r_1 : \exists s \in r_2 : \forall i \in \{1, \dots, p\} : t[A_i] = s[K_i]$
 - Impongono una restrizione al contenuto di una relazione e specificano una condizione che ogni tupla deve soddisfare rispetto alle tuple di altre relazioni della base di dati.



Algebra relazionale

È un insieme di operatori su relazioni.

- Operatore: rappresenta un'operazione algebrica chiusa sull'insieme delle relazioni.
 - **Unari**: $op(r_1) = r_2$;
 - **Binari**: $op(r, r_2) = r_3$;

Classificazione degli operatori (di **base** o **derivati**):

❖ Operatori insiemistici

Premessa: Tutte le **tuple** della relazione sono **omogenee** (hanno gli **stessi attributi X**) \Rightarrow Posso **applicare un operatore insiemistico solo a relazioni con lo stesso schema**.

➤ Unione (di base)

- Sintassi: $r_1 \cup r_2$
- Semantica:
 - Schema: X
 - Istanza: $\{t \mid t \in r_1 \vee t \in r_2\}$
 - Cardinalità: $\max(|r_1|, |r_2|) \leq |r_1 \cup r_2| \leq |r_1| + |r_2|$

➤ Differenza (di base)

- Sintassi: $r_1 - r_2$
- Semantica:
 - Schema: X
 - Istanza: $\{t \mid t \in r_1 \wedge t \notin r_2\}$
 - Cardinalità: $0 \leq |r_1 - r_2| \leq |r_1|$

➤ Intersezione (derivato)

- Sintassi: $r_1 \cap r_2$
- Semantica:
 - Schema: X
 - Istanza: $r_1 - (r_1 - r_2)$

❖ Operatori specifici

Premessa: sia r una relazione di schema $R(x)$ con $X = \{A_1, \dots, A_n\}$ si definiscono i seguenti operatori:

➤ Ridenominazione (di base): *[cambio nome degli attributi]*

- Sintassi: $P_{A_1, \dots, A_n \rightarrow B_1, \dots, B_n}(r)$
- Semantica:
 - Schema: Y
 - Istanza: $\{t \mid \exists t' \in r: \forall_i \in [1, \dots, n]: t[B_i] = t'[A_i]\}$

Note: molto utile per poi poter effettuare operazioni di unione e differenza rendendo prima gli schemi delle due relazioni uguali.

➤ Selezione (di base): *[come una WHERE in SQL]*

- Sintassi: $\sigma_F(r)$
- Semantica:
 - Schema: X
 - Istanza: $\{t \mid t \in r \wedge F(t)\}$ (dove $F(t)=F$ valutata su t)

F è una formula proposizionale ottenuta componendo attraverso i connettivi logici:

AND(), OR(), NOT formule atomiche del tipo $A \theta B$ oppure $A \theta C$, dove:

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

- Una formula atomica $A \theta B$ oppure $A \theta C$ è vera nella tupla T se la seguente condizione è soddisfatta: $t[A] \theta t[B]$ oppure $t[A] \theta C$.

Una formula proposizionale $F_1 \wedge F_2, F_1 \vee F_2, \neg F_1$ è valutata sulla tupla T seguendo quanto specificato nelle tabelle di verità degli operatori logici \wedge, \vee, \neg .

➤ **Proiezione (di base):** *[come una SELECT in SQL]*

Premessa: sia r una relazione di schema $R(x)$ con $X = \{A_1, \dots, A_n\}$

- Sintassi: $\Pi_y(r)$ dove $y \subset x, y = \{A_1, \dots, A_m\} m < n$
- Semantica:
 - Schema: Y
 - Istanza: $\{t \mid \exists t' \in r: t = t'[y]\}$ dove $t'[y]$ è una tupla t' tale che:

❖ **Operatori di giunzione (join)**

Premessa: permettono di “unire” in un'unica relazione (tabella) le informazioni contenute in **due relazioni** dove i vari operatori di questa categoria si distinguono per il tipo di condizione che le tuple delle due relazioni devono soddisfare per produrre una tupla nel risultato.

➤ **Join naturale (di base)**

La **condizione** di join è una condizione di **UGUAGLIANZA sugli attributi comuni** alle due relazioni (dipende dallo schema delle relazioni).

Siano r_1 e r_2 di schema $R_1(x_1)$ e $R_2(x_2)$ rispettivamente, si definisce il join naturale come segue:

- Sintassi: $r_1 \bowtie r_2$
- Semantica:
 - Schema: $x_1 \cup x_2$
 - Istanza $\{t \mid \exists t_1 \in r_1 : \exists t_2 \in r_2: t[x_1] = t_1 \wedge t[x_2] = t_2\}$
 - Cardinalità: $\emptyset \leq |r_1| \bowtie |r_2| \leq |r_1| \cdot |r_2|$
 - ◆ Se $x_1 \cup x_2$ è superchiave per r_2 : $\emptyset \leq |r_1| \bowtie |r_2| \leq |r_1|$
 - ◆ Se $x_1 \cap x_2$ è una superchiave per r_2 e vale un vincolo di integrità ref. tra gli attributi $x_1 \cap x_2$ di r_1 e r_2 : $|r_1| \bowtie |r_2| = |r_1|$

➤ **θ join (derivato)**

Premessa: date due relazioni r_1 e r_2 di schema $R_1(x_1)$ e $R_2(x_2)$ rispettivamente e dove $x_1 \cap x_2 = \emptyset$ (condizione necessaria per applicare correttamente il θ join) si definisce come segue:

- Sintassi: $r_1 \bowtie_{\theta} r_2$
- Semantica: $r_1 \bowtie_{\theta} r_2 = \sigma_{\theta}(r_1 \bowtie r_2) \Rightarrow$ poiché $x_1 \cap x_2 = \emptyset$ (prodotto cartesiano)

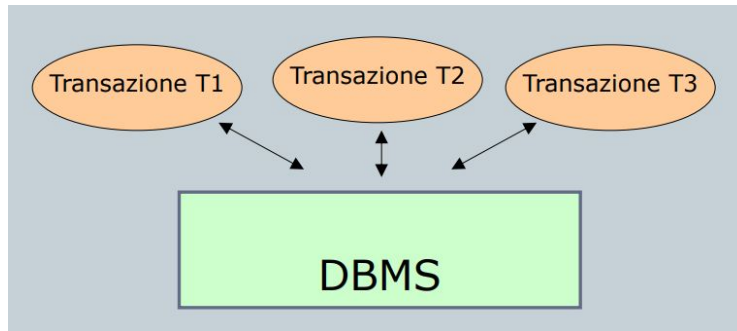
Note: θ può essere una qualsiasi condizione di relazione ammessa da σ :

- θ come congiunzione di condizioni di uguaglianza $A = B \wedge C = D \dots$, in questo caso si chiama equi-join.
- θ può contenere anche condizioni che non sono di uguaglianza: $A > B \wedge C < D$
- L'equivalenza tra join naturale e θ join è espressa da il join naturale tra due relazioni r_1 e r_2 di schemi $R_1(x_1)$ e $R_2(x_2)$ rispettivamente dove $x_1 \cap x_2 = \{c_1 \dots c_n\}$ equivale in θ join:

$$r_1 \bowtie r_2 = \Pi_{x_1 \cup x_2} (r_1 \bowtie_{c_1 = c_1', \dots, c_m = c_m'} \rho_{c_1, \dots, c_m \rightarrow c_1', \dots, c_m'}(r_2))$$

Transazioni

Un DBMS è un sistema transazione, cioè fornisce un meccanismo per la definizione e esecuzione (concorrente) di transazioni.



Transazione: unità di lavoro svolto da un programma applicativo, che interagisce con la base di dati, per la quale si vogliono garantire alcune proprietà:

- **Atomicità:** una transazione è un'unità di esecuzione indivisibile, o viene eseguita completamente, o non viene eseguita affatto.
 - Se una transazione viene interrotta prima del commit, bisogna effettuare il rollback delle modifiche ripristinando lo stato iniziale.
 - Se una transazione viene interrotta al momento del commit, bisogna garantire che le modifiche abbiano effetto sulla base di dati.
- **Consistenza:** l'esecuzione di una transazione non deve violare i vincoli di integrità.
 - **Verifica immediata:** viene abortita l'ultima operazione e restituisce un'errore.
 - **Verifica differita:** la verifica viene effettuata al commit, in caso di errore viene abortita l'intera transazione.
- **Isolamento:** l'esecuzione di una transazione deve essere indipendente dalla contemporanea esecuzione di altre transazioni.
- **Persistenza:** l'effetto delle transazioni che hanno effettuato il commit non deve andare perso.
 - L'effetto di una transazione che ha effettuato il commit al momento di un guasto non deve andare perso.

⇒ **A.C.I.D.**

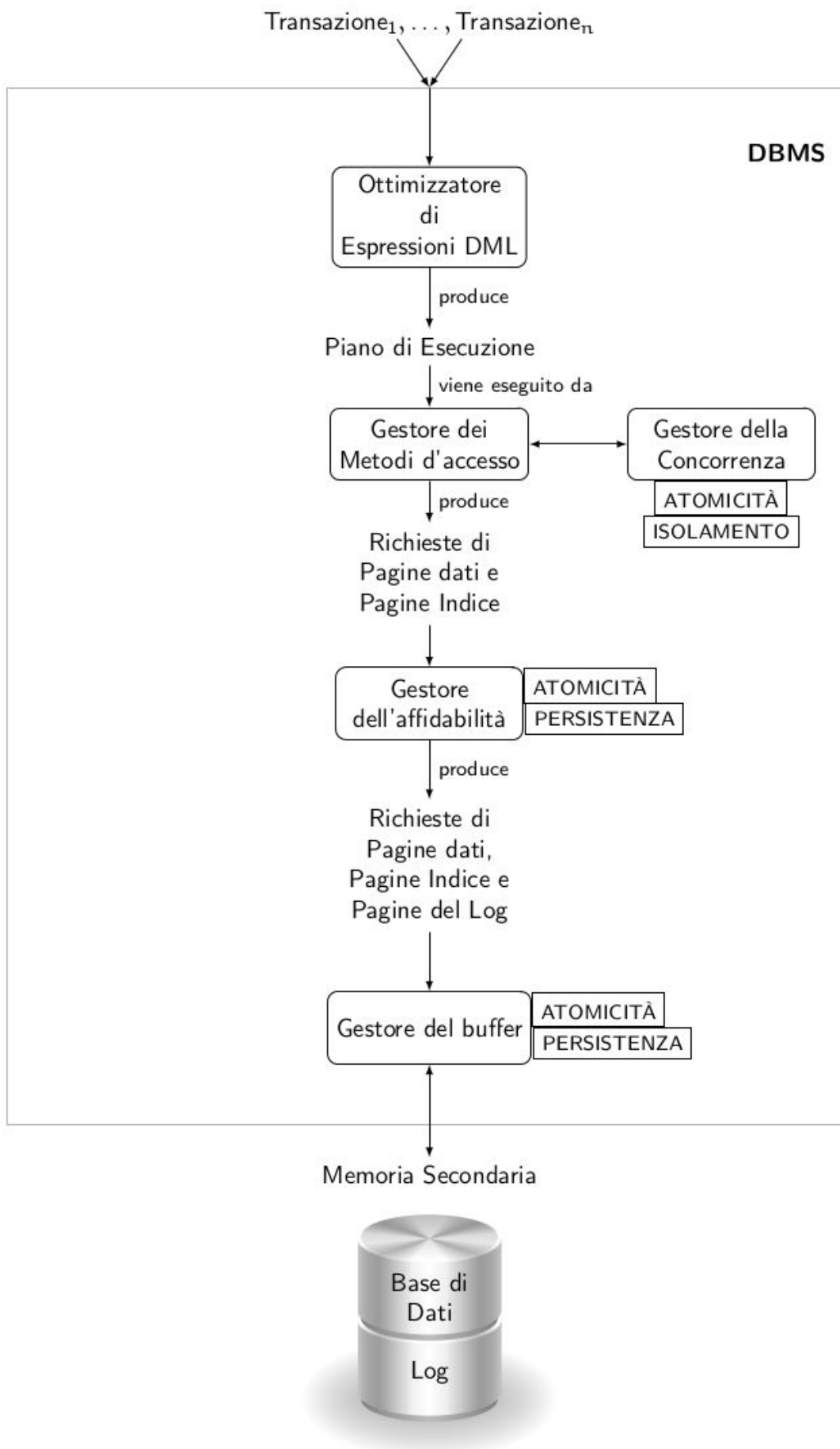
Sintassi:

```
BEGIN TRANSACTION;  
...  
    <ISTRUZIONE> | COMMIT WORK | ROLLBACK WORK;  
...  
END TRANSACTION;
```

(Dopo commit o rollback non vengono effettuati altri accessi alla base di dati.)

