



NoiStudiamo

APPUNTI

Basi di Dati

Modello relazionale, progettazione, normalizzazione,
transazioni e organizzazione fisica dei dati

A CURA DI

Carola Aiello

Ottobre 2026

Indice degli argomenti

Gli argomenti principali degli appunti e la pagina in cui iniziano.

1. Base di dati	3
2. Il modello relazionale dei dati	5
3. Algebra relazionale	7
4. Viste (relazioni derivate)	13
5. Algoritmo di Query Optimization	15
6. Progettazione di una base di dati	18
7. Modello Entità Relazione	19
8. Strategie di progetto	27
9. Progettazione logica	30
10. Anomalie e dipendenze funzionali	35
11. Decomposizioni di uno schema	41
12. Forme normali	43
13. Gestione delle transazioni	46
14. Controllo di concorrenza	49
15. Organizzazione fisica dei dati	52
16. Ottimizzazione e progettazione fisica	54

Base di dati

Quando parliamo di base di dati parliamo di un insieme organizzato di dati, che esistono e si evolvono nel tempo, disponibili in una certa struttura per lo svolgimento della propria attività.

All'interno di un'organizzazione distinguiamo due concetti fondamentali: il **sistema organizzativo**, che comprende l'insieme delle *risorse* e delle *regole* che consentono di svolgere in maniera coordinata le attività, e il **sistema informativo**, che ne costituisce una parte e si concentra sulla gestione delle informazioni necessarie al funzionamento della struttura. Mentre il sistema organizzativo ha una visione complessiva, il sistema informativo si occupa nello specifico dei processi informativi, cioè di come i dati vengono raccolti, archiviati e utilizzati.

Le informazioni possono essere gestite in modi diversi: dal linguaggio naturale ai numeri, da disegni e grafici alla carta o alla memoria elettronica. Nei sistemi informatici, però, vengono rappresentate in maniera essenziale tramite **dati**. I dati, da soli, sono grezzi e necessitano di interpretazione per trasformarsi in informazione utile. Per esempio, una stringa con scritto "Mario – 2075" diventa informazione solo se sappiamo che si riferisce a un nome e a un numero di interno telefonico.

I dati hanno un ruolo fondamentale perché sono più stabili rispetto ad altri elementi di un'organizzazione e quindi rappresentano una risorsa strategica. Per gestirli in modo efficace si utilizzano i **DBMS (Database Management System)**, software specializzati che permettono di archiviare, interrogare e manipolare grandi quantità di dati in modo persistente, affidabile e condiviso. Un DBMS fornisce linguaggi potenti di interrogazione, garantisce affidabilità, sicurezza e controllo degli accessi, oltre a efficienza ed efficacia nell'uso delle risorse.

Un concetto fondamentale è quello di **modello dei dati**, un formalismo matematico che permette sia di *descrivere* i dati (attraverso una *notazione*) sia di *manipolarli* (attraverso degli *operatori*).

Descrizioni e rappresentazioni dei dati a livelli diversi permettono l'indipendenza dei dati dalla rappresentazione fisica, ad esempio i programmi fanno riferimento alla struttura a livello più alto (logica), e le rappresentazioni sottostanti possono essere modificate senza necessità di modifica dei programmi.

Si distinguono due tipi principali:

- **Modelli logici**, utilizzati dai DBMS per organizzare i dati (relazionale, gerarchico, reticolare, a oggetti), indipendenti dalle strutture fisiche. Si chiama logico per sottolineare il fatto che le strutture utilizzate da questi modelli, pure essendo astratte, riflettono una particolare organizzazione, spesso utilizzati dai programmi.

- **Modelli concettuali**, che rappresentano i dati in modo più astratto e indipendente dal sistema, come il modello Entità-Relazione, molto utile nelle fasi di progettazione.

Il **modello relazionale** è il più diffuso: utilizza il concetto di tabella (relazione), composta da attributi (colonne) e tuple (righe). Su questo modello si basano le query, che permettono di estrarre e combinare informazioni da una o più tabelle.

Dal punto di vista architetturale, un DBMS è organizzato su più livelli:

- lo **schema logico**: che descrive la struttura dei dati nel modello logico;
- lo **schema interno (fisico)**: che rappresenta la memorizzazione effettiva;
- lo **schema esterno**: che rappresenta viste parziali della base di dati.

Questa stratificazione porta al concetto di **indipendenza dei dati**: modifiche al livello fisico o logico non devono obbligare a riscrivere i programmi o le viste che usano i dati.

Si distinguono i linguaggi usati nei DBMS:

- il **DDL (Data Definition Language)**, per creare e definire schemi e strutture;
- il **DML (Data Manipulation Language)**, per interrogare e modificare i dati.

Il modello relazionale dei dati

Un **modello dei dati** è un insieme di meccanismi di astrazione utilizzati per definire una base di dati, a cui si accompagna un insieme predefinito di *operatori* e di *vincoli di integrità*. Questi meccanismi svolgono un ruolo simile a quello delle strutture dati nei linguaggi di programmazione.

Questo modello fu introdotto nel 1970 e ad oggi è il modello più diffuso, fu implementato nei dbms commerciali a partire dal 1981 poiché garantisce una forte indipendenza dei dati.

La parola **relazione** presenta 3 accezioni:

- **relazione matematica**: come nella teoria degli insiemi (sottoinsiemi di prodotti cartesiani);
- **relazione nel modello relazionale**: cioè un insieme di tuple su determinati attributi;
- la “**relationship**” del modello Entità-Relazione: indica una classe di associazioni tra entità.

Il modello relazionale traduce le relazioni matematiche in strutture non più solo posizionali (basate sull'ordine degli attributi), ma **nominate**: ogni dominio di valori ha un nome (attributo) che ne identifica il ruolo. Da qui nasce il concetto di **schema di relazione**, che associa un nome (es. *Studenti*) a un insieme di attributi con i rispettivi tipi. L'insieme degli schemi e dei vincoli costituisce lo **schema relazionale** (aspetto intensionale), mentre le istanze concrete delle tabelle rappresentano l'**aspetto estensionale**.

Ogni attributo ha un **dominio**, cioè l'insieme dei valori che può assumere. Per garantire la qualità e la coerenza dei dati, si introducono i **vincoli di integrità**, che stabiliscono condizioni da rispettare (ad esempio: i voti devono essere compresi tra 18 e 30; la matricola deve essere unica; i riferimenti devono riguardare corsi esistenti). Un'**istanza valida** di uno schema di relazione è una relazione dello schema che soddisfa tutti i vincoli di integrità.

Un insieme X di attributi di uno schema di relazione R è una **superchiave** dello schema se ogni istanza valida dello schema non contiene due n -uple distinte t_1 e t_2 con $t_1[X] = t_2[X]$, ossia identifica univocamente le tuple di una relazione. Una **chiave** è una superchiave minimale rispetto alla relazione $\subseteq$ (non contiene un'altra superchiave). Una relazione non può contenere n -uple distinte ma uguali. Ogni relazione ha come superchiave l'insieme degli attributi su cui è definita e quindi ha (almeno) una chiave.

La **chiave primaria** è una delle chiavi scelta per un dato schema.

- Non ammette valori NULL
- Notazione: Sottolineatura

Una **chiave esterna** è un attributo, o un insieme di attributi, di una tabella che fa riferimento alla chiave primaria (o a un candidato chiave) di un'altra tabella, garantendo l'integrità referenziale tra le due tabelle. La sua notazione è una linea tratteggiata.

Algebra relazionale

L'algebra relazionale è un insieme di operatori definiti su relazioni (tabelle) che producono come risultato una relazione e possono essere combinati per formare **espressioni complesse** analoghe a query sui database. Possiamo affermare che i diversi operatori dell'algebra relazionale prendono in **input delle relazioni** e restituiscono come **output delle relazioni**. Esistono due tipi di operatori, gli operatori primitivi (o operatori di base) e gli operatori derivati.

Operatori primitivi:

- ridenominazione
- unione
- differenza
- proiezione
- restrizione (o selezione)
- prodotto (cartesiano)

Ridenominazione (δ, ρ)

È un operatore unario e consente di **rinominare** un attributo.

$$\delta_{\text{Matricola} \rightarrow \text{Codice Studente}}(\text{Studente})$$

Unione, differenza, intersezione ($\cup, -, \cap$)

Possiamo applicare questi operatori solo a relazioni che hanno lo stesso insieme di attributi, nello **stesso ordine** e con gli stessi tipi di valori.

$$R \cup S = \{t | t \in R \vee t \in S\}$$

$$R - S = \{t | t \in R \wedge t \notin S\}$$

$$R \cap S = \{t | t \in R \wedge t \in S\}$$

Proiezione (π)

Questo operatore consente di mascherare alcune informazioni agendo verticalmente, ossia **oscurando determinate colonne**.

$$\pi_{A_1, A_2, \dots, A_n}(R) = \{t[A_1, A_2, \dots, A_n] | t \in R\}$$

Se nella relazione R sono presenti duplicati rispetto agli attributi selezionati, la cardinalità di $\pi_{A_1, A_2, \dots, A_n}(R)$ sarà **minore** di quella di R , poiché la proiezione elimina le tuple duplicate.

Selezione (σ)

Questo operatore consente di mascherare alcune informazioni agendo orizzontalmente, ossia **oscurando le righe che soddisfano una condizione logica**.

$$\sigma_{\lambda}(R) = \{t | t \in R \wedge \lambda(t) = \text{vero}\}$$

Dove λ è una formula proposizionale formata a partire da atomi $A \theta B$ utilizzando connettivi $\vee, \wedge$, dove A e B sono attributi di R o costanti e $\theta = \{=, <, >, \neq, \leq, \geq\}$.

$$\sigma_{\text{Age} < 30 \vee \text{Salary} > 4000}(\text{Employers})$$

Prodotto cartesiano ($\times$)

Combina ogni tupla di R con ogni tupla di S .

$$R(A_1, A_2, \dots, A_n) \times S(A_{n+1}, \dots, A_{n+m}) = \{t | t_R \in R \wedge t_S \in S\}$$

Operatori derivati

Gli **operatori derivati** sono operatori ottenuti a partire dagli operatori primitivi, i principali sono:

- intersezione
- join (giunzione)

Join (giunzione) ($\bowtie$)

Questo operatore permette di **combinare tuple da relazioni diverse** sulla base di condizioni di uguaglianza o altre relazioni tra attributi.

Esistono due tipi principali di join:

- Natural join
- Theta join (o Equi join)

Natural Join ($\bowtie$)

Unisce automaticamente le tuple che hanno **attributi comuni con lo stesso nome**.

$$R(XY) \bowtie S(YZ) = \{t | t[XY] \in R \wedge t[YZ] \in S\}$$

Può essere espresso come:

$$R \bowtie S = \pi_{XYZ}(R \bowtie_{Y=Y'} S')$$

Dove S' è ottenuta rinominando gli attributi Y in Y' .

Theta join (o Equi join) ($\bowtie$)

Forma generale del join, un'estensione del Natural join, con una **condizione** F .

$$R \bowtie_F S = \sigma_F(R \times S)$$

Se la condizione F è una congiunzione di uguaglianza ($A_i = A_k$) allora si parla di **Equi join**.

Siano $R(A_1:T_1, \dots, A_n:T_n)$ e $S(A_{n+1}:T_{n+1}, \dots, A_{n+m}:T_{n+m})$ con $\{A_1, \dots, A_n\} \cap \{A_{n+1}, \dots, A_{n+m}\} = \emptyset$. Allora si pone:

$$R \bowtie_{A_i=A_k} S = \{tu | t \in R, u \in S, t.A_i = u.A_k\}$$

La giunzione è derivata perchè si ha:

$$R \bowtie_{A_i=A_k} S = \sigma_{A_i=A_k}(R \times S)$$

L'Algebra relazionale può essere usata per interrogare una base di dati. Una **query** è una funzione che, dato un database (insieme di relazioni), restituisce una **nuova relazione**.

L'algebra relazionale rappresenta la base teorica dei linguaggi di interrogazione come SQL.

Join incompleti

Quando si effettua una join tra due relazioni può accadere che alcuni valori tra gli attributi comuni non coincidano, in questo caso si parla di **join incompleti** e le tuple scartate vengono chiamate **dangling tuple**.

Si possono presentare due casi estremi quando si utilizza una join:

- **join vuoto**, ossia quando non vi sono corrispondenze fra le tuple e il risultato è un **insieme vuoto**;
- **join degenerato** in prodotto cartesiano, ogni tupla di una relazione si combina con quelle dell'altra sfociando così nel **prodotto cartesiano** delle due relazioni.

Outer join (giunzione esterna)

Per mantenere le tuple scartate da una join si utilizza l'**outer join** che pone a NULL le *dangling tuple*. Esistono 3 varianti dell'outer join:

- **left outer join** conserva le tuple non abbinate della relazione sinistra,
$$R \bowtie^{\leftarrow} S = (R \bowtie S) \cup ((R - \pi_{XY}(R \bowtie S)) \times \{Z = NULL\})$$
- **right outer join** conserva le tuple non abbinate della relazione destra, $R \bowtie^{\rightarrow} S = (R \bowtie S) \cup (\{X = NULL\} \times (S - \pi_{YZ}(R \bowtie S)))$
- **full outer join** conserva le dangling tuple di entrambe le relazioni,
$$R \bowtie^{\leftrightarrow} S = (R \bowtie S) \cup ((R - \pi_{XY}(R \bowtie S)) \times \{Z = NULL\}) \cup (\{X = NULL\} \times (S - \pi_{YZ}(R \bowtie S)))$$

la full outer join o giunzione esterna è vista come un'estensione del natural join, arricchita con le tuple uguali a NULL

Il join presenta due **proprietà** importanti, è **commutativo**, ossia l'ordine delle relazioni non cambia il risultato ed è **associativo**, ossia è possibile concatenare più join senza generare ambiguità, è dunque possibile eseguire **join multipli** in sequenza ($R \bowtie S \bowtie T \dots$).

Quando due relazioni **non hanno attributi in comune** il join produce automaticamente un **prodotto cartesiano**: non essendoci condizioni di uguaglianza da rispettare tutte le tuple di R si combinano con tutte le tuple di S . Se invece due relazioni hanno la **stessa lista di attributi** allora il natural join conciderà con la loro **intersezione** poiché restituisce solo le tuple identiche presenti in entrambe.

Semi join (o semi giunzione) ($\bowtie$)

La semi join è un operatore derivato che data una relazione R con attributi XY e una S con attributi YZ , $R \bowtie S$ restituisce le tuple di R che trovano almeno una corrispondenza attraverso una join.

$$R \bowtie S = \pi_{XY}(R \bowtie S)$$

Studenti $\bowtie$ Esami

Nome	Matricola	Indirizzo	Telefono
Mario Rossi	123456	Via Etnea 1	222222
Ugo Bianchi	234567	Via Roma 2	333333
Teo Verdi	345678	Via Torino 3	444444

Corso	Matricola	Voto
Architettura	123456	30
Programmazione	234567	18
Architetture	234567	27

Studenti $\bowtie$ Esami

Nome	Matricola	Indirizzo	Telefono
Mario Rossi	123456	Via Etnea 1	222222
Ugo Bianchi	234567	Via Roma 2	333333

Unione esterna ($\cup^{\leftrightarrow}$)

L'unione esterna estende le relazioni R e S con le colonne mancanti riempite di NULL e ne unisce i contenuti.

$$R \cup^{\leftrightarrow} S = R \times \{Z = NULL\} \cup \{X = NULL\} \times S$$

Esempio di Unione Esterna

A	B	C	D	B	C	D	E
X1	Y	Z	X	Y	Z	X	Y1
X2	Y	Z	X	Y	Z	X	M1
X3	Y	W	X	Y	W	X	Y2
X4	Y	W	X	Y	W	X	M2

R
S

$R \cup^{\leftrightarrow} S$

A	B	C	D	E
X1	Y	Z	X	NULL
X2	Y	Z	X	NULL
X3	Y	W	X	NULL
X4	Y	W	X	NULL
NULL	Y	Z	X	Y1
NULL	Y	Z	X	M1
NULL	Y	W	X	Y2
NULL	Y	W	X	M2

In presenza di **attributi NULL** una condizione di selezione, ad esempio "Età>40" produrrà un risultato inatteso perché non è in grado di stabilire la sua veridicità e arriveremo ad un risultato errato, come in questo caso:

$$\sigma_{Età>30}(Persone) \cup \sigma_{Età\leq 30}(Persone) \neq Persone$$

Per risolvere il problema si possono utilizzare condizioni esplicite come **IS NULL** o **IS NOT NULL**, oppure si può utilizzare la **logica a 3 valori** (vero, falso, sconosciuto) che permette di trattare i NULL, avendo così:

$$\sigma_{Età>30}(Persone) \cup \sigma_{Età\leq 30}(Persone) \cup \sigma_{Età \text{ IS NULL}}(Persone)$$

LOGICA A 3 VALORI

<i>p</i>	<i>q</i>	<i>p and q</i>	<i>p or q</i>	<i>not p</i>
T	T	T	T	F
T	F	F	T	F
T	U	U	T	F
F	F	F	F	T
F	U	F	U	T
U	U	U	U	U

Quoziente (÷)

Il **quoziente** (o divisione) è un operatore derivato che permette di individuare gli elementi di una relazione che sono in relazione con **tutti** gli elementi di un'altra relazione.

$$R \div S = \{t | \{t\} \times S \subseteq R\}$$

Esempio

Matricola	Corso	Voto
123456	DB	30
234567	DB	20
123456	Prog	26
345678	Prog	30

R

Corso
DB
Prog

S

123456	DB
123456	Prog

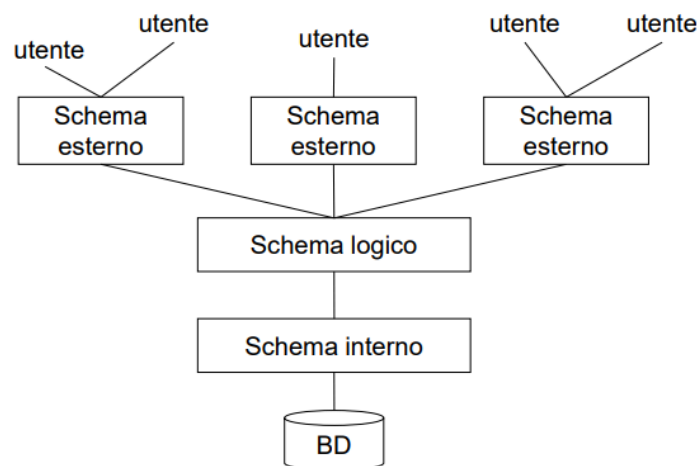
← è contenuto in *R*? sì, allora verrà mostrata in output

Si continua così per tutti gli elementi di R , se non viene soddisfatta almeno una condizione (ossia se la matricola non ha superato almeno uno dei due esami) allora non verrà mostrato in output.

$$R \div S = \pi_X(R) - \pi_X[\pi_X(R) \times S - R]$$

Viste (relazioni derivate)

Una stessa base di dati può essere rappresentata in modi diversi a seconda delle esigenze dell'utente o dell'applicazione. Le **viste** fanno parte dello **schema esterno** ANSI/SPARC per i DBMS, che è suddiviso in **schema interno** (come i dati sono memorizzati), **schema logico** (la struttura concettuale dei dati) e **schema esterno** (la rappresentazione dei dati per singolo utente).



Le viste sono **relazioni derivate**, ossia relazioni il cui contenuto è funzione di altre relazioni definite da **query** (interrogazioni) e possono essere definite su altre derivate. Le **relazioni di base** sono autonome e memorizzate nella base dati.

Un esempio di vista è la relazione **supervisione** definita come:

Impiegato	Reparto
Rossi	A
Neri	B
Bianchi	B

Afferenza

Reparto	Capo
A	Mori
B	Bruni

Direzione

$$SUPERVISIONE = \pi_{Impiegato, Capo}(Afferenza \bowtie Direzione)$$

Esistono due principali categorie di viste:

- **viste materializzate**, ovvero relazioni derivate ma **memorizzate** fisicamente nella base di dati. Il vantaggio è la **rapidità** di accesso, gli svantaggi sono la **ridondanza** e la complessità negli aggiornamenti. Un altro aspetto negativo è il fatto che i DBMS le supportano raramente.
- **viste virtuali**, non vengono memorizzate, non contengono dati propri e il contenuto viene costantemente calcolato al momento dell'interrogazione.

Ad esempio l'interrogazione:

$$\sigma_{\text{Capo}='Leoni'}(\text{Supervisione})$$

viene eseguita come.

$$\sigma_{\text{Capo}='Leoni'}(\pi_{\text{Impiegato}, \text{Capo}}(\text{Afferenza} \bowtie \text{Direzione}))$$

cioè sostituendo la definizione della vista all'interno della query.

Le viste possono essere usate per creare **schermi esterni personalizzati**, in modo che ogni utente vede solo ciò che gli interessa e nel modo in cui gli interessa, sono utili anche per motivi di **sicurezza**, ossia mostra solo le informazioni per cui l'utente è autorizzato.

Sono anche un valido **strumento di programmazione**, poiché permettono di semplificare query complesse e riutilizzando sottoespressioni migliorando la leggibilità e la manutenzione delle query. L'uso delle viste non influisce sull'efficienza delle interrogazioni.

Un esempio di utilizzo può essere trovare gli impiegati che hanno lo stesso capo di Rossi. Senza usare viste la query sarebbe:

$$\pi_{\text{Impiegato}}(((\text{Afferenza} \bowtie \text{Direzione}) \bowtie \delta_{\text{ImpR, RepR} \leftarrow \text{Imp, Reparto}}(\sigma_{\text{Impiegato}='Rossi'}(\text{Afferenza} \bowtie \text{Direzione}))))$$

con la vista diventa invece:

$$\pi_{\text{Impiegato}}(\text{Supervisione} \bowtie \delta_{\text{ImpR, RepR} \leftarrow \text{Imp, Reparto}}(\sigma_{\text{Impiegato}='Rossi'}(\text{Supervisione})))$$

Questo mostra come le viste semplificano la scrittura e la comprensione delle interrogazioni.

Algoritmo di Query Optimization

Due espressioni si dicono **equivalenti** se producono lo stesso risultato. I DBMS sfruttano espressioni equivalenti per ridurre il costo dell'esecuzione per una query.

Per ottimizzare le query bisogna **ridurre la dimensione del risultato intermedio**, cioè il costo dell'operazione. In particolare è fondamentale **anticipare** l'applicazione delle restrizioni e delle proiezioni rispetto alle operazioni di prodotto cartesiano o join.

Supponiamo di voler trovare tutti i professori che hanno dato a Mario Rossi più di 27:

$$\pi_{Professore}(\sigma_{Nome='Mario Rossi' \wedge Voto > 27}(STUDENTI \bowtie ESAMI \bowtie CORSI))$$

Possiamo riscrivere questa query in un altro modo, che risulta essere più efficiente:

$$\pi_{Professore}(\sigma_{Nome='Mario Rossi'}(STUDENTI) \bowtie \sigma_{Voto > 27}(ESAMI) \bowtie (CORSI))$$

In questo modo invece di unire tutte le tabelle (Studenti, Esami, Corsi) e poi filtrare è più efficiente applicare i filtri sulle singole tabelle. Questa tecnica è nota come **Push Selection**, che riduce drasticamente la dimensione delle tabelle intermedie prima che vengano eseguite le operazioni di join.

Per manipolare formalmente le espressioni un algoritmo sfrutta le seguenti regole:

1. Raggruppamento di restrizioni:

$$\sigma_{c(X)}(\sigma_{c(Y)}(R)) = \sigma_{c(X) \wedge c(Y)}(R)$$

2. Commutatività di σ e di π :

- a. $\sigma_{c(X)}(\pi_Y(R)) = \pi_Y(\sigma_{c(X)}(R))$ se $X \subseteq Y$
- b. $\pi_Y(\sigma_{c(X)}(\pi_{XY}(R))) = \pi_Y(\sigma_{c(X)}(R))$ se $X \not\subseteq Y$

3. Anticipazione di σ rispetto a $\times$:

- a. $\sigma_{c(X)}(E \times F) = \sigma_{c(X)}(E) \times \sigma_{c(X)}(F)$ se $X \subseteq attr(E)$
- b. $\sigma_{c(X) \wedge c(Y)}(E \times F) = \sigma_{c(X)}(E) \times \sigma_{c(Y)}(F)$ se $X \subseteq attr(E)$ e $Y \subseteq attr(F)$

4. Raggruppamento di proiezioni:

$$\pi_X(\pi_Y(E)) = \pi_X(E) \text{ se } X \subseteq Y$$

5. Eliminazione di proiezioni superflue:

$$\pi_X(E) = E \text{ se } X = \text{attri}(E)$$

6. Anticipazione della π rispetto a $\times$:

$$\pi_{XY}(E \times F) = \pi_X(E) \times \pi_Y(F) \text{ se } X \subseteq \text{attr}(E) \text{ e } Y \subseteq \text{attr}(F)$$

L'applicazione di queste regole segue un ordine preciso, si parte dall'ottimizzazione della selezione, finché possibile:

A. Si anticipa σ rispetto a π usando la 2.a

$$\sigma_{c(X)}(\pi_Y(E)) = \pi_Y(\sigma_{c(X)}(E))$$

B. Si raggruppano le restrizioni usando la 1

$$\sigma_{c(X)}(\sigma_{c(Y)}(E)) = \sigma_{c(X) \wedge c(Y)}(E)$$

C. Si anticipa l'esecuzione di σ su $\times$ usando la 3.

Si sposta la selezione più verso il basso nell'albero della query, raggruppando i criteri di filtro e anticipandoli rispetto ai prodotti. Successivamente si cerca di ottimizzare la proiezione finché possibile:

D. Si eliminano le proiezioni superflue usando la 5

$$\pi_X(E) = E \text{ se } X = \text{attr}(E)$$

E. Si raggruppano le proiezioni mediante la regola 4

$$\pi_X(\pi_Y(E)) = \pi_X(E) \text{ se } X \subseteq Y$$

F. Si anticipa l'esecuzione delle proiezioni rispetto al prodotto usando ripetutamente la 2

$$[\pi_Y(\sigma_{c(X)}(\pi_{XY}(E)))] = \pi_Y(\sigma_{c(X)}(E)) \text{ se } X \not\subseteq Y] \text{ e la 6 [Anticipazione della } \pi \text{ rispetto a } \times].$$

Si eliminano le proiezioni superflue e si anticipano quelle necessarie rispetto ai prodotti, per evitare di trascinare attributi non richiesti durante i calcoli intermedi.