Architettura di un DBMS

L'architettura mostra i moduli principali che possiamo individuare nei DBMS attuali, considerando le diverse funzionalità che il DBMS svolge durante l'esecuzione delle transazioni.



Strutture fisiche e di accesso ai dati

Le basi di dati sono **grandi** e **persistenti**, quindi non possono essere mantenute in memoria centrale, ma in **memoria secondaria**:

- Il **costo** delle operazioni è **ordini di grandezza maggiore**.
- **Non** direttamente **accessibile** dai **programmi**.
- **Organizzata** in **blocchi**, operazioni di lettura/scrittura su più blocchi (o **pagine**).

Gestore del Buffer (Cache)

- Si occupa del **trasferimento delle pagine** dalla memoria secondaria alla memoria centrale, e viceversa.
- Il **buffer** è una **zona condivisa** da diverse applicazioni.
- La gestione ottimale del buffer è strategica per avere buone prestazioni di accesso alla base di dati.
- Politica di gestione del buffer:
 - Se un blocco è **già presente nel buffer**, non si esegue nessuna lettura da memoria secondaria e si **ritorna il puntatore al buffer**.
 - Se un blocco **non è nel buffer**, **carica il blocco** nel buffer in una pagina libera e torna il puntatore
 - In caso di **scrittura** su memoria secondaria di un blocco del buffer, essa **può essere ritardata**.
⇒ L'obiettivo è aumentare la velocità di accesso ai dati (R/W).
- Per la gestione delle pagine, sono necessari:
 - Per ogni **pagina**, il **file** e il **numero di blocco** in memoria secondaria.
 - Viene **memorizzato uno stato**: **count** del numero di transazioni che usano la pagina, **flag di modifica** della pagina.
- Primitive per la gestione delle pagine:
 - **FIX**: richiede l'accesso a un blocco:
 - Se il blocco è nel buffer, ne viene ritornato il puntatore
 - Altrimenti:
 - Se si trova una pagina libera(=nessuna transazione sta usando un blocco) si carica il blocco.
 - Altrimenti, si può applicare una politica **STEAL**, **scaricando un blocco in uso**, o **NO_STEAL**, mettendo la transazione in una **coda di attesa**.
 - ⇒ Quando una transazione accede a una pagina, count++.
 - **SET_DIRTY**: la transazione indica che un blocco è stato modificato (flag di modifica a 1).
 - **UNFIX**: la transazione indica che ha terminato di usare il blocco (count--).
 - **FORCE**: salva il blocco in memoria secondaria in modo sincrono (flag di modifica a 0)
 - **FLUSH**: salva il blocco in memoria secondaria in modo asincrono (usato sulle pagine dirty).

Gestore dell'affidabilità

È il modulo responsabile della **gestione delle transazioni** e realizzare le operazioni necessarie al **ripristino della base di dati** dopo possibili malfunzionamenti.

- Viene usato un **LOG** su un dispositivo di **memoria stabile** (resistente ai guasti \Rightarrow ridondanza) che registra sequenzialmente le modifiche apportate dalle transazioni alla base di dati.
 - **Record di transazione:**
 - begin transaction T: B(T)
 - commit transaction T: C(T)
 - abort transaction T: A(T)
 - insert, update e delete sull'oggetto O:
 - record I(T,O,AS): AS=After State
 - record D(T,O,BS): BS=Before State
 - record U(T,O,BS,AS)
 - I **record di transazione** memorizzati nel log consentono di eseguire le seguenti **operazioni** (viene sempre controllata l'esistenza di O \Rightarrow no duplicati):
 - **UNDO:** per disfare un'azione su un oggetto O è sufficiente ricopiare in O il valore BS; l'insert/delete viene disfatto cancellando/inserendo O.
 - **REDO:** per rifare un'azione su un oggetto O è sufficiente ricopiare in O il valore AS; l'insert/delete viene rifatto inserendo/cancellando O;
- **Record di sistema:** vengono effettuate le seguenti operazioni:
 - **DUMP** della base di dati: viene effettuata una copia dell'intera base di dati.
 - **CHECKPOINT:** record CK(T1, ..., Tn) indica che all'esecuzione del CheckPoint le transazioni attive erano T1, ..., Tn.
 - Viene effettuata periodicamente dal gestore dell'affidabilità:
 - Sospensione di scrittura, commit e abort delle transazioni.
 - Esecuzione di FORCE sulle pagine "dirty" (vengono scritte su memoria secondaria).
 - Scrittura sincrona sul file di log del record di CHECKPOINT con gli identificatori delle transazioni attive.
 - Ripresa di scritture, commit e abort delle transazioni.
- **Regole per la scrittura del log:**
 - **WAL (Write Ahead Log):** i record di log devono essere scritti prima di effettuare l'operazione sulla base di dati.
 - \Rightarrow Posso sempre fare un UNDO.
 - **Commit-Precedenza:** record di log devono essere scritti prima dell'esecuzione del commit di una transazione
 - \Rightarrow Posso sempre fare un REDO

COMMIT DI UNA TRANSAZIONE: Una transazione sceglie in modo atomico l'esito di COMMIT nel momento in cui scrive nel file di log in modo sincrono (primitiva FORCE) il suo record di COMMIT.

Le operazioni di ripristino variano in base al guasto subito:

- **Guasto di sistema: perdita** del contenuto della **memoria centrale**.
 - **Ripresa a caldo:**
 - Si accede all'ultimo blocco del log e si ripercorre all'indietro fino al più recente record di **CHECKPOINT**.
 - Si **decidono le transazioni da rifare/disfare** iniziando l'insieme UNDO con le transazioni attive al CHECKPOINT e l'insieme REDO con l'insieme vuoto.
 - Si **ripercorre in avanti il log** e per ogni record B(T) incontrato si aggiunge T a UNDO e per ogni record C(T) incontrato si sposta T da UNDO a REDO.

- Si **ripercorre all'indietro il log** disfacendo le operazioni eseguite dalle transazioni in UNDO risalendo fino alla prima azione della transazione più vecchia.
- Si **rifanno le operazioni delle transazioni** dell'insieme REDO.
- **Guasto di un dispositivo: perdita di parte o tutto del contenuto della memoria secondaria.**
 - **Ripresa a freddo:**
 - Si **accede al DUMP** della base di dati e si **ricopia** selettivamente la **parte deteriorata** della base di dati.
 - Si **accede** al log **risalendo** al record di **DUMP**.
 - Si **ripercorre in avanti** il log, rieseguendo tutte le operazioni relative alla parte deteriorata comprese le azioni di COMMIT e ABORT.
 - Si **applica una ripresa a caldo**.

Gestione dei metodi di accesso

È il modulo del DBMS che **applica il piano di esecuzione** prodotto dall'ottimizzatore e **genera le sequenze di accesso** alle **pagine** presenti in **memoria secondaria**.

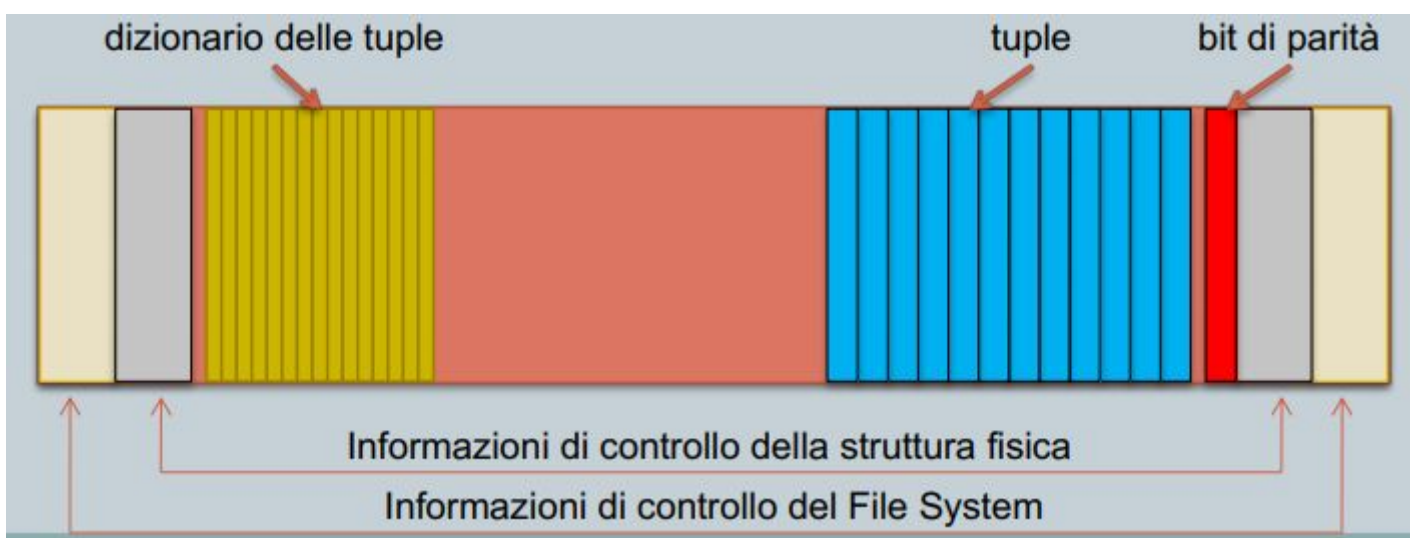
Metodi di accesso: implementano gli algoritmi di accesso e manipolazione dei dati memorizzati in determinate strutture fisiche.