Infine è importante ricordare che la selezione è **distributiva** rispetto all'unione ($\cup$) e alla differenza ($-$), mentre la proiezione è distributiva rispetto all'unione ma **non** rispetto alla differenza.

- $\sigma_c(R_1 \cup R_2) = \sigma_c(R_1) \cup \sigma_c(R_2)$
- $\sigma_c(R_1 - R_2) = \sigma_c(R_1) - \sigma_c(R_2)$
- $\pi_X(R_1 \cup R_2) = \pi_X(R_1) \cup \pi_X(R_2)$

Le operazioni booleane sui filtri possono inoltre essere tradotte in operazioni tra insiemi, dove la congiunzione ($\wedge$) corrisponde all'intersezione ($\cap$) e la disgiunzione ($\vee$) corrisponde all'unione ($\cup$).

Progettazione di una Base di Dati

Come ogni ciclo di vita di un sistema informativo, il progetto di una base di dati consiste nei seguenti 5 passi:

1. **Raccolta ed analisi dei requisiti:** individuare l'organizzazione e funzionalità del sistema;
2. **Progettazione:** individuare l'organizzazione e la struttura della base di dati, schematizzare le operazioni sui dati;
3. **Implementazione;**
4. **Validazione e collaudo;**
5. **Funzionamento e manutenzione;**

Quando progettiamo una base di dati è importante avere una netta separazione delle decisioni relative a cosa rappresentare in una base di dati (prima fase) da quelle relative a come farlo.

Distinguiamo quindi 3 tipi di progettazione:

- **Progettazione concettuale:** tradurre la descrizione informale della realtà, risultato dell'analisi dei requisiti del DB, in uno *schema formale* e completo dovrà essere *indipendente dai criteri di rappresentazione* del DBMS usato.
- **progettazione logica:** traduzione dello schema concettuale in termini di un determinato *modello logico* (ad esempio il modello relazionale) di dati usato dal DBMS che si intende utilizzare. Il risultato è lo *schema logico*, questo comprende anche l'*ottimizzazione* della rappresentazione in funzione delle operazioni eseguite.
- **progettazione fisica:** si completa lo schema logico con la *specificazione dei parametri fisici* di memorizzazione dei dati. Si produce lo *schema fisico* che fa riferimento ad un certo *modello fisico* dei dati che dipende dal DBMS scelto.

Analisi dei requisiti

Durante questa fase bisogna:

- individuare i requisiti hardware e software;
- individuare le **proprietà** e le **funzionalità** del sistema;
- produrre una descrizione dei *dati* in linguaggio naturale e come sono coinvolti nelle varie *operazioni*,

Tutto questo attraverso uno **studio di fattibilità** e **piano di sviluppo** dove vengono specificati i costi in termini di budget, priorità e tempi di realizzazione.

Per eseguire l'analisi dei requisiti bisogna seguire questa procedura:

1. Analizzare le descrizioni dei dati per evitare ambiguità (omonimie, conflitti e similitudini);
2. Controllare insieme ai responsabili di settore le frasi relative alle varie categorie di dati e alle operazioni che li coinvolgono;
3. Costruire a partire da queste frasi verificate un **glossario di termini**, tipicamente contiene la descrizione, l'elenco dei sinonimi e l'elenco dei termini collegati;
4. Verificare la **completezza** controllando che tutti gli aspetti vengano considerati;
5. Verificare la **consistenza** delle specifiche controllando che tutti i termini sono stati definiti, tutti i termini compaiono in delle operazioni e le operazioni fanno riferimento a termini definiti.

Esempio

TERMINE	DESCRIZIONE	SINONIMI	LEGAME
fornitore	p. iva, denom., indirizzo, num. tel.		prodotto
cliente	p. iva, denom., indirizzo, num. tel.	acquirente	prodotto
prodotto	codice, nome, genere....	articolo voce	fornitore scaffale cliente
Scaffale	supporto numerato	ripiano (incertezza)	operaio prodotto
operaio	dati anagrafici, matricola, qualifica	addetto magazziniere	scaffale

glossario

Modello Entità-Relazione

Il modello Entità-Relazione (E-R) è un **modello concettuale di dati** che contiene alcuni costrutti atti a descrivere la realtà in maniera semplice, questi costrutti sono:

- **Entità:**
 - entità: classi di oggetti che hanno delle proprietà;
 - istanza di entità: oggetto in memoria della quale si vogliono registrare fatti specifici, viene identificata univocamente;
- **Relazioni o Associazioni:**
 - relazioni: legami logici fra due o più entità dell'applicazione;
 - istanza di relazione: fatto che descrive un'azione e quindi un legame tra istanze di entità;
- **Proprietà o Attributi:** fatti che descrivono le caratteristiche delle istanze di entità, le proprietà delle istanze assumono *valori*;
- **Cardinalità;**
- **Identificatori;**
- **Generalizzazioni;**
- **Sottoinsiemi;**

Notazione grafica modello E-R

Il modello E-R usa simboli grafici per favorire l'immediatezza della comprensione:

- Un'entità viene rappresentata con un *quadrato* con al suo interno il nome di quella identità:



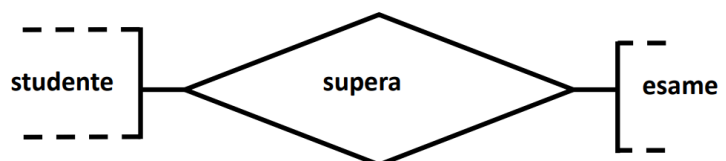
esempi:



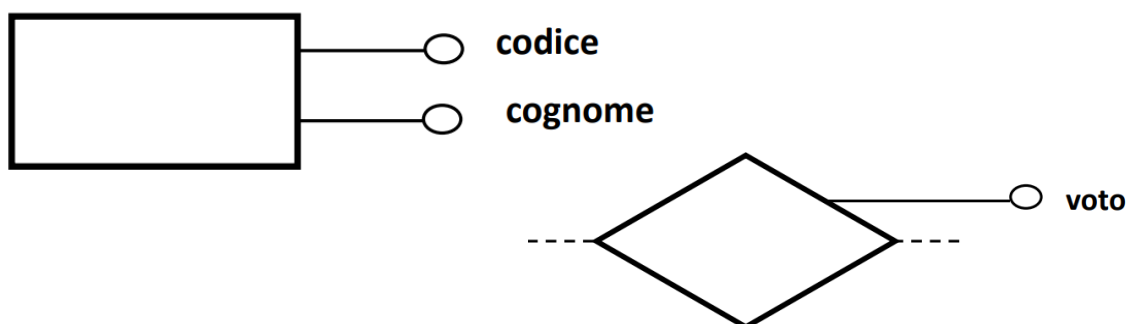
- Le Associazioni o Relazioni vengono rappresentate con un *rombo* con al suo interno il nome dell'associazione:



esempio:



- Le Proprietà o gli Attributi escono dalle entità/associazioni sottoforma di *pallini* che specificano il nome degli attributi:

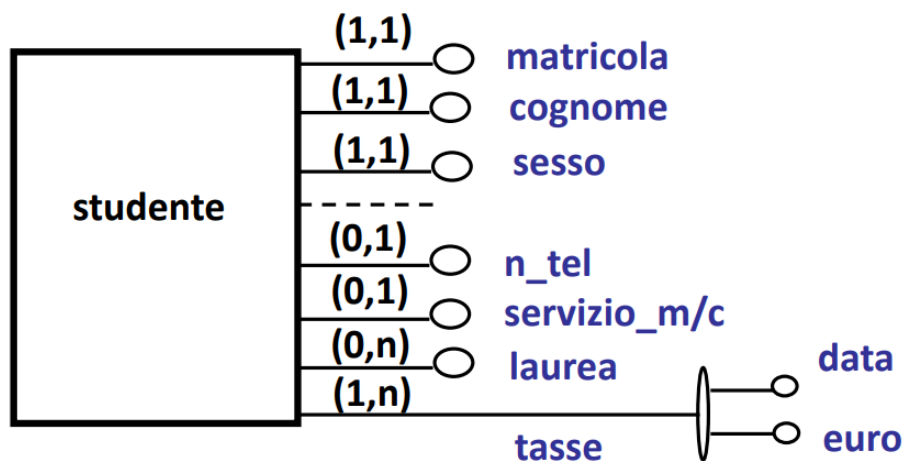


Prima dello schema E-R vero e proprio si creano i cosiddetti **schemi scheletro** che descrivono una prima struttura di massima dello schema senza indicazioni sul tipo delle entità e delle associazioni. Descrivono in generale i collegamenti tra le entità di interesse e le associazioni che le legano.



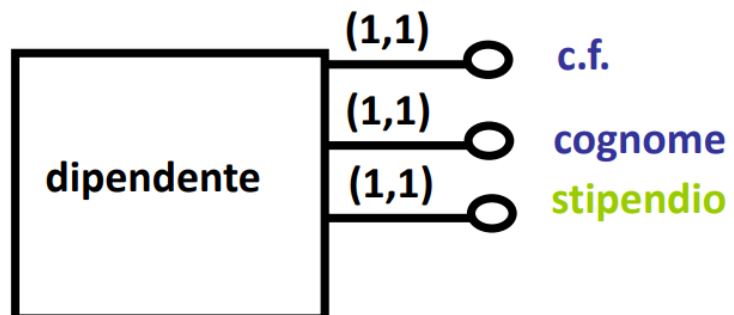
Le Proprietà sono di diversi tipi, il simbolo (n,m) esprime la **cardinalità** minima e massima della proprietà:

- **(1,1)** indica che l'attributo è obbligatorio (minimo 1) e singolo (massimo 1);
- **(1,n)** indica che l'attributo è obbligatorio (minimo 1) e multipli (massimo n valori, illimitati);
- **(0,1)** indica che l'attributo è opzionale (minimo 0) e singolo (massimo 1);
- **(0,n)** indica che l'attributo è opzionale (minimo 0) e multiplo (massimo n valori);

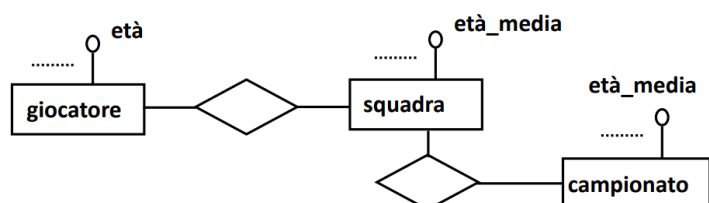


Proprietà speciali

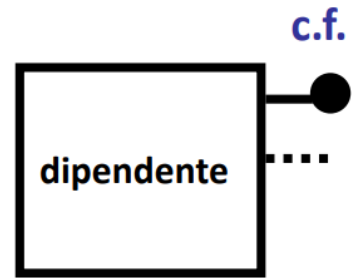
- **Proprietà calcolate (o derivate)**: il valore di questo tipo di proprietà è calcolato con un algoritmo specifico a differenza delle proprietà *esplicite*. La regola di calcolo va espressa a parte usando un linguaggio di specifica, es. *stipendio = salario_giornaliero * presenze*



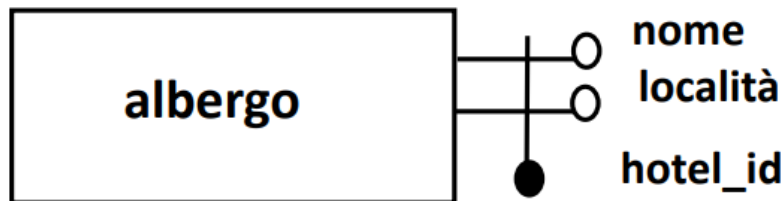
- **Proprietà di sintesi**: esprime valori medi, massimi e minimi, sono anche queste delle proprietà calcolate ma che vanno a sintetizzare il contenuto della tabella



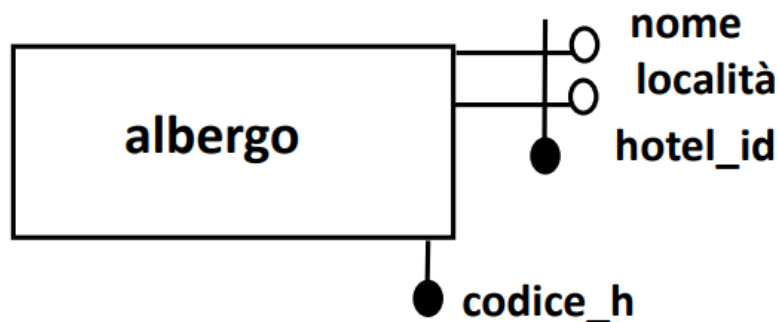
- **Proprietà chiave:** questa proprietà identifica in modo univoco la singola istanza di un'entità, è obbligatoria e può essere composta, come simbolo usa quello dell'attributo base ma con il pallino pieno. In generale non è modificabile.



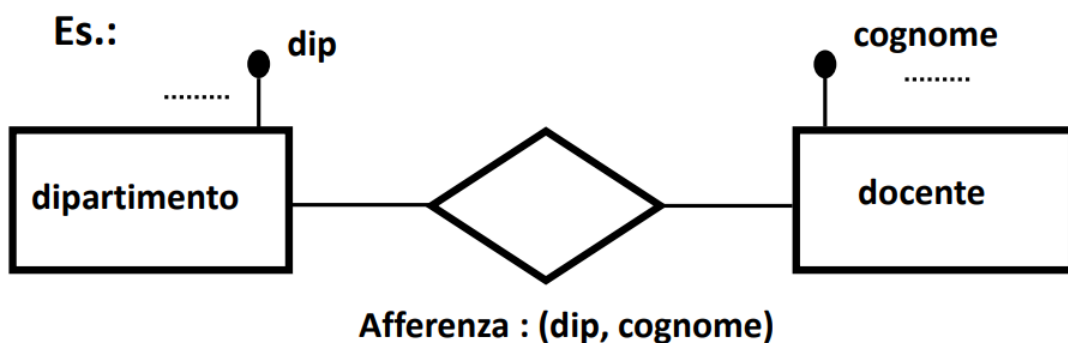
La chiave di un'entità può essere composta



e si possono avere più chiavi in una tabella che vanno ad identificare univocamente lo stesso record, sono dette **chiavi alternative**



Le Associazioni (o relazioni) **hanno sempre** delle chiavi composte definite come l'insieme delle chiavi primarie delle tabelle partecipanti alla relazione

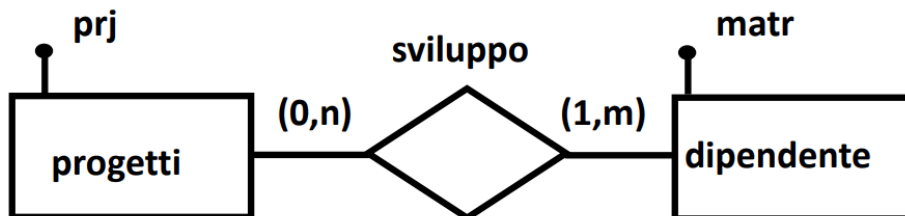


La relazione **eredita sempre** le chiavi dalle entità partecipanti, non si possono definire chiavi nelle associazioni!

Cardinalità

Per cardinalità si intende il numero di volte che data un'istanza di entità deve o può partecipare all'associazione (definite poco prima i vari tipi di cardinalità), vediamo degli esempi:

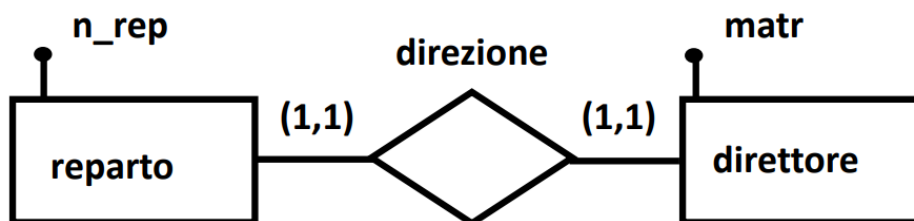
Associazioni N:M



- **Con vincolo:** un dipendente **deve** partecipare ad almeno un progetto (1,m)
- **Senza vincolo:** ad un progetto **possono** partecipare dipendenti, ma può esistere anche un progetto senza dipendenti (0,n)

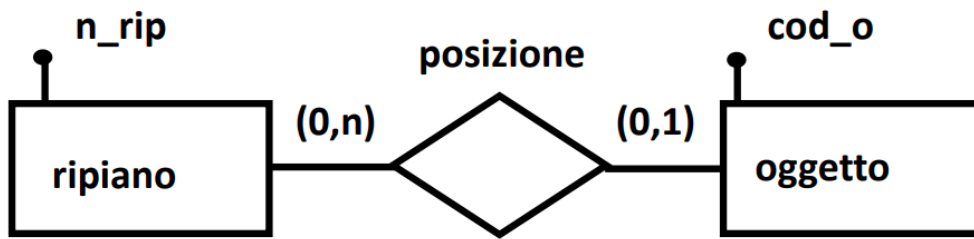
Chiamata anche: **associazione molti a molti**

Associazioni 1:1



- **Con doppio vincolo:**
 - un reparto **deve** avere un direttore ed il direttore è **uno solo** (1,1)
 - un direttore **deve** dirigere uno ed **un solo** reparto (1,1)
- Il vincolo diventerebbe (0,1) se avessimo la generica entità "impiegato" al posto dell'entità "direttore" (alcuni impiegati non dirigono reparti)

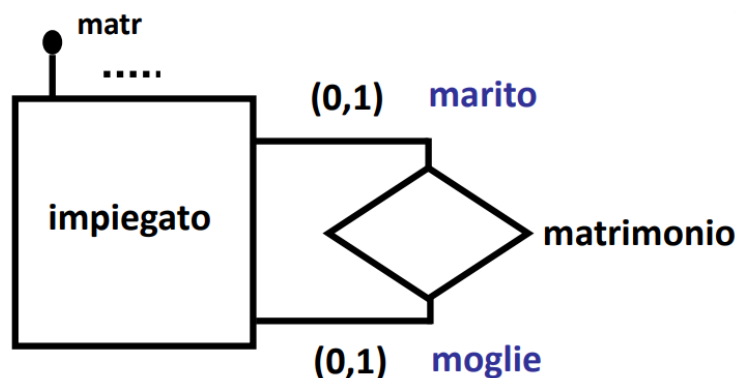
Associazioni N:1



- **Con vincolo:** un oggetto **può** stare su un ripiano, ma su **un solo** ripiano (0,1)
- **Senza vincolo:** su un ripiano **possono** stare n oggetti
- È possibile vincolare la partecipazione, per esempio ponendo $n = 5$, **non più di 5** oggetti per ripiano (0,5)
- Chiamata anche: associazione molti a uno

Auto-Associazioni

Le associazioni che hanno come partecipanti istanze provenienti dalla stessa entità sono dette auto-associazioni o anche unarie o ad anello

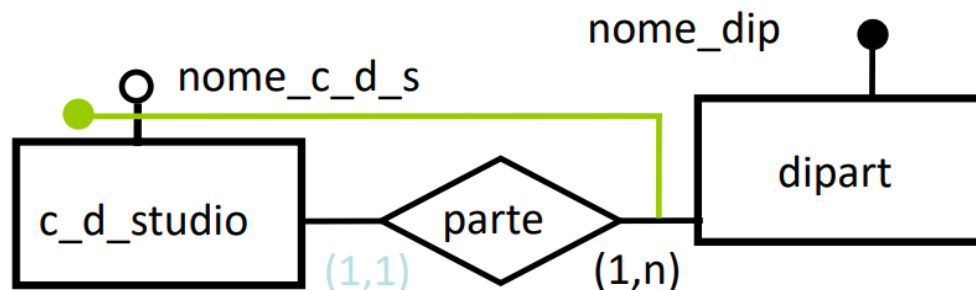


Su ogni ramo può essere riportato il “ruolo” del partecipante all’associazione.

Identificazione esterna

In alcuni casi un'entità può essere identificata da altre ad essa collegato, nell'esempio di seguito il corso di studio è identificato da un nome proprio e da quello del dipartimento che li eroga, ad esempio:

Laurea in informatica del dipartimento di Matematica e Informatica



Le identificazioni esterne avvengono sempre tramite *associazioni binarie* in cui l'entità da identificare partecipa con cardinalità 1,1. Un'identificazione esterna può coinvolgere un'entità che a sua volta è identificata esternamente a patto che non si creino cicli di identificazione.

Gerarchie

Spesso nell'analisi di un settore aziendale può capitare che più entità risultino simili o in casi particolari l'una rappresenta un sotto-tipo dell'altra. Emerge quindi la necessità di evidenziare *sottoclassi* di alcune classi, definendo così una **gerarchia di specializzazione**, ossia il legame logico che esiste tra classi e sottoclassi.

Gerarchia concettuale

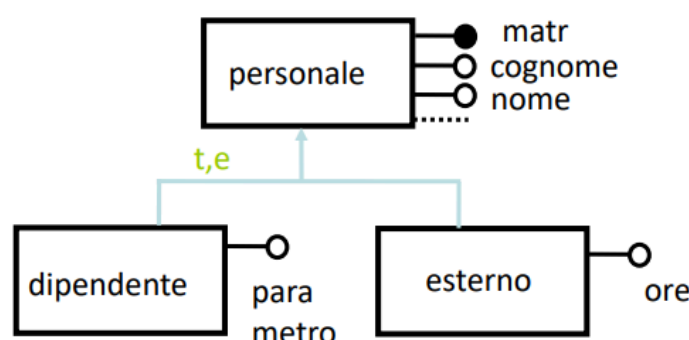
È il legame logico tra un'entità padre E ed alcune entità figlie $E_1, E_2, \dots, E_n$ dove:

- E è la **generalizzazione** di $E_1, E_2, \dots, E_n$;
- $E_1, E_2, \dots, E_n$ sono le **specializzazioni** di E ;
- un'istanza E_k **è** anche istanza di E ;
- un'istanza E **può** essere un'istanza di E_k

Guardiamo questi due esempi:

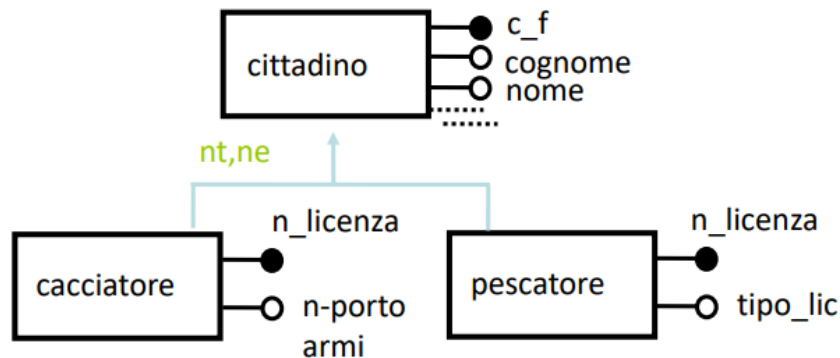
Classificazione del personale

un'azienda si avvale dell'opera di professionisti esterni, quindi il suo personale si suddivide in esterni e dipendenti:



Anagrafe comunale

un comune gestisce l'anagrafe ed i servizi per i suoi cittadini alcuni di questi richiedono i dati relativi alla licenza di pesca e/o di caccia:



- **t** sta per **totale**, ogni istanza dell'entità padre deve far parte di una delle entità figlie (il personale si divide in esterni e dipendenti);
- **nt** sta per **non totale**, le istanze dell'entità padre possono far parte di una delle entità figlie (i pescatori sono un sottoinsieme di cittadini);
- **e** sta per **esclusiva**, ogni istanza dell'entità padre deve far parte di una sola delle entità figlie (un'istanza di personale non può essere sia dipendente che che esterno);
- **ne** sta per **non esclusiva**, ogni istanza dell'entità padre può far parte di una più entità figlie (un cittadino può essere sia pescatore che cacciatore).