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

Ogni **metodo di accesso** conosce:

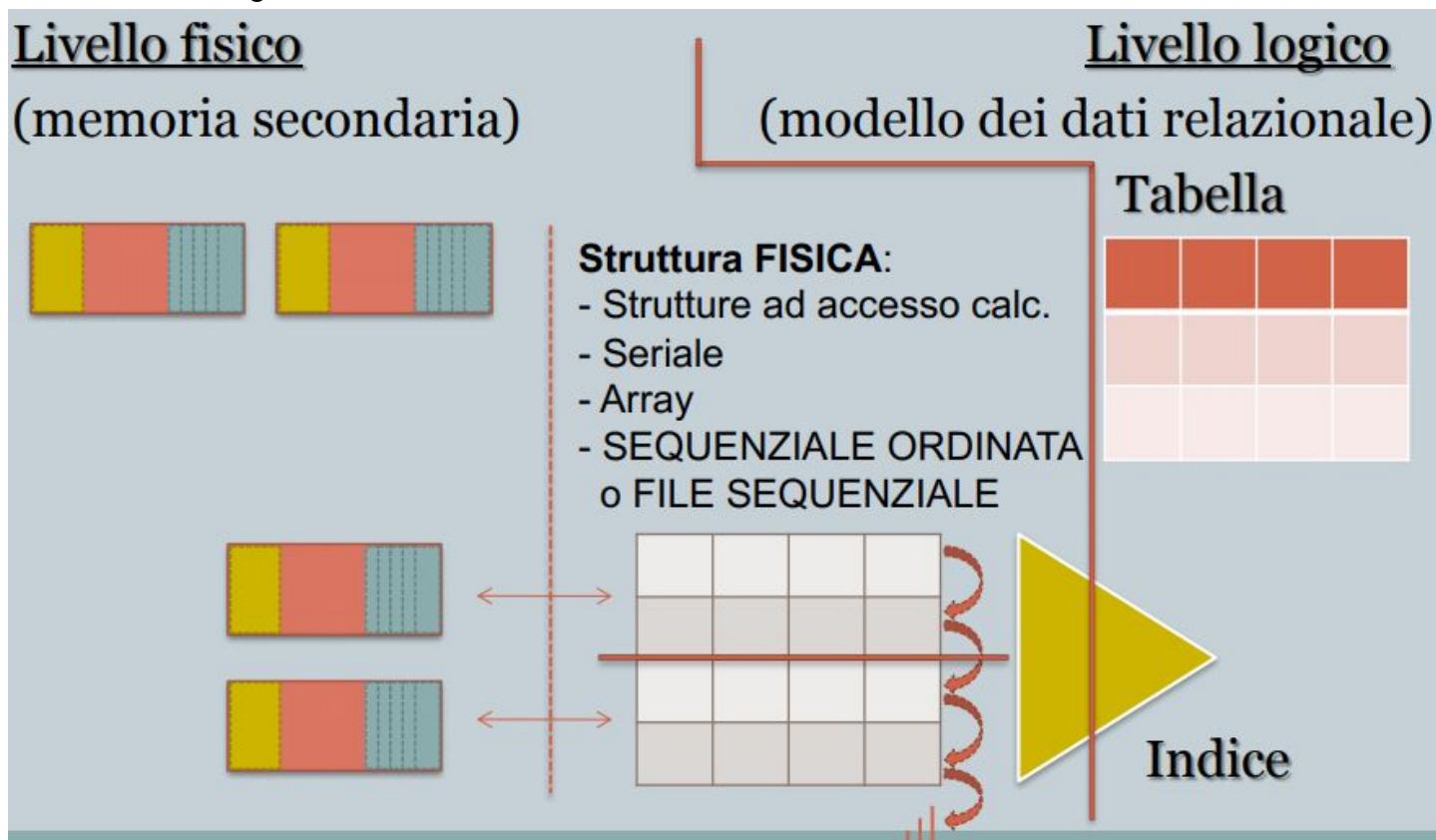
- **L'organizzazione delle tuple** nelle pagine DATI salvate in memoria secondaria (come una tabella viene organizzata in pagine DATI della memoria secondaria).
- **L'organizzazione fisica** delle pagine DATI contenenti tuple e delle pagine che memorizzano le strutture fisiche di accesso o INDICI (come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

In ogni **pagina** dati sono presenti sia le **tuple della tabella**, sia **informazioni di controllo**, come **dizionari**, **bit di parità**, informazioni del filesystem o della specifica struttura fisica.



Struttura di una pagina dati:

- **Dizionario:** necessario per poter memorizzare le tuple
 - Se le **tuple** sono di **lunghezza fissa non** è **necessario**, basta memorizzare l'**offset** dal punto iniziale e la dimensione, altrimenti, il dizionario memorizza l'offset di ogni tupla presente nella pagina e di ogni attributo della tupla.
 - Se la **tupla supera la dimensione disponibile su una pagina**, bisogna gestire le tuple memorizzando su **diverse pagine**.
- **Operazioni:**
 - **Inserimento:**
 - Se spazio contiguo ok \Rightarrow inserimento semplice.
 - Se spazio ok ma non contiguo \Rightarrow riorganizzazione e inserimento semplice.
 - Se non c'è spazio sufficiente \Rightarrow inserimento rifiutato.
 - **Cancellazione:** sempre possibile anche senza riorganizzare.
 - **Accesso a una tupla**
 - **Accesso ad un attributo di una tupla**
 - **Accesso sequenziale** (ordine di chiave primaria)
 - **Riorganizzazione**



Struttura di un file sequenziale:

- È un file dove le **tuple sono ordinate** secondo una **chiave di ordinamento**.
- **Operazioni:**
 - **Inserimento:**
 - Individuare la pagina che contiene la tupla che precede, nell'ordine della chiave.
 - Inserire la nuova tupla in quella pagina, se non c'è spazio, si aggiunge una nuova pagina, e si aggiusta la catena di puntatori.
 - **Scansione sequenziale** secondo la chiave
 - **Cancellazione:** si individua la pagina che contiene la tupla, la si cancella, e si aggiusta la catena di puntatori.
- **Riorganizzazione:** si assegnano le tuple alle pagine in base ad opportuni coefficienti di riempimento, aggiustando i puntatori.

Indici

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

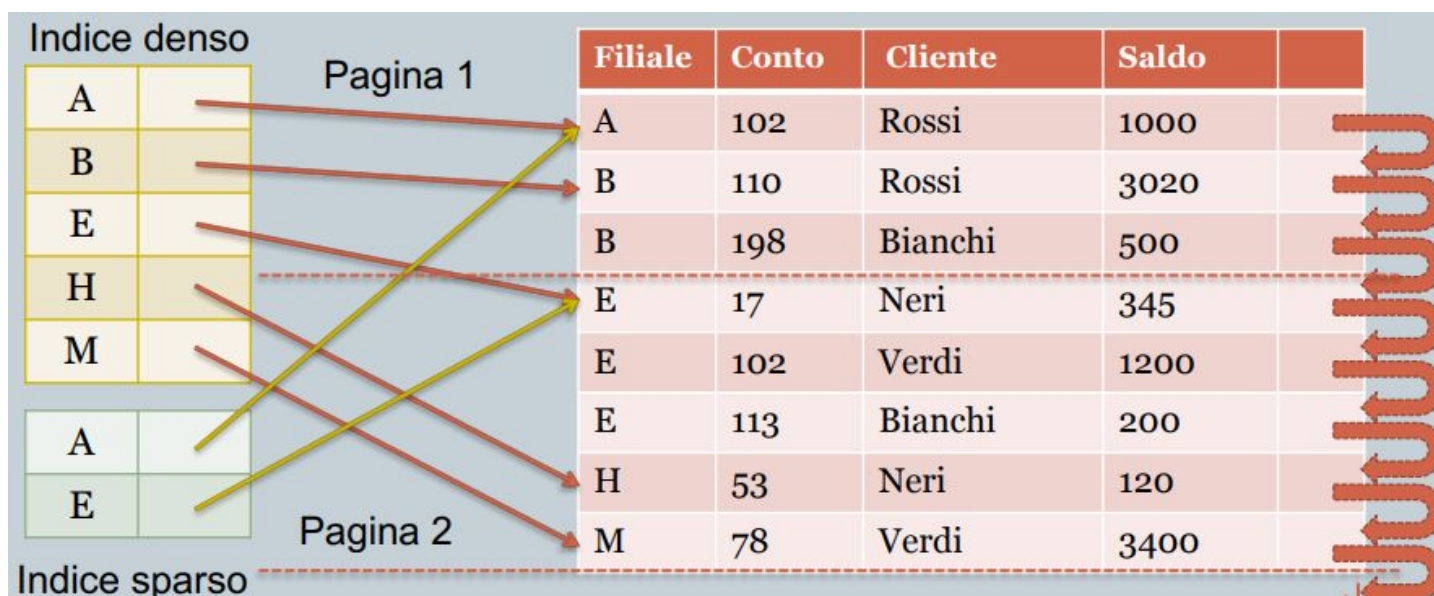
- **Primario**: la chiave di ordinamento del file sequenziale coincide con la chiave di ricerca dell'indice.
- **Secondario**: chiave di ordinamento e di ricerca sono diverse.

Indice primario: Ogni record dell'indice primario contiene una coppia $\langle v_i, p_i \rangle$:

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

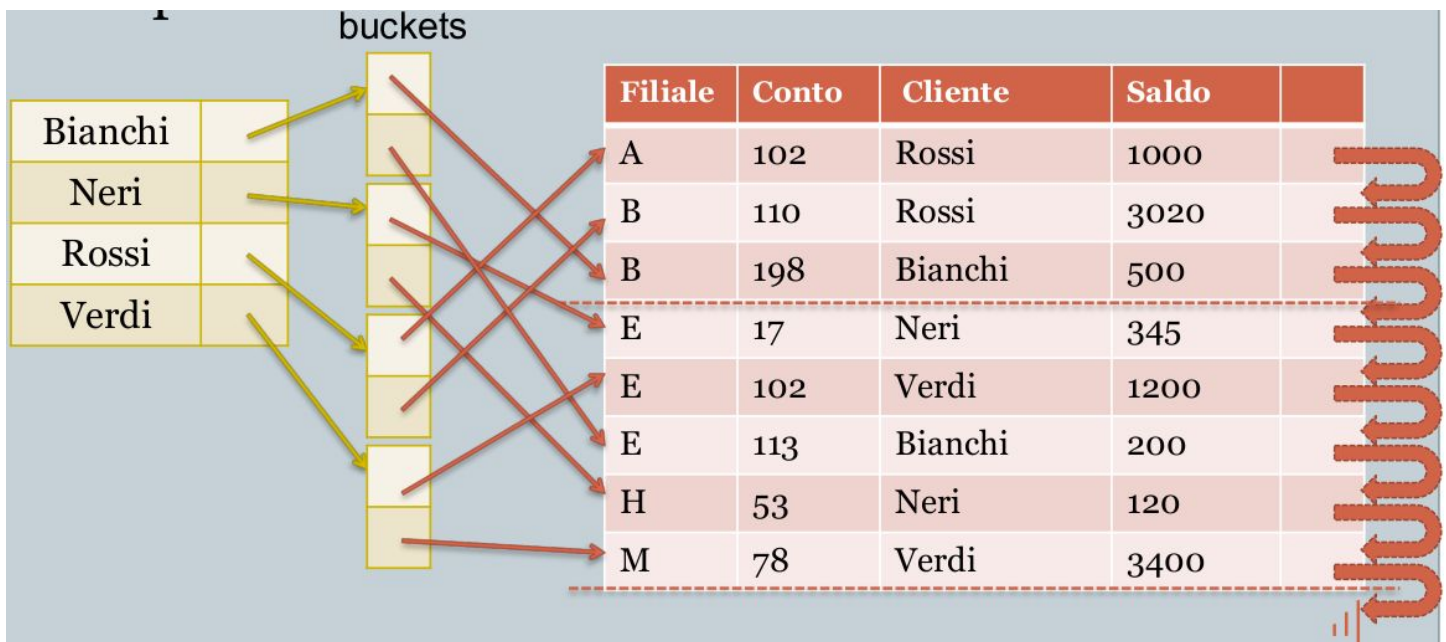
Due possibili varianti:

- **Indice denso**: per ogni occorrenza della chiave presente nel file esiste un corrispondente record nell'indice.
 - **Ricerca**: tupla con chiave K (K è presente nell'indice):
 - Scansione sequenziale dell'indice per trovare il record (K, p_k) .
 - Accesso al file attraverso il puntatore p_k .
 - COSTO: 1 accesso indice + 1 accesso pagina dati.
 - **Inserimento**: avviene solo se la tupla inserita nel file ha un valore di chiave K che non è già presente.
 - **Cancellazione**: avviene solo se la tupla cancellata nel file è l'ultima tupla con valore di chiave K.
- **Indice sparso**: solo per alcune occorrenze della chiave presenti nel file esiste un corrispondente record nell'indice, tipicamente una per pagina.
 - **Ricerca**: tupla con chiave K (K potrebbe non essere presente nell'indice):
 - Scansione sequenziale dell'indice fino al record (K', p_k') dove K' è il valore più grande che sia minore o uguale a K.
 - Accesso al file attraverso il puntatore p_k' e scansione del file (pagina corrente) per trovare le tuple con chiave K.
 - COSTO: 1 accesso indice + 1 accesso pagina dati.
 - **Inserimento**: avviene solo quando, per effetto dell'inserimento di una nuova tupla, si aggiunge una pagina dati alla struttura; in tutti gli altri casi l'indice rimane invariato.
 - **Cancellazione**: avviene solo quando K è presente nell'indice e la corrispondente pagina viene eliminata; altrimenti, se la pagina sopravvive, va sostituito K nel record dell'indice con il primo valore K' presente nella pagina.