Ereditarietà

Le proprietà dell'entità padre non devono essere replicate sull'entità figlia in quanto questa la *eredita*:

- il tipo di **personale** è: (*matricola*, *cognome*, *nome*, *indirizzo*, *data_nascita*);
- il tipo di **dipendente** è: (*matricola*, *cognome*, *nome*, *indirizzo*, *data_nascita*, *parametro*);
- il tipo di **esterno** è: (*matricola*, *cognome*, *nome*, *indirizzo*, *data_nascita*, *ore*);

Le proprietà dell'entità padre fanno parte del *tipo* dell'entità figlia ma non vale il viceversa.

Strategie di progetto

Nei nostri progetti possiamo avere 3 tipi diversi di approcci:

- Top-Down;
- Bottom-Up;
- Generale.

Strategia Top-Down

Le primitive di trasformazione top-down sono regole che partendo da uno schema semplice riescono ad aggiungere dettagli e sono:

- **T1:** si applica quando un'entità descrive *due concetti diversi* legati tra di loro;



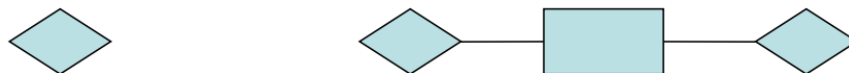
- **T2:** un'entità è composta da **sotto-entità distinte**;



- **T3:** una relazione in realtà descrive due relazioni diverse tra le stesse entità;



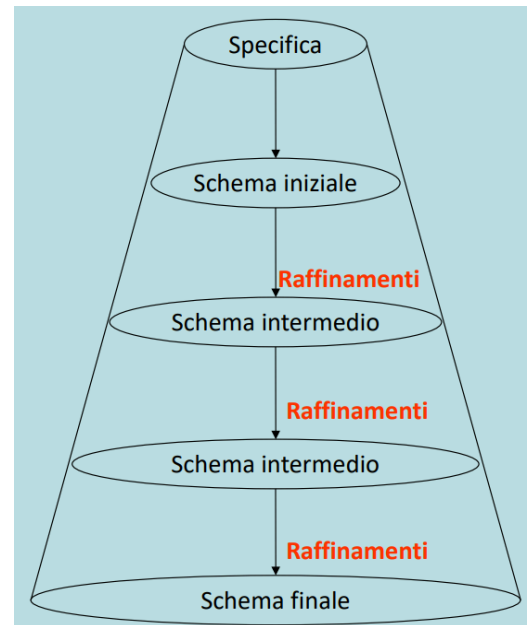
- **T4:** una relazione descrive un concetto con **esistenza autonoma**. In questo caso va sostituito con un'entità;



- **T5:** si applica per aggiungere **attributi ad entità**;



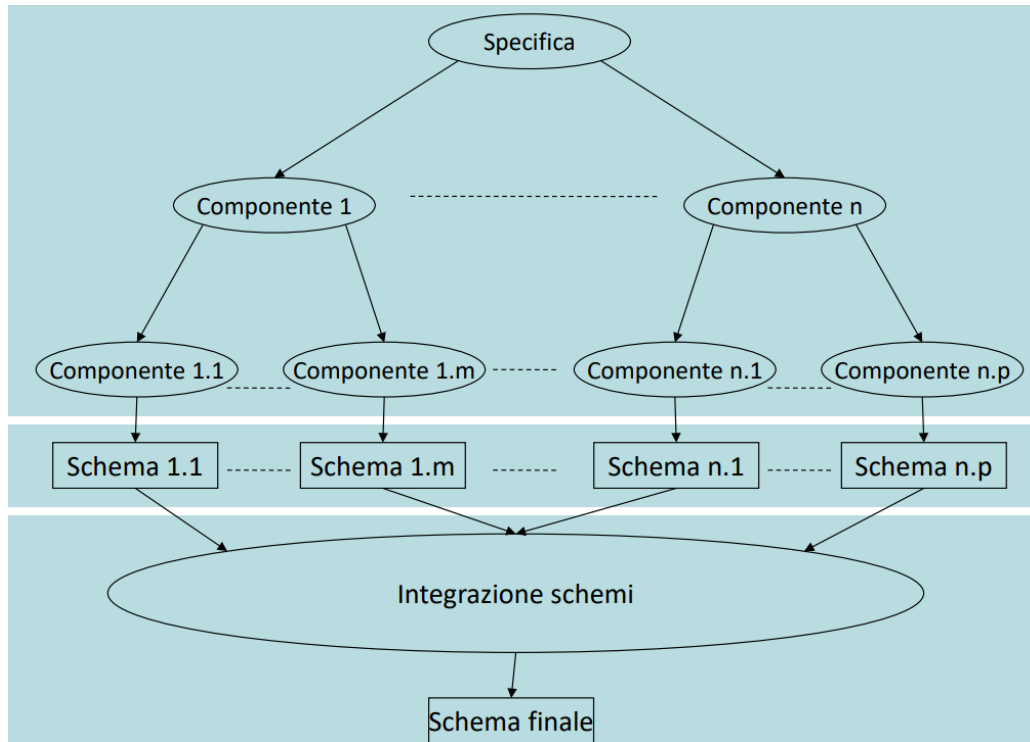
- **T6:** si applica per aggiungere **attributi a relazioni**;



Usando questo approccio abbiamo un **vantaggio** notevole per il progettista: descrive inizialmente lo schema trascurando i dettagli precisando lo schema gradualmente. Tuttavia si presenta anche un **problema**: non va bene per applicazioni complesse poiché è difficile avere una visione globale e precisa di tutte le componenti del sistema.

Strategia Bottom-Up

Le specifiche nascono suddivise per sotto-progetti descriventi frammenti limitati della realtà da schematizzare, sviluppati i sottoschemi separati si fondono per ottenere lo schema finale



Anche in questo caso abbiamo degli step precisi da seguire:

- **T1**: si individua nella specifica una classe di oggetti con proprietà comuni e si introduce un'**entità** corrispondente;

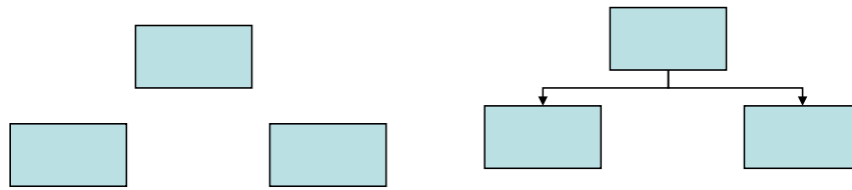
concetto



- **T2**: si individua nella specifica un legame logico fra entità e si introduce una **associazione** fra esse;



- **T3:** si individua una **generalizzazione** fra entità;



- **T4:** a partire da una serie di attributi si individua un'**entità** che li aggrega;



- **T5:** a partire da una serie di attributi si individua una **relazione** che li aggrega;



L'uso di questo approccio si adatta bene ad una **progettazione di gruppo** in cui diversi progettisti possono sviluppare parti disgiunte che possono essere assemblate successivamente, però l'**integrazione di sistemi concettualmente diversi** porta notevoli difficoltà.

Strategia Generale

Per applicare questa metodologia dobbiamo definire:

- Glossario dei termini;
- I requisiti senza ambiguità;

Dopo aver fatto ciò applichiamo i seguenti step:

- **Passo base:** individuare i concetti più rilevanti e rappresentarli in uno schema scheletro;
- **Passo di decomposizione:** effettuare una decomposizione dei requisiti facendo riferimento ai concetti definiti nello schema scheletro (questo step è opzionale);
- **Passo iterativo:** raffinare i concetti sulla base delle loro specifiche;
- **Passo di integrazione:** integrare i vari sottoschemi in un unico schema generale facendo riferimento allo schema scheletro;
- **Analisi di qualità:** fare un'analisi della qualità dello schema.

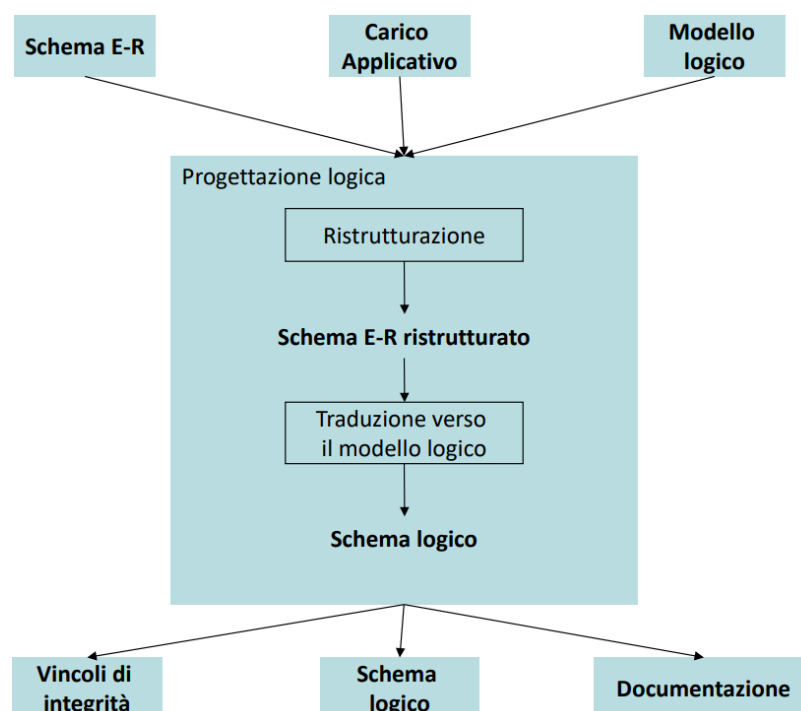
La qualità di uno schema relazionale viene giudicata in base a quelle proprietà che lo schema deve possedere:

- **Correttezza:** non ci devono essere errori:
 - *sintattici*: uso non ammesso dei costrutti
 - *semantici*: uso che non rispetta il loro significato
- **Completezza:** tutti i dati di interesse sono rappresentati e tutte le operazioni possono essere eseguite a partire dai concetti dello schema;
- **Leggibilità:** i requisiti sono rappresentati in maniera naturale e facilmente comprensibile, per fare ciò bisogna seguire alcune regole:
 - disporre al centro costrutti con più legami
 - usare linee perpendicolari cercando di minimizzare le intersezioni
 - disporre i padri di generalizzazioni sopra i figli
- **Minimalità:** tutte le specifiche vanno rappresentate una sola volta, non devono contenere *ridondanze*.

Fasi della progettazione logica

La progettazione logica di una base di dati costituisce il passaggio fondamentale per trasformare il modello concettuale in una struttura dati implementabile, articolandosi in due fasi sequenziali mirate all'efficienza e alla corretta traduzione delle informazioni:

- **Ristrutturazione dello schema E-R:** è una fase indipendente dal modello logico e si basa su criteri di *ottimizzazione* dello schema e di successiva *semplificazione*;
- **Traduzione verso il Modello Logico:** fa riferimento ad un modello logico (ad esempio relazionale) e può includere un'ulteriore *ottimizzazione* che si basa sul modello logico stesso (ad esempio normalizzazione).



Analisi delle prestazioni su schemi E-R

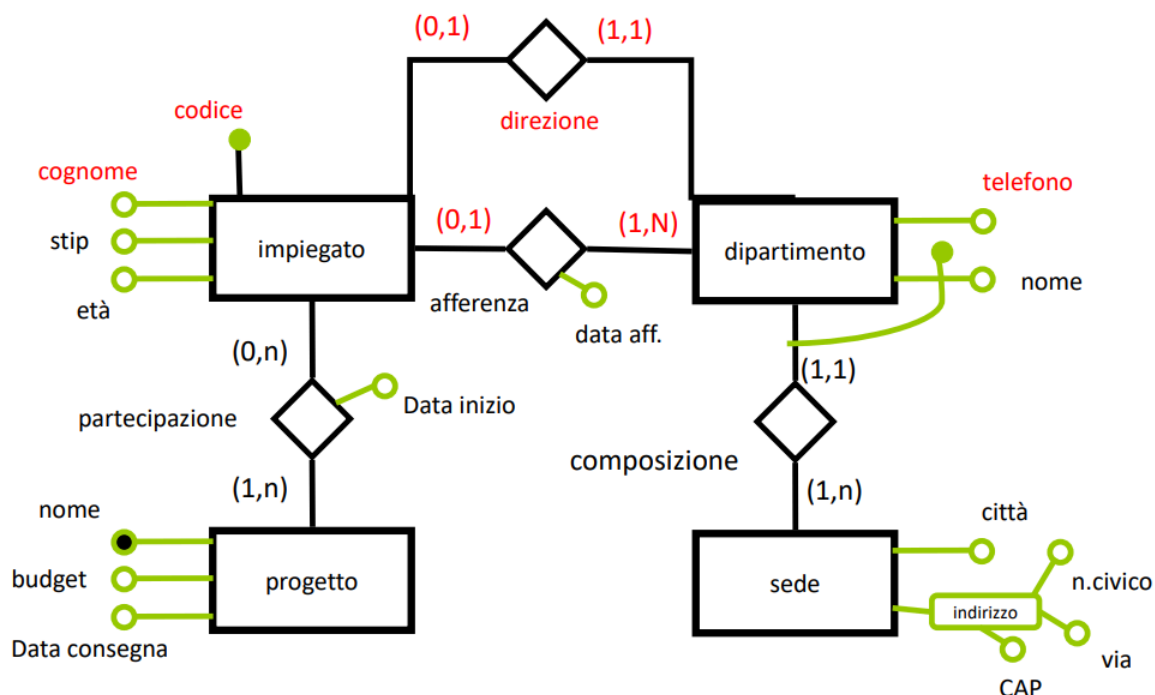
Abbiamo due indici per la valutazione di schemi E-R:

- **Costo di un'operazione:** in termini di numero di occorrenze di entità ed associazioni che mediamente vanno visitate per rispondere a quella operazione sulla base di dati;
- **Occupazione di memoria:** viene valutata in termini dello spazio di memoria, misurato in byte, necessario per memorizzare i dati del sistema.

Per studiare questi due parametri abbiamo bisogno di conoscere

- **Volume di dati:** numero medio di occorrenze di ogni entità ed associazione e la dimensione di ciascun attributo;
- **Caratteristiche delle operazioni:** tipo di operazione, frequenza e i dati coinvolti.

Prendiamo come esempio di analisi una ditta con sedi in città diverse:



Le operazioni dell'esempio sono:

- **Operazione 1:** assegna un impiegato ad un progetto;
- **Operazione 2:** trova i dati di un impiegato del dipartimento nel quale lavora e dei progetti in cui è coinvolto;
- **Operazione 3:** trova i dati di tutti gli impiegati di un certo dipartimento;
- **Operazione 4:** per ogni sede, trova i dipartimenti con il cognome del direttore e l'elenco degli impiegati.

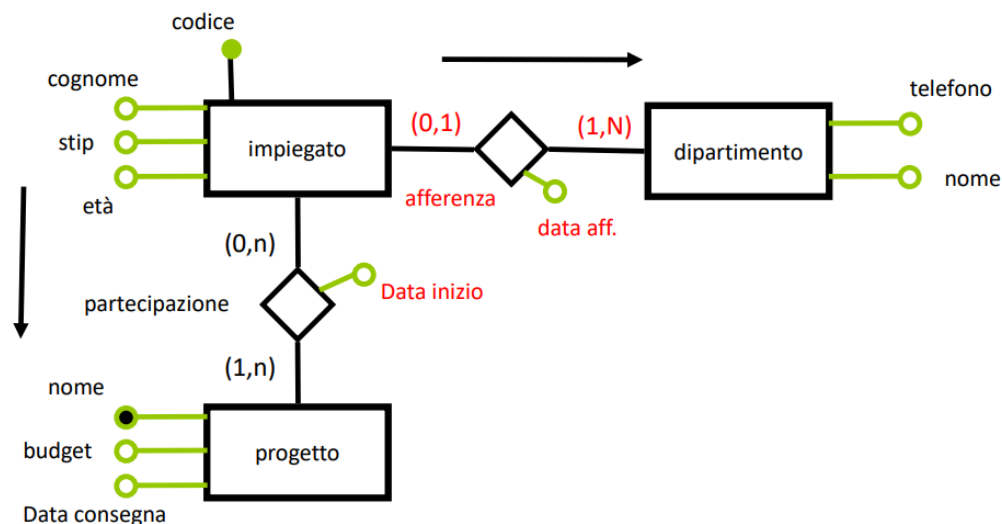
TABELLA DEI VOLUMI

Concetto	Tipo	Volume
Sede	E	10
Dipartimento	E	80
Impiegato	E	2000
Progetto	E	500
Composizione	R	80
Afferenza	R	1900
Direzione	R	80
Partecipazione	R	6000

TABELLA DELLE OPERAZIONI

Concetto	Tipo	Frequenza
Op. 1	I	50 al giorno
Op. 2	I	100 al giorno
Op. 3	I	10 al giorno
Op. 4	B	2 a settimana

Avendo a disposizione queste informazioni è possibile stimare il costo delle varie operazioni, prendiamo in esame l'**operazione 2**, quest'ultima prende in esame la seguente sezione di schema:



Per soddisfare la query dobbiamo accedere:

- Un occorrenza di Impiegato e di Afferenza e quindi di Dipartimento;
- Poi per avere i dati dei progetti a cui lavora, dobbiamo accedere (in media) a tre occorrenze di Partecipazione e quindi a tre entità Progetto.

Tutto viene riassunto nella seguente tabella degli accessi:

CONCETTO	COSTRUTTO	ACCESSI	TIPO
Impiegato	Entita'	1	L
Afferenza	Relazione	1	L
Dipartimento	Entita'	1	L
Partecipazione	Relazione	3	L
Progetto	Entita'	3	L

L = lettura, S = scrittura essendo la scrittura più onerosa della lettura avremo un rapporto di $1S = 2L$.

Come effettuare una ristrutturazione dello schema E-R:

- **Analisi delle Ridondanze:** si decide se eliminare o meno eventuali ridondanze;
- **Eliminazione delle Generalizzazioni:** tutte le generalizzazioni vengono analizzate e sostituite da altro;
- **Partizionamento/Accorpamento di entità e associazioni:** si decide se partizionare concetti in più parti o viceversa accorpare;
- **Scelta degli identificatori primari:** si sceglie un identificatore per quelle entità che ne hanno più di uno.

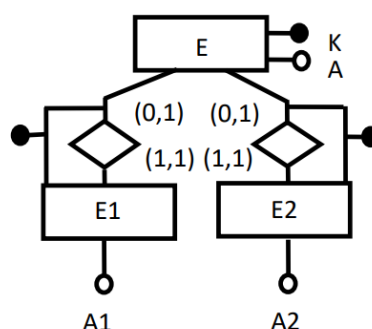
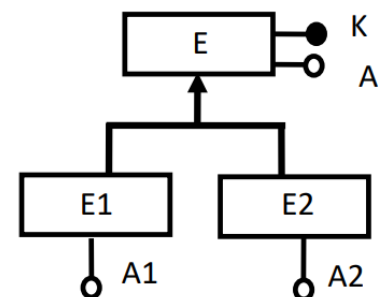
Costi di un dato derivato

Quando si hanno dei dati derivati da altri dati ($importoAnnuale = importoMensile \times nmeroMensilità$) si deve fare particolare attenzione, perché questi **riducono gli accessi** poiché non dobbiamo calcolare il dato derivato in fase di lettura ma ovviamente occuperanno **più spazio**. Per capire se vanno mantenuti o no è necessario fare un'analisi dei costi. (se vuoi un esempio guarda slide 75 - lezione14 "Progettazione 2").

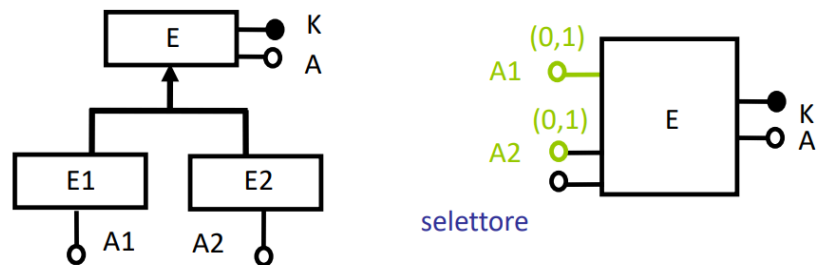
Eliminazione delle Gerarchie

Il modello relazionale non rappresenta le gerarchie, queste sono sostituite da entità e associazioni, e viene fatto attraverso:

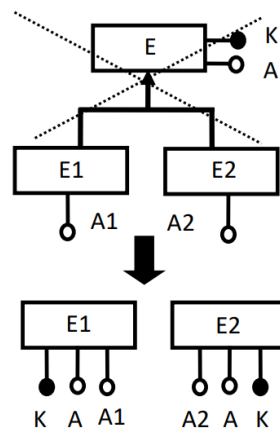
1. **Mantenimento delle entità con associazioni:** tutte le entità vengono mantenute, le entità *figlie* sono in associazione con l'entità padre e le entità figlie sono *identificate esternamente* tramite l'associazione.



2. **Collasso verso l'alto:** riunisce tutte le entità figlie direttamente nell'entità padre inserendo i campi dei figli come null, *selettore* è un attributo che specifica se una istanza di E appartiene a una delle sottoentità.



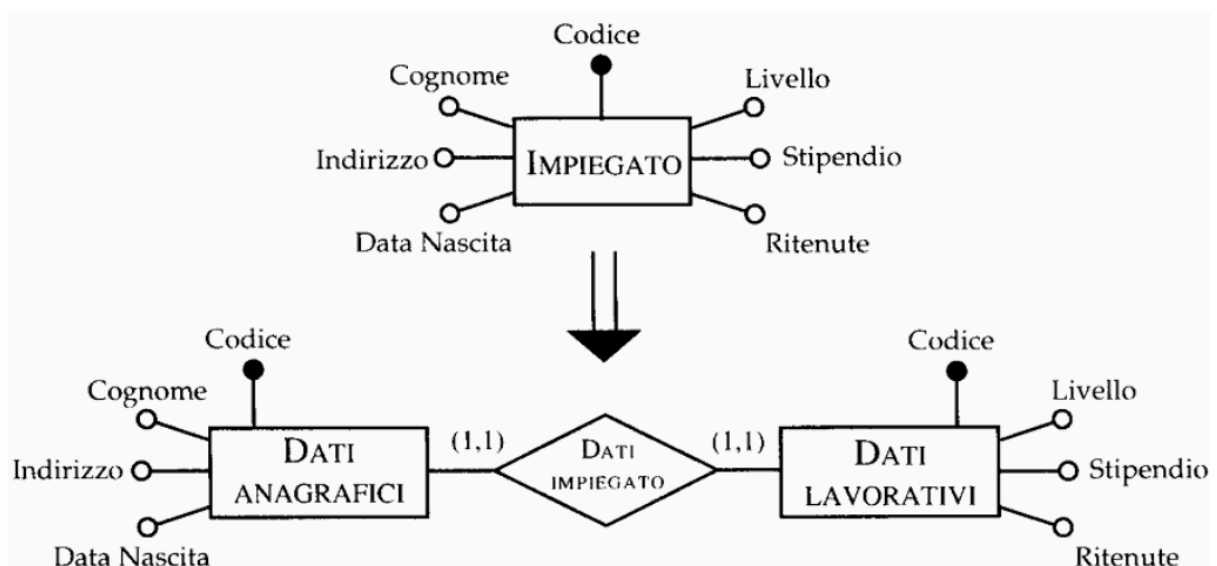
3. **Collasso verso il basso:** si elimina l'entità padre trasferendone gli attributi su tutte le entità figlie.



Partizionamento/Accoppiamento

In questa fase cerchiamo di ridurre gli accessi ai dati:

- **Separando** attributi di uno stesso concetto che vengono acceduti da operazioni diverse.
- **Raggruppando** attributi di concetti diversi che vengono acceduti dalle medesime operazioni.



Scelta della chiave primaria

Per poter scegliere la chiave primaria nel migliore dei modi seguiamo le seguenti regole:

1. Scegliere la chiave che è usata *più frequentemente* per accedere all'entità.
2. Scegliere delle chiavi *semplici* ed evitare chiavi composte.

Traduzione

Per poter gestire meglio le entità e le relazioni nasce la **traduzione standard**, dove:

- Ogni entità è tradotta con una relazione con gli stessi attributi;
- Ogni associazione è tradotta con una relazione con gli stessi attributi più gli attributi di ogni entità ad essa associata.

Problema generale

La progettazione concettuale e logica produce uno schema relazionale che rappresenta la realtà dei dati nella nostra applicazione, ma alcune volte ci sono delle cose che possono essere ottimizzate, vediamo alcune tecniche che ci permettono di farlo.

Ridondanze

Data la seguente identità: *StudenteEsami*(*Matricola*, *Nome*, *Telefono*, *Corso*, *Voto*), notiamo subito dei problemi:

- **Anomalia di aggiornamento:** se il telefono dello studente cambia allora deve essere cambiato ogni suo esame;
- **Anomalia della cancellazione:** se vengono annullati gli esami dati non rimane traccia dello studente;
- **Anomalia di Inserimento:** se uno studente non ha ancora dato esami non può essere inserito.