Indice Secondario: usa una **chiave** di ricerca che **NON coincide con la chiave di ordinamento** del file sequenziale.

- Ogni record dell'indice secondario contiene una coppia $\langle v_i, p_i \rangle$:
 - v_i : valore della chiave di ricerca.
 - p_i : puntatore al bucket di puntatori che individuano nel file sequenziale tutte le tuple con valore di chiave v_i .
- Gli indici secondari sono **sempre densi**.
- **Operazioni:**
 - **Ricerca:** tupla con chiave K:
 - Scansione sequenziale dell'indice per trovare il record (K, p_k) .
 - Accesso al bucket B di puntatori attraverso il puntatore p_k .
 - Accesso al file attraverso i puntatori del bucket B.
 - **COSTO:** 1 accesso indice + 1 accesso al bucket + n accessi pagine dati.
 - **Inserimento e cancellazione:** come indice primario denso con in più l'aggiornamento dei bucket.



B⁺-tree

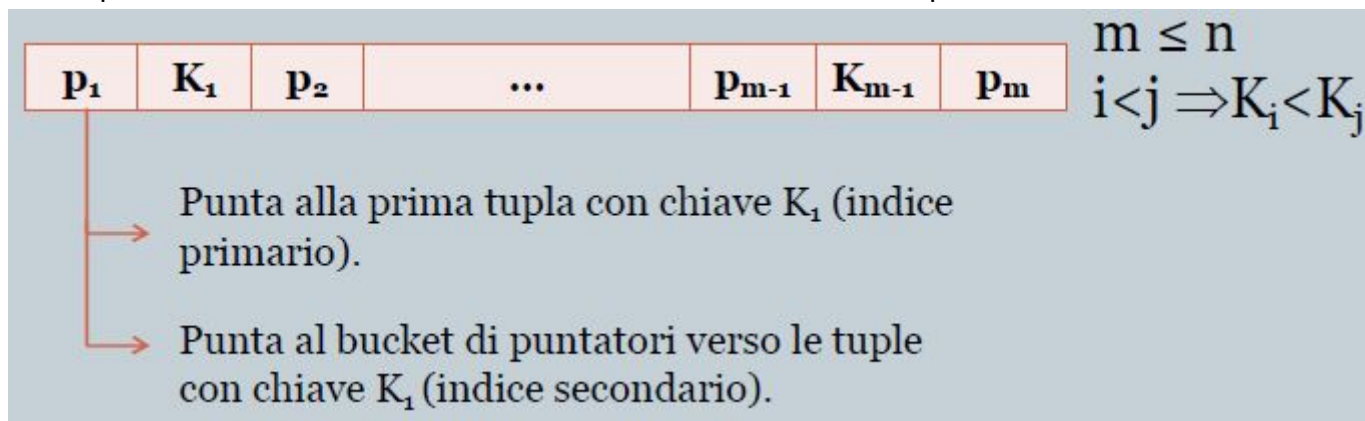
- è una struttura ad **albero bilanciato**, il **percorso da radice a foglie è costante**.
- **ogni nodo** corrisponde a una **pagina** in memoria secondaria.
- **ogni legame** tra i nodi è un **puntatore a pagina**.
- **ogni nodo** ha **tanti nodi figli**, quindi ha pochi livelli e tante foglie.
- **inserimenti e cancellazioni non alterano le prestazioni** di accesso ai dati: l'albero rimane bilanciato.

Struttura di un B⁺-tree

fan-out = numero di figli di ogni nodo = **n**

Nodo foglia

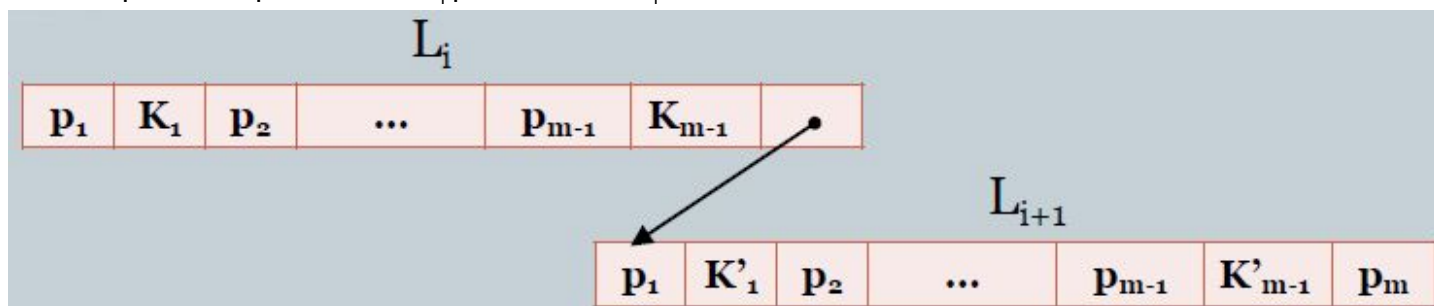
- può contenere fino a **n-1** valore della chiave di ricerca e fino a n puntatori.



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

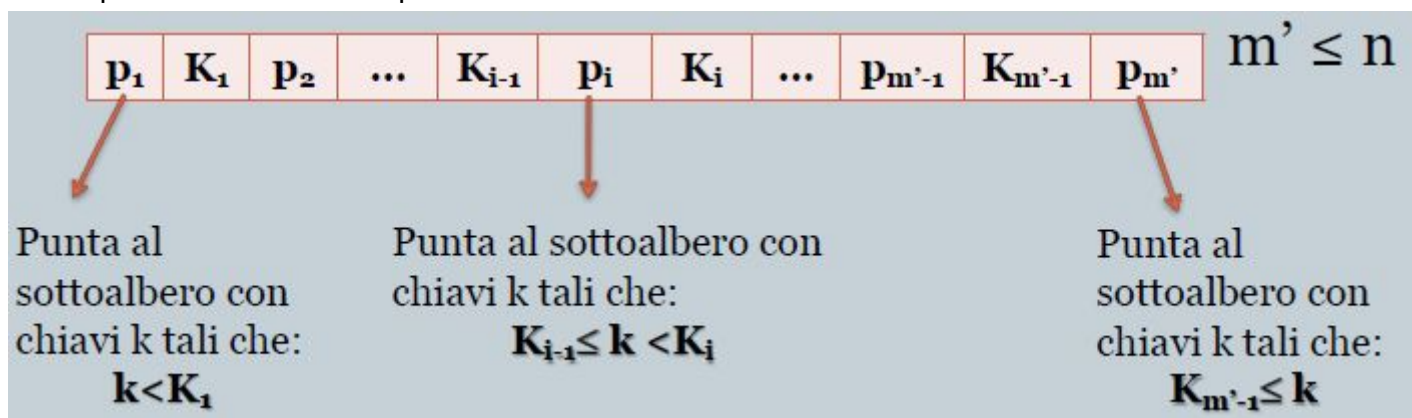
$$\forall K_t \in L_i: \forall K_s \in L_j: K_t < K_s$$

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



Nodo intermedio

- può contenere fino a n puntatori a nodo.



Regole di bilanciamento

- **Nodo foglia:** contiene un numero di **#chiavi**:

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

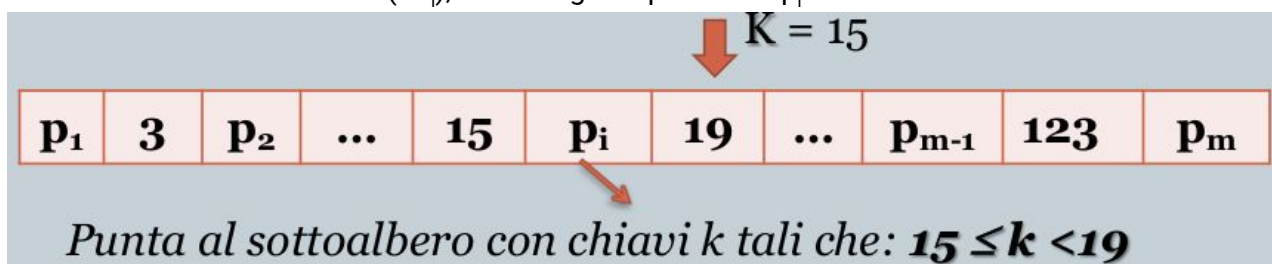
- **Nodo intermedio:** contiene un numero di puntatori **#puntatori** vincolato come segue (per la **radice non vale il minimo**):

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

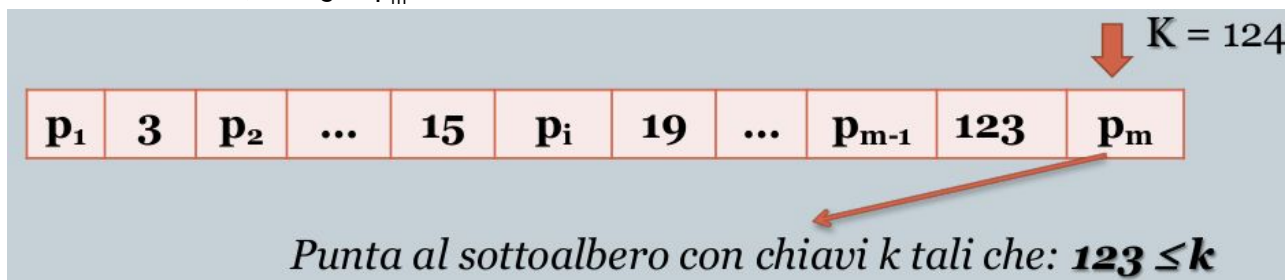
Operazioni

Ricerca: per cercare un record con chiave K:

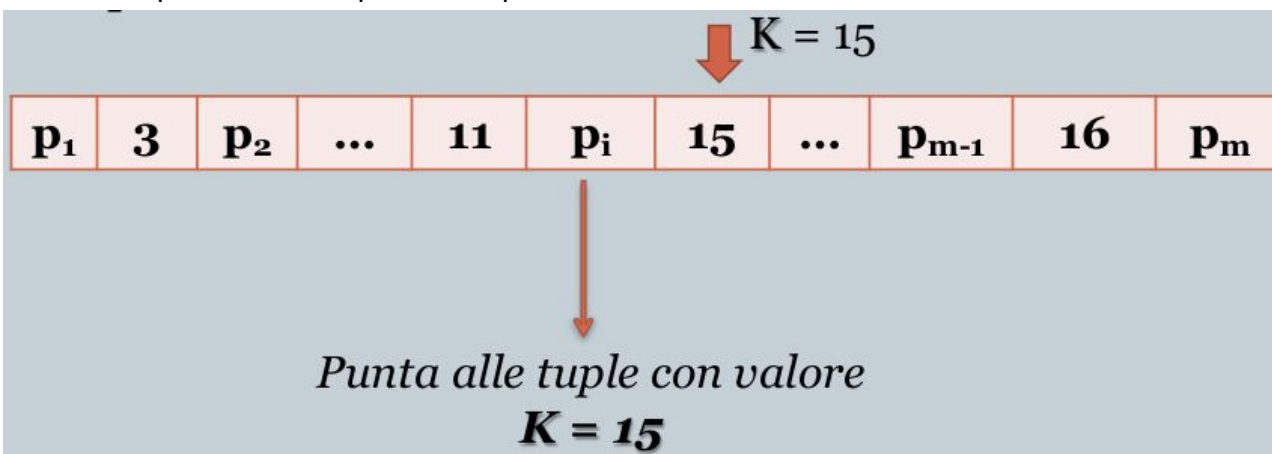
1. Cerco nella radice il più piccolo valore di chiave maggiore di K.
 - a. Se tale valore esiste ($=K_i$), allora seguo il puntatore p_i :



- b. Altrimenti, si segue p_m :

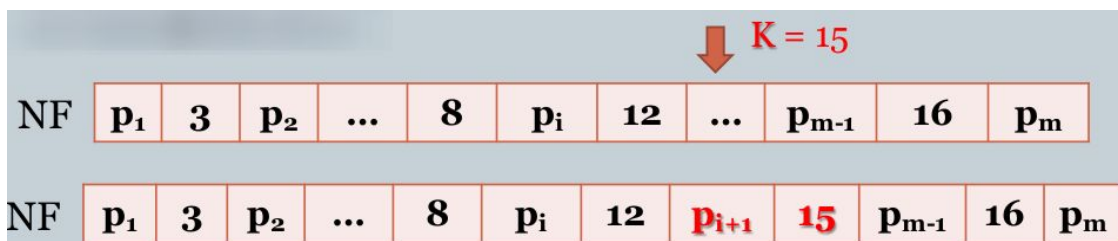


2. Se il nodo raggiunto è un nodo foglia cercare il valore K nel nodo e seguire il corrispondente puntatore verso le tuple, altrimenti riprendere il passo 1.



Inserimento di un valore della chiave K:

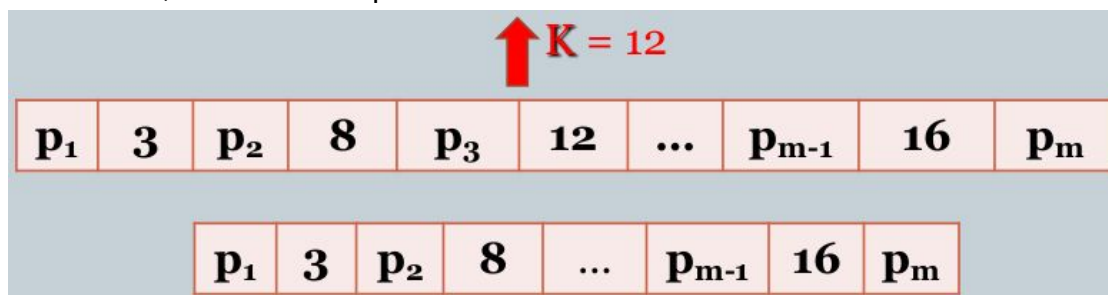
- ricerca del nodo foglia NF dove il valore K va inserito.
- se K è presente in NF, allora:
 - Indice primario: nop.
 - Indice secondario: aggiorna i bucket.
- altrimenti, inserire K in NF rispettando l'ordine:
 - Indice primario: inserire puntatore alla tupla con valore K della chiave
 - Indice secondario: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



- Se non è possibile inserire K in NF, allora è necessario effettuare uno SPLIT di NF.

Cancellazione di un valore:

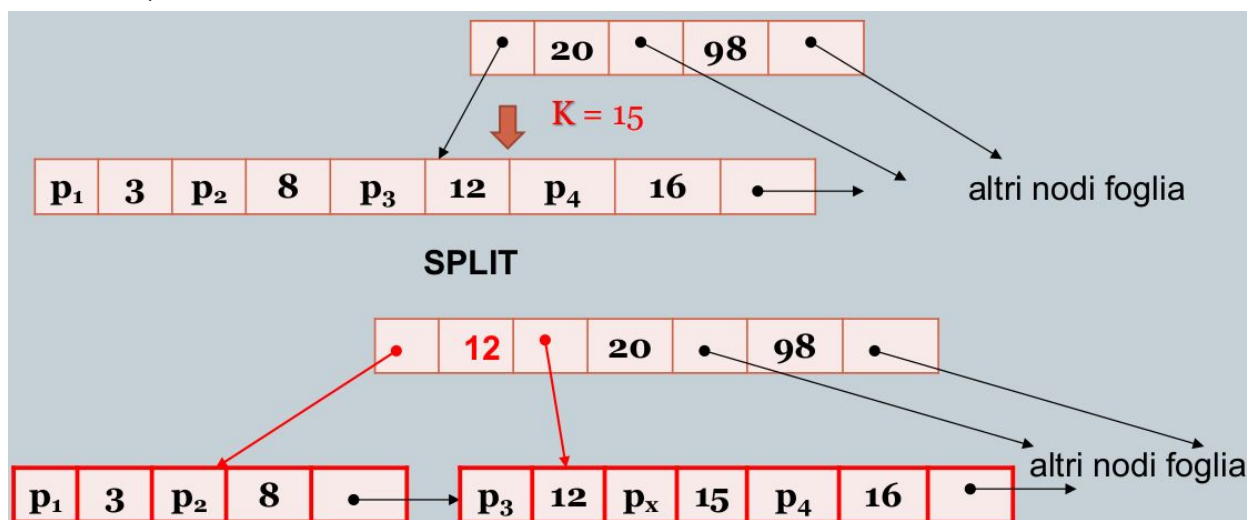
- Ricerca del nodo foglia NF dove il valore K va cancellato.
- Cancella K da NF, insieme al suo puntatore.



- Se dopo la cancellazione, viene violato il vincolo di riempimento minimo di NF, è necessario effettuare un MERGE di NF.

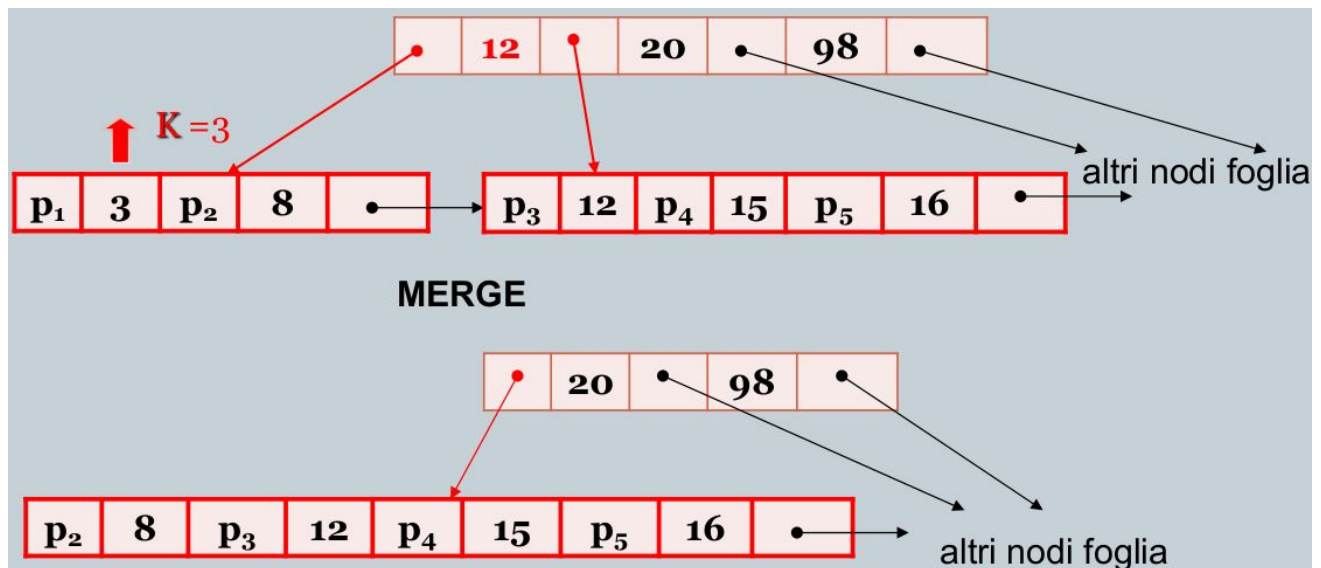
SPLIT di un nodo con n valori:

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



MERGE di un nodo con $\lceil (n-1) / 2 \rceil - 1$ valori chiave:

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



Profondità dell'albero

Il **costo di una ricerca**, in termini di accesso a memoria secondaria, è pari al **numero di nodi acceduti** nella ricerca stessa. Tale numero è pari alla profondità dell'albero, in funzione del numero di #valoriChiave:

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

Strutture ad accesso calcolato

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

$$h: K \rightarrow B$$

K: dominio delle chiavi, B: dominio degli indirizzi

Esecuzione concorrente di transazioni

Per avere **prestazioni** accettabili, un DBMS deve eseguire le **transazioni in modo concorrente**. Senza controllo però si possono generare **anomalie**, sono quindi necessari dei **sistemi per il controllo dell'esecuzione concorrente**.

Notazione:

- t_i = transazione di indice i .
- $r_i(x)$ = lettura della transazione t_i su x .
- $w_i(x)$ = scrittura della transazione t_i su x .

Anomalie di esecuzione concorrente

• Lettura sporca:

t_1 : bot $r_1(x)$; eot

t_2 : bot $r_2(x)$; $x = x+1$; $w_2(x)$; ... abort;

Operazioni eseguite da t_1	Segmento di memoria centrale di t_1	Memoria secondaria [buffer]	Segmento di memoria centrale di t_2	Operazioni eseguite da t_2
Stato iniziale		$x=2$	Stato iniziale	
bot		2		
	2	[2]		bot
	2	[2]	2	$r_2(x)$
	2	[2]	3	$x = x + 1$
	2	[3]	3	$w_2(x)$
$r_1(x)$	3	[3]		
eot		2		abort

Lettura sporca

⇒ t_1 legge un valore che è frutto di una transazione non committata.

• Lettura inconsistente:

t_1 : bot $r_1(x)$; $r_1'(x)$; eot

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

Operazioni eseguite da t_1	Segmento di memoria centrale di t_1	Memoria secondaria [buffer]	Segmento di memoria centrale di t_2	Operazioni eseguite da t_2
Stato iniziale		$x=2$	Stato iniziale	
bot		2		
$r_1(x)$	2	[2]		
	2	[2]		bot
	2	[2]	2	$r_2(x)$
	2	[2]	3	$x = x + 1$
	2	[3]	3	$w_2(x)$
	2	3	3	commit
	2	3		eot
$r_1'(x)$	3	[3]		
eot		3		

Lettura inconsistente

⇒ la prima lettura di t_1 è diversa dalla seconda lettura, ho quindi una lettura inconsistente.

- **Perdita di aggiornamento:**

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

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

Operazioni eseguite da t_1	Segmento di memoria centrale di t_1	Memoria secondaria [buffer]	Segmento di memoria centrale di t_2	Operazioni eseguite da t_2
Stato iniziale		$x=2$	Stato iniziale	
bot		2		
$r_1(x)$	2 ←	[2]		
$x = x + 1$	3	[2]		bot
	3	[2] →	2	$r_2(x)$
	3	[2]	3	$x = x + 1$
	3	[3] ←	3	$w_2(x)$
	3	3	3	commit
	3	3		eot
$w_1(x)$	3 →	[3]		
commit	3	3		
eot		3		

⇒ il valore di x alla fine è **3 invece di 4**, come se la transazione t_1 non fosse mai stata eseguita.