La soluzione a tutte queste anomalie è decomporre in due relazioni: la *Matricola*, il *Nome* ed il *Telefono* da una parte e il *Corso* e il *Voto* dall'altra.

Dipendenze Funzionali

Una dipendenza funzionale è un particolare vincolo di integrità che esprime legami funzionali tra gli attributi e una relazione. Ad esempio se prendo in esame l'entità *StudenteEsami* descritta prima avremo:

- *Matricola* → *Nome*, *Telefono*;
- *Matricola*, *Corso* → *Voto*.

Dati:

- Uno schema di relazione $R(A_1, A_2, \dots, A_n)$
- X e Y sottoinsiemi di $A_1, A_2, \dots, A_n$

Diciamo che X **implica funzionalmente** Y , in simboli $X \rightarrow Y$ per ogni relazione r dello schema R . Se due tuple t_1 e t_2 di r coincidono su tutti gli attributi di X allora devono anche coincidere su tutti gli attributi di Y .

Vediamo la notazione:

- $A, B, \dots$ attributi;
- U, B, W, X, Y, Z insiemi di attributi;
- R schema di relazione, r relazione;
- ABC sta per $\{A, B, C\}$, XY sta per $X \cup Y$, XA e AX stanno per $X \cup \{A\}$.

Soddisfazioni di dipendenze funzionali

Diciamo che una relazione r soddisfa la dipendenza funzionale $X \rightarrow Y$ se per ogni coppia di tuple t_1 e t_2 in r si ha che:

- $t_1[X] = t_2[X]$ implica $t_1[Y] = t_2[Y]$

Logica delle dipendenze funzionali

Sia F un insieme di dipendenze funzionali per uno schema R e sia $X \rightarrow Y$ una dipendenza funzionale, diciamo che F **logicamente implica** $X \rightarrow Y$ e si scrive $F \models X \rightarrow Y$, se per ogni relazione r di R che se soddisfa tutte le dipendenze di F , r soddisfa anche $X \rightarrow Y$

Chiusura di un insieme di dipendenze funzionali

Dato un insieme F di dipendenze funzionali, la sua chiusura F^+ è l'insieme delle dipendenze funzionali che sono implicate logicamente da F , in simboli: $F^+ = \{X \rightarrow Y | F \models X \rightarrow Y\}$.

Chiavi per uno schema con insieme di dipendenze funzionali

Sia $R(A_1, A_2, \dots, A_n)$ uno schema, F un insieme di dipendenze funzionali su R ed X un sottoinsieme di $\{A_1, A_2, \dots, A_n\}$. Si dice che X è una *chiave* di (R, F) se:

- $X \rightarrow A_1, A_2, \dots, A_n \in F^+$;
- Per ogni sottoinsieme proprio di Y di X la dipendenza $Y \rightarrow A_1, A_2, \dots, A_n \notin F^+$.

Commenti

Di seguito una serie di osservazioni sulle dipendenze funzionali:

- Una dipendenza funzionale è **dettata dalla semantica degli attributi** di una relazione e non può essere inferita da una particolare istanza dello schema.
- Un'istanza di uno schema che rispetti una data dipendenza funzionale viene detta **istanza legale**.
- Se X è una chiave in uno schema R allora ogni altro attributo di R dipende funzionalmente da X .

- Dire che $X \rightarrow Y$ significa asserire che i valori della componente Y dipendono da i valori della componente X .
- Se $X \rightarrow Y$ non necessariamente $Y \rightarrow X$.
- Il concetto di superchiave si esprime facendo uso delle dipendenze funzionali: $K \subseteq T$ è superchiave di $R(T)$ se e solo se $K \rightarrow T$.

Necessità di un Calcolo Logico

Il problema è quello di calcolare la chiusura di un insieme F di dipendenze funzionali. Per fare ciò definiamo un calcolo logico tale che $F \models X \rightarrow Y$ se e soltanto se $X \rightarrow Y$ si può sintatticamente dedurre da F nel calcolo logico.

Assiomi di Armstrong

Gli Assiomi di Armstrong sono un insieme di regole di inferenza, ovvero regole logiche e matematiche che ti permettono di scoprire **nuove dipendenze funzionali** a partire da un insieme di dipendenze iniziali già note. Vediamo quali sono, sia $U = \{A_1, A_2, \dots, A_n\}$ un universo di attributi:

- **Riflessività:** se $Y \subset X \subset U$ allora $F \vdash X \rightarrow Y$.
- **Aumento:** se $F \vdash X \rightarrow Y$ allora $F \vdash XZ \rightarrow YZ$.
- **Transitività:** se $F \vdash X \rightarrow Y$ e $Y \rightarrow Z$ allora $F \vdash X \rightarrow Z$

Notazione: $X \rightarrow Y$ si deduce da F applicando gli assiomi di Armstrong indica con $F \vdash X \rightarrow Y$.

“La dipendenza funzionale di Y da X si deduce dall'insieme F ” traduzione di $F \vdash X \rightarrow Y$.

Deducibilità delle dipendenze funzionali

Diciamo che $F \vdash X \rightarrow Y$ se $X \rightarrow Y$, si può dedurre da F applicando un numero finito di volte gli assiomi di Armstrong.

Ad esempio $\{A \rightarrow B, B \rightarrow C, C \rightarrow D\} \vdash A \rightarrow D$ applicando 2 volte la *transitività* il risultato dovrà rispettare queste due proprietà:

- **Correttezza:** se $F \vdash X \rightarrow Y$ allora $F \models X \rightarrow Y$.
Se usi gli Assiomi di Armstrong per dimostrare una nuova dipendenza funzionale, hai la garanzia matematica che quella dipendenza sarà vera e valida anche sui dati reali.
- **Completezza:** $F \models X \rightarrow Y$ allora $F \vdash X \rightarrow Y$.
Se nel mondo reale dei dati esiste una dipendenza funzionale logica, allora sei matematicamente in grado di dimostrarla a tavolino usando gli Assiomi di Armstrong.

Se gli assiomi di Armstrong sono dimostrabili allora siamo in grado di capire se una dipendenza funzionale è valida. Seguono le dimostrazioni.

Dimostrazione della Correttezza

- Basta dimostrare che gli assiomi di Armstrong sono corretti
- **Riflessività**
Se $Y \subset X \subset U$ allora $F \vdash X \rightarrow Y$
- **Dimostrazione:**
se $t_1[X]=t_2[X]$ allora ovviamente anche $t_1[Y]=t_2[Y]$ perché ogni attributo di Y sta anche in X .

Correttezza della TRANSITIVITA'

- **Transitività**
Se $F \vdash X \rightarrow Y$ e $F \vdash Y \rightarrow Z$ allora $F \vdash X \rightarrow Z$
- **Dimostrazione:**
Supponiamo che $r \models F$ allora, per ipotesi, $r \models X \rightarrow Y$ e $r \models Y \rightarrow Z$. Supponiamo che $t_1[X]=t_2[X]$ allora per la prima $t_1[Y]=t_2[Y]$ da cui per la seconda $t_1[Z]=t_2[Z]$.

Lemmi per derivare le dipendenze funzionali

Prima di verificare la completezza degli assiomi di Armstrong introduciamo altre regole per derivare dipendenze funzionali.

- **Regola di Decomposizione:** se $F \vdash X \rightarrow Y$ e $Z \subset Y$ allora $F \vdash X \rightarrow Z$.
Dimostrazione: se $F \vdash X \rightarrow Y$ e $Z \subset Y$ allora per riflessività $F \vdash Y \rightarrow Z$ da cui per transitività si ottiene $F \vdash X \rightarrow Z$.
- **Regola dell'Unione:** se $F \vdash X \rightarrow Y$ e $F \vdash X \rightarrow Z$ allora $F \vdash X \rightarrow YZ$.
Dimostrazione: se $F \vdash X \rightarrow Y$ e $F \vdash X \rightarrow Z$ aumentando la prima di X e la seconda di Y si ha $F \vdash X \rightarrow XY$ e $F \vdash XY \rightarrow ZY$ che per transitività implicano $F \vdash X \rightarrow YZ$.
- **Regola di Pseudotransitività:** se $F \vdash X \rightarrow Y$ e $F \vdash WY \rightarrow Z$ allora $F \vdash WX \rightarrow Z$.
Dimostrazione: se $F \vdash X \rightarrow Y$ e $F \vdash WY \rightarrow Z$ aumentando la prima di W si ha $F \vdash WX \rightarrow WY$ e $F \vdash WY \rightarrow Z$ che per transitività implicano $F \vdash WX \rightarrow Z$.

Correttezza dell'Aumento

• Aumento

Se $F \vdash X \rightarrow Y$ allora $F \vdash XZ \rightarrow YZ$

• Dimostrazione:

Supponiamo che $r \models F$ allora, per ipotesi, $r \models X \rightarrow Y$. Supponiamo che $t_1[XZ]=t_2[XZ]$ allora $t_1[X]=t_2[X]$ da cui segue, per ipotesi, $t_1[Y]=t_2[Y]$ e quindi $t_1[YZ]=t_2[YZ]$.

Correttezza

- Supponiamo che $F \vdash X \rightarrow Y$ e sia $D_1 D_2 \dots D_n = X \rightarrow Y$ la relativa catena. Procedendo per induzione, se D_i è in F allora ovviamente $F \models D_i$. Se D_i si ottiene da precedenti D_j mediante gli assiomi di Armstrong allora per l'ipotesi induttiva $F \models D_j$ e per la correttezza dei singoli assiomi si ha $F \models D_i$. Segue $F \models D_n$ cioè $F \models X \rightarrow Y$

- **Lemma fondamentale:** definiamo $X_F^+ = \{A | F \vdash X \rightarrow A\}$ (omettiamo l'indice F quando non c'è ambiguità). Il lemma afferma $F \vdash X \rightarrow Y$ se e solo se $Y \subseteq X^+$.

Dimostrazione: ($\Leftarrow$) sia $Y = A_1, A_2, \dots, A_n$ e supponiamo che $Y \subseteq X^+$. Allora per definizione $F \vdash X \rightarrow A_j$ per ogni $i = 1, 2, \dots, n$. Da questa per la regola dell'unione si ha $F \vdash X \rightarrow Y$.
 ($\Rightarrow$) viceversa se $F \vdash X \rightarrow Y$ allora per la regola di decomposizione si ha $F \vdash X \rightarrow A_j$ per ogni $i = 1, 2, \dots, n$ e quindi $Y \subseteq X^+$.

Possiamo adesso dimostrare la completezza degli assiomi di Armstrong:

- Dobbiamo dimostrare che $F \models X \rightarrow Y$ allora $F \vdash X \rightarrow Y$.
- Basta far vedere che se **not** $F \vdash X \rightarrow Y$ allora **not** $F \models X \rightarrow Y$.
- Infatti, supponiamo che **not** $F \vdash X \rightarrow Y$ allora per il Lemma $Y \not\subseteq X^+$ e $Y \neq X^+$. Allora è possibile considerare la relazione r dello stesso schema fatta dalle due tuple t_1 e t_2

$$\begin{array}{cc} t_1 = & 1 \ 1 \ \dots \ 1 \quad 1 \ 1 \ \dots \ 1 \\ & X^+ \quad \text{Not } X^+ \end{array} \qquad \begin{array}{cc} t_2 = & 1 \ 1 \ \dots \ 1 \quad 0 \ 0 \ \dots \ 0 \\ & X^+ \quad \text{Not } X^+ \end{array}$$

- Facciamo vedere che **r soddisfa tutte le dipendenze di F .**
 - Infatti, supponiamo che esista una $V \rightarrow W \in F$ tale che r non la soddisfa.
 - Questo vuol dire che t_1 e t_2 coincidono su V ma non su W . Questo vuol dire che $V \subseteq X^+$ e $W \not\subseteq X^+$.
 - Per il lemma segue che $F \vdash X \rightarrow V$ che assieme a $F \vdash V \rightarrow W$ per transitività da $F \vdash X \rightarrow W$ che contraddice $W \not\subseteq X^+$.
 - Quindi r soddisfa tutte le dipendenze di F .
- Tuttavia **r non soddisfa $X \rightarrow Y$** . Infatti $t_1[X] = t_2[X]$ ma $t_1[Y] \neq t_2[Y]$ poiché $Y \not\subseteq X^+$ in quanto, per ipotesi, **not** $F \vdash X \rightarrow Y$.
- Quindi **not** $F \models X \rightarrow Y$, che conclude la dimostrazione di completezza.