- **Aggiornamento fantasma:**

Risorse: x, y, z

Vincoli: $x+y+z = 100$

t_1 : bot $r_1(y)$; $r_1(x)$; $r_1(z)$; $s = x+y+z$; eot

t_2 : bot $r_2(y)$; $y = y+10$; $r_2(z)$; $z = z-10$; $w_2(y)$; $w_2(z)$ commit; eot

Operazioni eseguite da t_1	Segmento di memoria centrale di t_1	Memoria secondaria [buffer]	Segmento di memoria centrale di t_2	Operazioni eseguite da t_2
Stato iniziale		$(x,y,z)=20,20,60$	Stato iniziale	
bot		20,20,60		
$r_1(y)$	-20,- ←	20,[20],60		
	-20,-	20,[20],60		bot
	-20,-	20,[20],60 →	-20,-	$r_2(y)$
	-20,-	20,[20],60	-30,-	$y = y + 10$
	-20,-	20,[20],[60] →	-30,60	$r_2(z)$
	-20,-	20,[20],[60]	-30,50	$z = z - 10$
	-20,-	20,[30],[60] ←	-30,50	$w_2(y)$
	-20,-	20,[30],[50] ←	-30,50	$w_2(z)$
	-20,-	20,30,50	-30,50	commit
	-20,-	20,30,50		eot
$r_1(x)$	20,20,- ←	[20],30,50		
$r_1(z)$	20,20,50 ←	[20],30,[50]		
$s=x+y+z$	20,20,50	[20],30,[50]		
eot		20,30,50		

⇒ Al termine di t_1 , **s vale 90, ho violato il vincolo.**

Schedule

è una sequenza di **operazioni** di **lettura** e **scrittura** eseguite **da diverse transazioni concorrenti**.

- Per evitare anomalie, si stabilisce di **accettare** solo gli **schedule equivalenti a uno schedule seriale**.

Schedule seriale: le operazioni di ogni transazione compaiono in sequenza senza essere inframmezzate da operazioni di altre transazioni.

- Esempio

$s_1: r_1(x) \ r_2(x) \ w_2(x) \ w_1(x) \quad = \text{NON seriale!}$
 $s_2: r_1(x) \ w_1(x) \ r_2(x) \ w_2(x) \quad = \text{SERIALE!}$

Schedule serializzabile: schedule equivalente a uno schedule seriale, cioè che produce gli stessi effetti sulla base di dati rispetto a uno schedule seriale.

View-equivalenza: due schedule seriali S_1 e S_2 si dicono view-equivalenti se possiedono le stesse relazioni LEGGE_DA e scritture finali.

- **LEGGE_DA:** dato uno schedule S si dice che un'operazione di lettura $r_i(x)$, che compare in S , LEGGE_DA un'operazione di scrittura $w_j(x)$, che compare in S , se $w_j(x)$ precede $r_i(x)$ in S e non ci è alcuna operazione $w_k(x)$ tra le due.
- **Scritture finali:** $w_i(x)$ è una scrittura finale se è l'ultima scrittura della risorsa x in S .

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

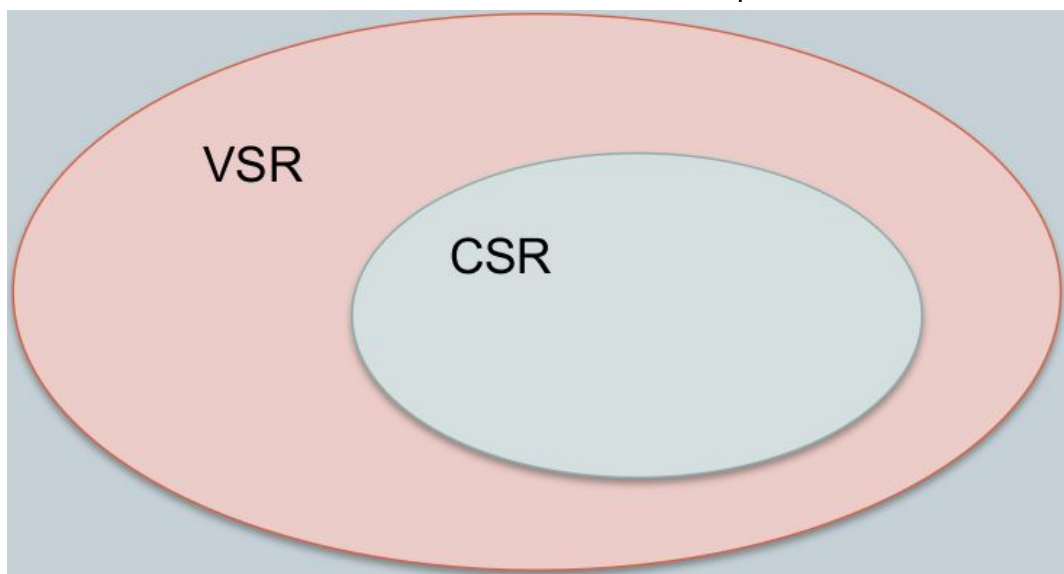
Conflict-equivalenza: due schedule S_1 e S_2 sono conflict-equivalenti ($S_1 \approx_c S_2$) se possiedono le stesse operazioni e gli stessi conflitti.

- In uno schedule S se una **coppia di operazioni** (a_i, a_j) rappresentano un **conflitto** se:
 - $i \neq j$: appartengono a **transazioni diverse**.
 - entrambe **operano** sulla **stessa risorsa**.
 - **almeno** una di esse è **un'operazione di scrittura**.
 - a_i compare in S **prima** di a_j

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

Uno schedule S è **conflict-serializzabile** \Leftrightarrow il suo grafo è **aciclico**.

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



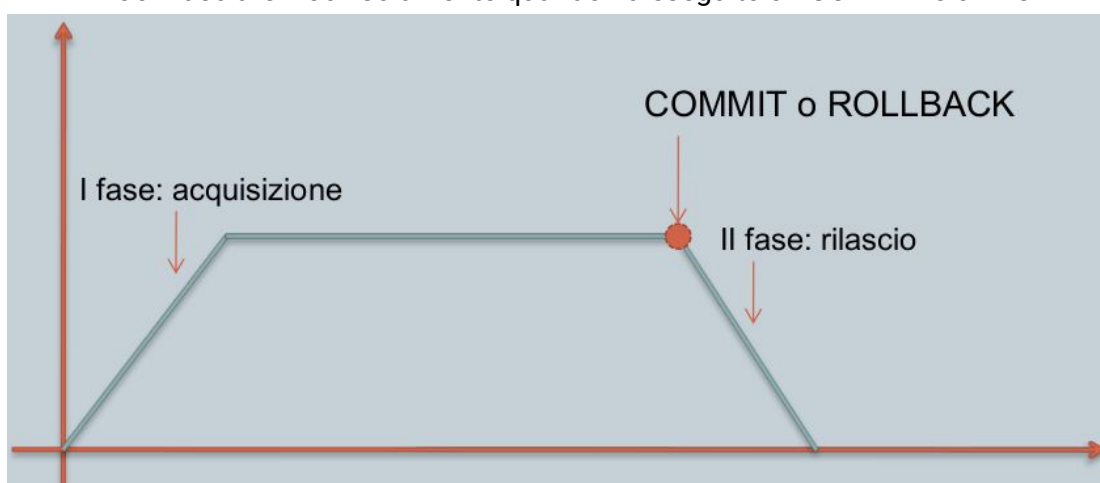
Tecniche applicate nei DBMS

Siccome gli algoritmi **VSR e CSR non sono applicabili nei sistemi reali**, è necessario utilizzare sistemi che non necessitano di conoscere l'esito delle transazioni:

- **TS buffer**: timestamp con scritture bufferizzate.
 - **Meccanismo** per la gestione dei lock: si basa sull'introduzione di alcune **primitive di locking**
 - **r_lock_k(x)**: richiesta di t_k di un lock condiviso in lettura sulla risorsa x.
 - **w_lock_k(x)**: richiesta di t_k di un lock esclusivo in scrittura sulla risorsa x.
 - **unlock_k(x)**: richiesta di t_k di liberare dal lock la risorsa x.
 - **Regole** per l'utilizzo dei lock:
 - Ogni lettura deve essere preceduta da un r_lock e seguita da un unlock. Sono ammessi più r_lock sulla stessa risorsa.
 - Ogni scrittura deve essere preceduta da un w_lock. No più w_lock sulla stessa risorsa.
 - **Politica di concessione** del lock:
 - Il **gestore della concorrenza** mantiene per ogni risorsa:
 - Stato: { r_lock, w_lock, free }
 - Transazioni in r_lock

Stato di x	LIBERO		R_LOCK		W_LOCK	
Richiesta						
r_lock _k (x)	Esito OK	Operazioni $s(x) = r_lock$ $c(x) = \{k\}$	Esito OK	Operazioni $c(x) = c(x) \cup \{k\}$	Esito Attesa	Operaz. -
w_lock _k (x)	Esito OK	Operazioni $s(x) = w_lock$	if $ c(x) =1$ and $k \in c(x)$ then		Esito Attesa	Operaz. -
			Esito OK	Operazioni $s(x) = w_lock$		
			else Attesa	-		
unlock _k (x)	Esito Errore	Operazioni -	Esito OK	Operazioni $c(x) = c(x) - \{k\}$ if $c(x) = \emptyset$ then $s(x) = \text{Libero}$ verifica coda	Esito OK	$c(x) = \emptyset$ $s(x) = \text{Libero}$ verifica coda

- **2PL strict**: locking a due fasi stretto.
 - Per **garantire la serializzabilità**, una transazione:
 - Non deve acquisire altri lock dopo averne rilasciati.
 - Può rilasciare i lock solamente quando ha eseguito un COMMIT o un ROLLBACK.



Deadlock

Si verifica quando due transazioni hanno bloccato delle risorse, e **sono in attesa rispettivamente l'una sulle risorse dell'altra**.

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

$$P(\text{deadlock lunghezza } 2) = \frac{1}{n^2}$$

Risoluzione del deadlock:

- **Timeout:** una transazione in attesa su una risorsa **trascorso il timeout viene abortita**.
- **Prevenzione:**
 - Una transazione acquisisce le risorse a cui deve accedere una volta sola.
 - Ogni transazione all'inizio dell'esecuzione acquisisce un timestamp TS_i . Una transazione può attendere una risorsa bloccata da t_j solo se vale una certa condizione sui TS ($TS_i < TS_j$), altrimenti viene abortita e fatta ripartire con lo stesso TS.
- **Rilevamento:** si esegue un'analisi periodica della tabella dei lock, per rilevare la presenza di un blocco critico (=ciclo nel grafo delle condizioni di attesa).
 - Questa è la tecnica più frequentemente utilizzata nei sistemi reali.

Starvation: se una risorsa x viene costantemente bloccata da una **serie di r_lock**, una transazione in w_lock viene **bloccata** per lungo tempo **fino alla fine della sequenza di letture**.

- Anche se poco probabile nel contesto delle basi di dati, è possibile analizzare la tabella delle relazioni di attesa e verificare da quanto tempo le transazioni stanno attendendo la risorsa e di conseguenza sospendere temporaneamente la concessione di lock in lettura condivisi per consentire la scrittura da parte della transazione in attesa.

Gestione della concorrenza in SQL

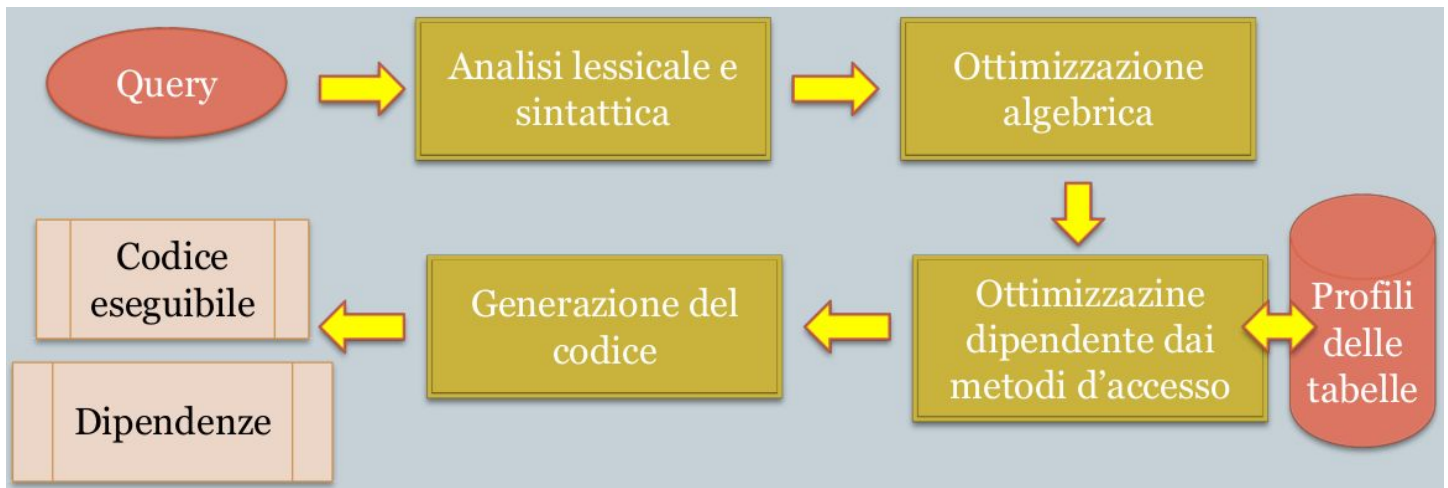
Siccome risulta **oneroso** il sistema di gestione della concorrenza in termini di prestazioni, è **possibile rinunciare del tutto** o in **parte al controllo** per aumentare le prestazioni.

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

- **Serializable:** richiede il **2PL strict** anche per le **letture** e applica il **lock di predicato** per evitare l'inserimento fantasma.
- **Repeatable read:** richiede il **2PL strict** per tutti i lock in **lettura a livello di tupla**, ma non di tabella.
- **Read committed:** **lock condivisi** per le **letture**, ma non 2PL strict.
- **Read uncommitted:** **nessun lock in lettura**.

Ottimizzazione

Ogni **interrogazione** sottomessa al DBMS è **espressa in forma dichiarativa**, per poterla eseguire, è **necessario** trovarne l'**equivalente** in un **linguaggio procedurale** (come l'algebra relazionale). L'espressione va poi **ottimizzata** a livello fisico.



Notazione:

- NP = numero pagine.
- NB = numero buffer.
- R = relazione.

Ottimizzazione dipendente dai metodi di accesso:

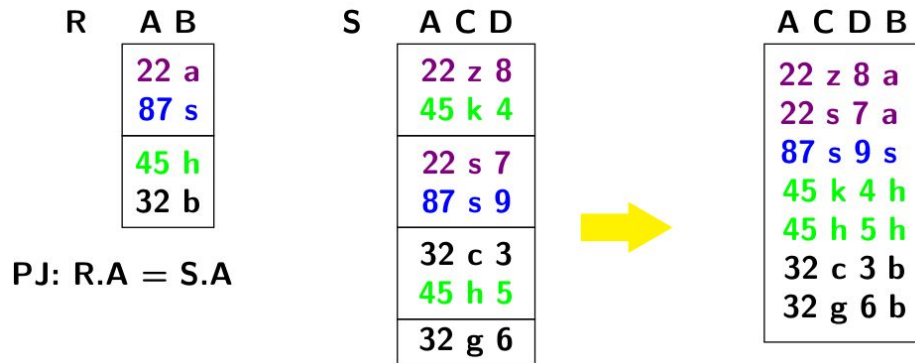
- **Scansione** delle tuple di una relazione (**SCAN**).
 - Diverse varianti:
 - scan + proiezione con duplicati.
 - scan + selezione in base a predicato.
 - scan + inserimento/scansione/modifica.
 - **COSTO**: $NP(R)$
- **Ordinamento** di un insieme di tuple.
 - Casi d'uso:
 - Ordinare il risultato (**ORDER BY**)
 - Eliminare duplicati (**DISTINCT**)
 - Raggruppare tuple (**GROUP BY**)
 - Ordinamento **su memoria secondaria** con "**Z-Way Sort-Merge**":
 - **Sort interno**: si leggono una alla volta le pagine della tabella, le tuple vengono ordinate, e infine la pagina (run) scritta su un file temporaneo.
 - **Merge**: in uno o più passi, vengono fuse le run in una singola.
 - **COSTO**: $2 \times NP \times (\lceil \log_2 NP \rceil + 1)$
- **Accesso diretto** delle tuple attraverso un indice.
 - Selezioni con **condizione atomica di uguaglianza** ($A = v$): **hash o B⁺-tree**.
 - Selezioni con **condizione di range** ($A \geq v1 \text{ AND } A \leq v2$): **B⁺-tree**.
 - Selezioni con condizione costituita da una **congiunzione di condizioni di uguaglianza** ($A = v1 \text{ AND } B = v2$): **indice della condizione più selettiva**.
 - Selezioni con condizione costituita da una **disgiunzione di condizioni di uguaglianza** ($A = v1 \text{ OR } B = v2$): si usano **indici in parallelo** se presenti, **altrimenti, scansione sequenziale**.

- Diverse implementazioni del **join**.
 - **Nested Loop Join**: date 2 relazioni in input R e S tra cui sono definiti i predicati di join PJ, e supponendo che R sia la relazione esterna, algoritmo:

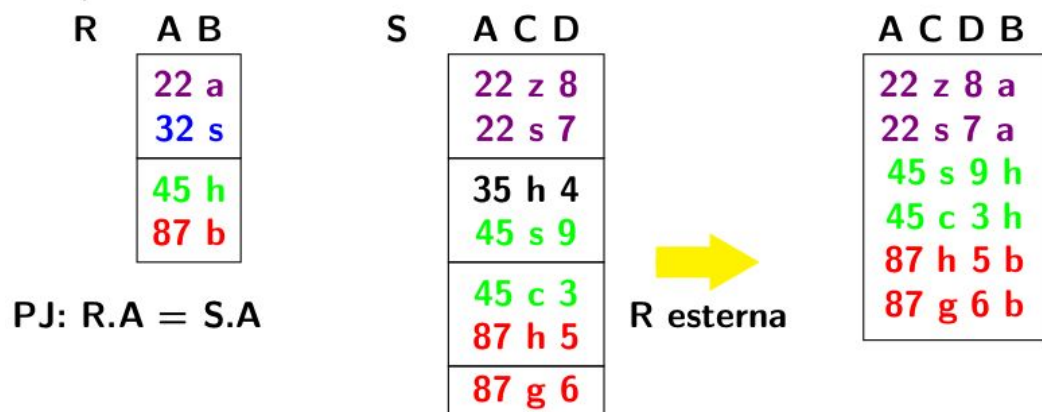
```

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

```



- **COSTO**: Dipende dai buffer a disposizione:
 - Se 1 buffer sia per S, sia per R, allora = $NP(R) + NR(R) * NP(S)$
 - Se disponibile un buffer anche per NP(S) = $NP(R) + NP(S)$
 - con indice B⁺ - tree = $NP(R) + NR(R) * (index_deep + NR(S) / VAL(A, S))$
accessi a memoria secondaria, con VAL(A, S) valori distinti dell'attributo A in S.
- **Merge Scan Join**: applicabile quando entrambi gli insiemi di tuple in input sono ordinati sugli attributi di join e vale **almeno una delle condizioni** per R o S:
 - La **relazione** è **fisicamente ordinata sugli attributi di join** (file sequenziale)
 - **Esiste un indice che consente la scansione ordinata** delle tuple.



- **COSTO**:
 - Se si accede sequenzialmente = $NP(R) + NP(S)$
 - Altrimenti = $NR(R) + NR(S)$
- **Hash-based Join**: applicabile **solo se in presenza di equi join**, è molto vantaggioso su relazioni molto grandi. Si basa sul principio che se $H(t_R \cdot J) = H(t_S \cdot J)$, **forse** $t_R = t_S$.
 - **COSTO**: $NP(R) + NP(S)$

Scelta finale del piano di esecuzione:

1. scrivere l'**espressione ottimizzata in algebra relazionale**.
2. generare tutti i possibili **piani di esecuzione**.
3. stimare gli **accessi a memoria secondaria**.
4. scegliere l'**albero col costo minore**.

Interazione tra **base di dati e applicazioni**:

- **Cursore**: API messa a disposizione del DBMS, risultati rappresentati come cursori (iteratori su tuple).
- **Cursore + API Standard**: ulteriore astrazione sul DBMS data dalla libreria (JDBC, ODBC).
 - JDBC (Java DataBase Connectivity): libreria Java per interagire coi DBMS; è necessario il produttore del DBMS abbia fornito un driver JDBC, un modulo sw che traduce i metodi della libreria in comandi SQL. I passi di interazione sono:
 - caricare il driver JDBC adatto al DBMS
 - definire un URL di connessione
 - stabilire la connessione
 - creare l'oggetto statement per sottoporre i comandi al DBMS
 - scrivere e inoltrare la query
 - se presente, processare il risultato della query
 - chiudere la connessione
 - Possiamo anche eseguire transazione: è necessario disattivare l'auto-commit
- **ORM**: persistenza sulla base di dati di istanze di oggetti, non viene scritto codice SQL.
 - Il gestore della persistenza si interpone tra applicazione e DB con la prima che vede il DB come un insieme di oggetti persistenti. Successivamente in una classe vengono stabilite le corrispondenze oggetti - tabelle

PROS	CONS
<ul style="list-style-type: none">● il gestore della persistenza genera parte dell'SQL● possiamo navigare tra gli oggetti	<ul style="list-style-type: none">● interrogazioni più complesse● interrogazioni ad alta cardinalità rischiano di saturare la memoria● riferimenti tra tabelle possono far caricare tutti il db in memoria

Tecnologie NoSQL

Esistono diversi sistemi **non basati su SQL** che permettono di gestire meglio **enormi quantità di dati**. I principali sistemi sono:

- **Key-Value**: rappresentazione del dato come valore in una **mappa** (map reduce). Molto **usato** per il **clustering**.
- **Document-Store**: i dati sono oggetti con **struttura complessa (documenti)**, **senza** uno **schema fisso**, e con **indici secondari su attributi comuni**. **Non** presenta **ridondanza** (può essere positivo o negativo) ed è utilizzato per il clustering (MongoDB, CouchDB).
- **Extensible-Record-Store**: collezioni di **record** (tabelle), nidificate a **struttura variabile** (BigTable, HBase, HyperTable).
- **Graph-Store**: rappresentazione dei dati tramite **grafi**.

Document Store

- **Dati** = collezioni di oggetti, con il supporto dell'encapsulation (nidificazione dei dati).
- Riferimenti o **nidificazione** permettono di rappresentare i **legami tra documenti**. I riferimenti non sono diretti ma avvengono attraverso l'uso di id:

Listing 1: Student Document

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

Listing 2: Exam Document

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

Listing 3: Contact Document

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

- **Approccio normalizzato:** no ridondanza ⇒ update più semplici
- **Approccio embedded:** update atomico ⇒ entità più piccole
- La maggior parte delle **interrogazioni** sul DB avvengono grazie all'**utilizzo di FIND**. Questo comando cerca in un documento; **permette** la **ricerca** anche sui **dati incapsulati** (anche determinando **condizioni**) e la **proiezione di valori**.

```
#trova gli studenti con cognome = Rossi
db.studenti.find({cognome: "Rossi"})

#trova gli esami con voto = 25
db.studenti.find({esami.voto: 25})

#trova gli esami con voto > 22
db.studenti.find({esami.voto: $gt: 22})

#trova il cognome degli studenti di nome Mario
db.studenti.find({nome:"Mario"}, {cognome:1})
```

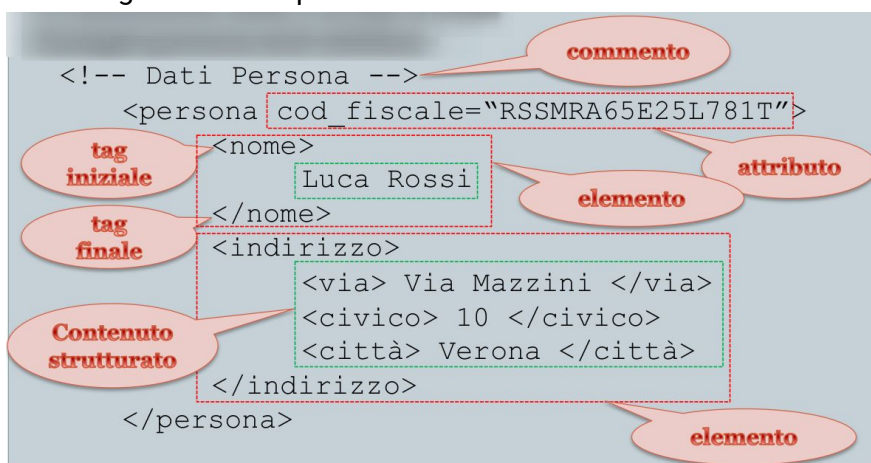
XML

Linguaggio di **markup**, utilizzato in diversi contesti (pagine web, scambio dati tra applicazioni, etc)

- Insieme non fisso di **tag**, posso essere personalizzati.
- Descrizione del **contenuto informativo** del documenti.
- Usato in molti **domini diversi**.

Struttura base:

- Ogni documento deve avere **una sola radice**.
- L'**annidamento** dei tag **deve** essere **rispettato** (Tag aperto ⇒ Tag chiuso).
- I **valori** degli **attributi** devono essere specificati **tra virgolette**.
- Ogni **elemento** può essere **corredato di attributi**.

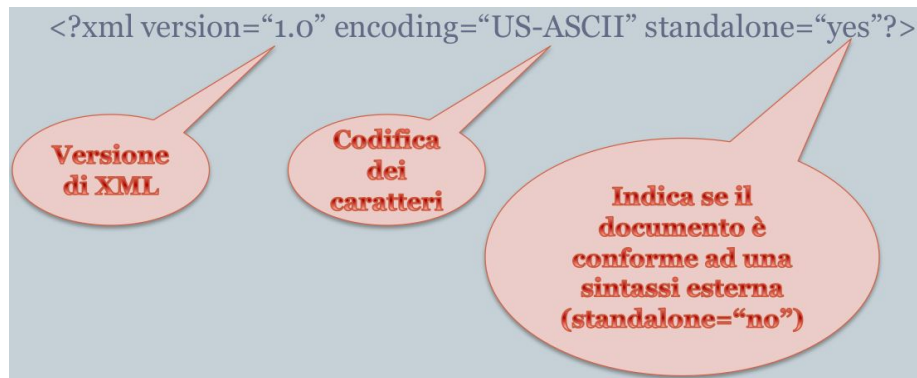


```
<titolo>  
  Essential XML:  
  Oltre il Markup  
</titolo>  
  
<autore>  
  Don Box  
</autore>  
<autore>  
  Aaron Skonnard  
</autore>  
<autore>  
  John Lam  
</autore>  
<casa editrice>  
  Addison-Wesley  
</casa editrice>
```

Elementi vs Attributi:

- **Non c'è** una **regola prefissata** sul quando usare attributi o elementi per denotare informazioni in XML.
- **Svantaggi** nell'uso degli attributi:
 - No più valori ⇒ elementi si
 - No a informazioni ulteriormente strutturate ⇒ elementi si
 - Non posso essere facilmente estesi ⇒ elementi si
- ⇒ **Attributi** per le **meta-informazioni** (identificatori, etc).
- ⇒ **Elementi** per le **informazioni**.
- Gli elementi non possono includere:
 - Caratteri di punteggiatura diversi da -, _, e .
 - Virgolette
 - Apostrofi
 - \$
 - %
 - <
 - >
 - Spazi

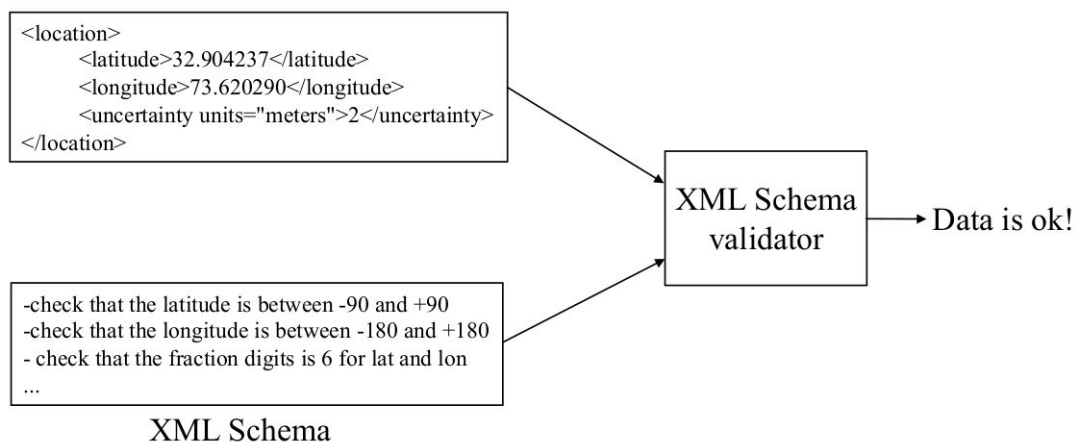
Ogni documento dovrebbe iniziare con:



Validazione: ogni documento XML è valido se è ben formato e rispetta la sintassi specificata nel suo file XSD.

XML Schemas

XSD (XML Schema Definition) è un vocabolario per XML, usato per **esprimere le regole sui dati**.



- **Obiettivi:**
 - specificare la **struttura delle istanze del documento**
 - specificare il **datatype di ogni attributo/elemento**
- Prima degli **XSD** esistevano i **DTD (Document Type Definition)** che però avevano una **sintassi diversa** rispetto all'XML e potevano **rappresentare pochi datatype**.

XSD: COMPONENTI

- **Namespace:** I file schema sono organizzati in namespaces, tutti i componenti descritti appartengono a un namespace target. Ogni namespace deve essere un URI:
 - Esempio di URI:
 - `http://www.ietf.org/rfc/rfc2396.txt`
 - `mailto:mduerst@ifi.unizh.ch`
 - `ftp://ftp.is.co.za/rfc/rfc1808.txt`
- **Referencing:** per riferirsi ad uno schema in un documento XML, dobbiamo aggiungere nell'elemento root i seguenti attributi:
 - `xmlns="URI"`
 - Tutti gli elementi usati provengono da questo namespace
 - Possiamo anche aggiungere un prefisso a questo namespace, che verrà utilizzato per richiamare gli elementi
`xmlns:bk="URI" → <element ref="bk:elementName">`
 - `xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`
 - Definisce che l'attributo `schemaLocation` che usiamo è quello definito da `XMLSchema-instance` come namespace

- `xsi:schemeLocation="URI
XSDFile.xsd"`
Il namespace viene definito dal file XSDFile.xsd

• DataType

- Possiamo dare un nome ad un complexType per riutilizzarlo in futuro


```
<xsd:element name="A" type="T1" />
<xsd:complexType name="T1">
  <xsd:sequence>
    <xsd:element name="B" .... />
    <xsd:element name="C" .... />
  </xsd:sequence>
</xsd:element>
```
- Una dichiarazione di elemento può avere un attributo type o un figlio complexType ma NON entrambi
- I valori di default di `minOccurs` e `maxOccurs` sono `minOccurs = 1` e `maxOccurs = 1`. `maxOccurs`, oltre a valori interi > 0 il valore **"unbounded"** per valori illimitati
- Il date dataType è usato per rappresentare le date nel formato CCYY - MM - DD con:
 - CC ∈ [00, 99], YY ∈ [00, 99]
 - MM ∈ [01, 12]
 - DD
 - DD ∈ [01, 28] if MM = 02
 - DD ∈ [01, 29] if MM = 02 and gYear = leap
 - DD ∈ [01, 30] if MM ∈ {04, 06, 09, 11}
 - DD ∈ [01, 31] otherwise
- Possiamo creare dataType personalizzati tramite l'uso di regular expressions


```
<xsd:simpleType name="ISBNType" />
<xsd:restriction base="xsd:string">
  <xsd:pattern value="\d{1}-\d{5}-\d{3}-\d{1}" />
  <xsd:pattern value="\d{1}-\d{3}-\d{5}-\d{1}" />
  <xsd:pattern value="\d{1}-\d{2}-\d{6}-\d{1}" />
</xsd:restriction>
</xsd:simpleType>
```
- La differenza tra simpleType e complexType è che il primo viene utilizzato per definire un nuovo tipo, mentre il secondo per definire una struttura contenente diversi tipi.
- Quando creiamo un nuovo simpleType, lo costruiamo a partire da un tipo primitivo definito da base e specificando diverse varianti dello stesso, le facets. I tipi primitivi sono:

• string	• gMonth	• ID
• boolean	• NOTATION	• IDREF
• decimal	• unsignedInt	• ENTITY
• float	• normalizedString	• integer
• double	• token	• nonPositiveInteger
• duration	• language	• hexBinary
• dateTime	• IDREFS	• base64Binary
• time	• ENTITIES	• anyURI
• date	• NMTOKEN	
• gYearMonth	• NMTOKENS	
• gYear	• Name	
• gMonthDay	• NCName	
• gDay	• QName	

Ogni dataType ha determinate facets: ad esempio, il tipo string ha come facets length, minLength, maxLength, pattern, enumeration e whitespace. Combinandoli posso specificare come deve essere il nuovo tipo:

```
<xsd:simpleType name="shape" />
<xsd:restriction base="xsd:string">
  <xsd:enumeration value="circle" />
  <xsd:enumeration value="triangle" />
  <xsd:enumeration value="square" />
</xsd:restriction>
</xsd:simpleType>
```

- A parte patterns ed enumerations che sono in OR, tutte le facets sono in AND tra loro
- Possiamo creare un simpleType partendo da un sympleType

```
<xsd:simpleType name="EarthSurfaceElevation">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="-1290" />
    <xsd:maxInclusive value="29035" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="BostonAreaSurfaceElevation">
  <xsd:restriction base="xsd:EarthSurfaceElevation">
    <xsd:minInclusive value="0" />
    <xsd:maxInclusive value="120" />
  </xsd:restriction>
</xsd:simpleType>
```

- Definiti di dataType possiamo creare degli elementi che li utilizzino

```
<xsd:element name="elevation" type="EarthSurfaceElevation"/>
```

- **Attributi:** non hanno elementi figli e vengono rappresentati in linea nell'XML

```
<xsd:attribute name="Category" use="required">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="autobiography"/>
      <xsd:enumeration value="non-fiction"/>
      <xsd:enumeration value="fiction"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>
```

Cosa è più facile trovare all'esame:

- Tipi, primitivi e derivati, soprattutto **string**, **int**, **dateTime**, **gYear**, **gDay**, **ID**, etc...
- Attributi obbligatori: use="**required**"
- Più ripetizioni di un nodo con maxOccurs="**unbounded**"
- Per riferirsi a un altro elemento da un altro elemento: ref="**TargetElement**"
- Enumerazione dei valori assumibili da un attributo: <xsd:enumeration value="**theValue**" />