Chiusure, equivalenze e ricoprimenti minimi

Calcolo delle chiusure

Ricordiamo che $F^+ = \{X \rightarrow Y \mid F \models X \rightarrow Y\}$, questo calcolo può essere molto costoso in quanti se $F = \{A \rightarrow B_1, A \rightarrow B_2, \dots, A \rightarrow B_n\}$ allora F^+ include $A \rightarrow Y$ per ogni Y sottoinsieme di $\{B_1, B_2, \dots, B_n\}$. Quindi ha cardinalità almeno 2^n .

Invece calcolare $X_F^+ = \{A \mid F \vdash X \rightarrow A\}$ è molto più semplice e si fa attraverso il seguente algoritmo:

1. $X^{(0)} := X, j = 0$
2. REPEAT
3. $X^{(j+1)} := X^{(j)} \cup \{A \mid \exists Y \rightarrow Z \in F \mid A \in Z \wedge Y \subseteq X^{(j)}\}$
4. UNTIL $(X^{(j+1)} = X^{(j)})$
5. SET $X^+ := X^{(j)}$

Equivalenze di dipendenze funzionali

Siano F, G insiemi di dipendenze funzionali, allora diciamo che sono *equivalenti* se $F^+ = G^+$. La relazione di equivalenza tra insiemi di dipendenze ci permette di capire quando due schemi di relazione rappresentano gli stessi fatti: basta controllare che gli attributi siano uguali e che abbiano le stesse dipendenze. Per fare ciò usiamo il seguente algoritmo:

- Per ogni $Y \rightarrow Z$ in F controlliamo se essa è in G^+ calcolando Y_G^+ e controllando se $Z \subseteq Y_G^+$. Questo implica $F^+ \subseteq G^+$.
- Viceversa in maniera analoga si può controllare se $G^+ \subseteq F^+$.

Insiemi di Dipendenze Minimali

Un insieme di dipendenze funzionali F è **minimale** se:

1. Ogni lato destro di una dipendenza è un singolo attributo;
2. Per ogni dipendenza $X \rightarrow A$ in F , $F \setminus \{X \rightarrow A\}$ non è equivalente a F ;
3. Per ogni $X \rightarrow A$ in F e $Z \subset X$, $F \setminus \{X \rightarrow A\} \cup \{Z \rightarrow A\}$ non è equivalente a F .

La *regola 2* garantisce che nessuna dipendenza in F è ridondante. La *regola 3* garantisce che nessun attributo in qualunque primo membro sia ridondante.

Dato F si dice che G è un suo **ricoprimento minimale** se G è *minimale* ed è *equivalente* a F .

Teorema: Ogni insieme di dipendenze funzionali ha un ricoprimento minimale.

Dimostrazione del teorema del ricoprimento minimale

- Dato l'insieme F costruiamo un insieme F' ad esso equivalente con la proprietà 1.
- Per garantire la proprietà 2. Basta cancellare ogni dipendenza $X \rightarrow A$ che non soddisfa 2.
- Analogamente se esiste una regola che non soddisfa la 3 si accorcia e si continua il processo che avrà termine ottenendo il ricoprimento minimale.

Si può dimostrare che basta applicare la regola 3. E solo alla fine la 2. Ma non viceversa!

Decomposizioni di uno schema, Decomposizioni che preservano i dati (loss-less join)

Dato uno schema $R = \{A_1, A_2, \dots, A_n\}$ una sua **decomposizione** è un insieme $d = \{R_1, R_2, \dots, R_k\}$ di sottoinsiemi di R tali che: $R = R_1 \cup R_2 \cup \dots \cup R_k$.

Ad esempio consideriamo lo schema

StudentiEsamiCorsi = {Matricola, Nome, Indirizzo, Telefono, Corso, Professore, Voto}

L'insieme $d = \{\text{Studenti}, \text{Esami}, \text{Corsi}\}$ con

Studenti = {Matricola, Nome, Indirizzo, Telefono}

Esami = {Matricola, Corso, Voto}

Corsi = {Corso, Professore}

è una sua decomposizione.

Preservazione dei dati

Decomporre lo schema iniziale porta il vantaggio di evitare le ridondanze nella rappresentazione. Nel nostro caso basta fare una *natural join* per ottenere la composizione iniziale. Questa proprietà si chiama **loss-less join**.

Dato uno schema R con un insieme F di dipendenze funzionali, una sua decomposizione $d = \{R_1, R_2, \dots, R_k\}$ si dice che **preserva i dati** se per ogni relazione r di R che soddisfa tutte le dipendenze di F si ha: $r = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \dots \bowtie \pi_{R_k}(r)$.

Siano R, F come detto prima e sia $m_d(r) = \pi_{R_1}(r) \bowtie \pi_{R_2}(r) \bowtie \dots \bowtie \pi_{R_k}(r)$ allora $r \subseteq m_d(r)$ è sempre vero ma $m_d(r) \subseteq r$ non è sempre vero.

Per controllare se una decomposizione preserva i dati seguiamo questo algoritmo:

Input:

$R = \{A_1, A_2, \dots, A_n\}$, F , $d = \{R_1, R_2, \dots, R_k\}$

Output:

Yes/No se d preserva i dati.

INIZIALIZZAZIONE:

Consideriamo una matrice

$M = \{R_1, R_2, \dots, R_k\} \times \{A_1, A_2, \dots, A_n\}$

dove nell'elemento $R_i A_j$ mettiamo a_j se A_j è in R_i
altrimenti mettiamo b_{ij} .

• ITERAZIONE:

Applichiamo finché è possibile ogni dipendenza $X \rightarrow Y$ in F nel seguente modo: se esistono due righe di M che coincidono su X allora facciamole coincidere anche in Y :

- se ho uno dei due a_j allora cambiamo l'altro (b_{ij}) in a_j ;
- Altrimenti prendiamo uno dei due e lo facciamo uguale all'altro.

- Se durante il passo precedente si produce la riga $a_1 a_2 \dots a_n$ allora rispondi *YES*
(La decomposizione preserva i dati)
- Altrimenti rispondi *NO*
(La decomposizione non preserva i dati)

Decomposizioni che conservano le dipendenze funzionali

La proiezione $\pi_Z(F)$ di F su un insieme Z di attributi è l'insieme delle dipendenze $X \rightarrow Y$ appartenenti a F^+ tali che $XY \subseteq Z$.

L'algoritmo per il calcolo della proiezione di un insieme di dipendenze è il seguente:

Input: $(R(A_1, A_2, \dots, A_n), F)$

Output: Una copertura della proiezione di F su $T \subseteq (A_1, A_2, \dots, A_n)$

Begin

$\pi_T(F) = \emptyset$

for each $Y \subset T$ **do**

$Z = Y_F^+ - Y$

$\pi_T(F) = \pi_T(F) \cup \{Y \rightarrow (Z \cap T)\}$

end for

return $\pi_T(F)$

end

Conservazione delle Dipendenze

Dato uno schema relazionale (R, F) ed una sua decomposizione $d = \{R_1, R_2, \dots, R_k\}$, si dice che essa **conserva le dipendenze funzionali** se F è implicata logicamente dall'unione delle proiezioni $\pi_{R_i}(F)$ di F sugli R_i .

L'algoritmo per controllare se una decomposizione preserva le dipendenze funzionali è il seguente:

- **Input:** $R = \{A_1, A_2, \dots, A_n\}$, F , $d = \{R_1, R_2, \dots, R_k\}$
- **Output:** Yes/No se d preserva F .
- **IDEA DELL'ALGORITMO:**
 - Sia $G = \bigcup_{i=1, \dots, k} \pi_{R_i}(F)$.
 - Dobbiamo provare che G è equivalente ad F (senza calcolare G).
 - Basta provare che per ogni $X \rightarrow Y$ in F , Y è contenuto in X^+ calcolato rispetto a G . Per questo calcolo basta fare la chiusura di X rispetto alle varie proiezioni di F sulle R_i .

Forme Normali

Una relazione r è detta in **forma normale di Boyce-Codd (BCNF)** se per ogni dipendenza funzionale $X \rightarrow A$ definita su di essa, X contiene una chiave K di r , cioè X è superchiave per r .

Ad esempio consideriamo lo schema $R(C, S, Z)$ con dipendenze $CS \rightarrow Z$ e $Z \rightarrow C$, le chiavi sono CS e ZS . Tuttavia $Z \rightarrow C$ non soddisfa la proprietà perché Z non è una superchiave quindi non è in **BCNF**.

Una relazione r è detta in **terza forma normale (3NF)** se per ogni dipendenza funzionale $X \rightarrow A$ definita su di essa si ha almeno una delle due condizioni:

- X contiene una chiave K di r .
- A appartiene ad almeno una chiave di r .

As esempio, consideriamo lo schema di prima $R(C, S, Z)$, abbiamo già osservato che non è in BCNF ma essa è in **3NF** infatti, nella dipendenza $Z \rightarrow C$, Z non è una superchiave, tuttavia C è primo perché è elemento della chiave CS .

Perchè scegliamo queste forme normali?

Lo scopo delle BCNF è di **eliminare ridondanze causate dalle dipendenze**:

Supponiamo che R sia in BCNF e supponiamo per assurdo che possiamo predire il valore di un attributo da qualche dipendenza (quindi che ci sia ridondanza). Quindi avremmo una situazione del tipo:

X	Y	A	Con y diversa da z Dove possiamo dedurre che $? = a$. Questo implica che per qualche $Z \subseteq XX$ vale la dipendenza $Z \rightarrow A$. Tuttavia poiché R è in BCNF allora Z è una superchiave così come X . Ma questo implicherebbe $y = z$ che contraddice l'assunzione che le due tuple sono distinte. Assurdo!

x	y	a	
x	z	?	

Consideriamo lo schema CSZ con le dipendenze $CS \rightarrow Z$ e $Z \rightarrow C$ che è in 3NF ma non in BCNF:

Con s **diversa** da t si può dedurre che $? = c$ perché $Z \rightarrow C$. Quindi ci possono essere ridondanze dovute a dipendenze.

C	S	Z

c	s	z
?	t	z

Algoritmo per BCNF

Prima di vedere l'algoritmo per la BCNF bisogna attenzionare alcuni lemmi preliminari:

Lemma 1

Sia R uno schema con dipendenze F e sia $d = \{R_1, R_2, \dots, R_k\}$ una decomposizione che preserva i dati rispetto a F , e sia $d' = \{S_1, S_2\}$ una decomposizione di R_1 che preserva i dati rispetto a $\pi_{R_1}(F)$. Allora la decomposizione di R , $d'' = \{S_1, S_2, R_2, \dots, R_k\}$ preserva i dati rispetto a F .

Lemma 2

- a) Ogni schema R con due attributi è in BCNF
- b) Se R non è in BCNF allora esistono due attributi A, B tali che: $(R - AB) \rightarrow A$.

Lemma 3

Dati (R, F) , se proiettiamo su $R_1 \subseteq R$ ottenendo F_1 , e successivamente proiettiamo su $R_2 \subseteq R_1$ ottenendo F_2 , allora si ha che $F_2 = \pi_{R_2}(F)$.

L'**input** dell'algoritmo è lo schema R e dipendenze F , l'**output** invece sarà una decomposizione che **preserva i dati** tale che ogni componente sia in BCNF rispetto alla proiezione di F su quella componente.

L'idea dell'algoritmo è quella di decomporre R in due schemi:

- XA in BCNF, dove vale $X \rightarrow A$
- $R - A$

Tali che $(R - A, XA)$ preserva i dati. Si parte da $R - A$, si calcola la proiezione di F (questo step ha un costo esponenziale) su tale schema e si continua come sopra, fino a quando ci si riduce a due soli attributi quando per il *Lemma 2*, è senz'altro in BCNF.

La decomposizione trovata non è l'unica, dipende dall'ordine con cui vengono analizzate le dipendenze.

begin

$\rho = \{(R_1(T_1), F_1) = (R(T), F)\}, n = 1$

while $\exists (R_i(T_i), F_i) \in \rho$ non in BCNF per $X \rightarrow A$ **do**

$n := n + 1$

$T' = X^+$

$F' = \pi_{T'}(F_i)$

$T'' = T_i - (T' - X)$

$F'' = \pi_{T''}(F_i)$

$\rho = \rho - \{(R_i(T_i), F_i)\} \cup \{(R_i(T'), F'), (R_n(T''), F'')\}$

end while

end

Sfortunatamente non è possibile prevedere pregi e difetti delle scomposizioni che si ottengono.

Algoritmo per 3NF

L'**input** è uno schema R e dipendenze F , con F ricoprimento minimale, l'**output** sarà una decomposizione di R che **conserva le dipendenze** e tale che ogni suo elemento è 3NF.

Se ci sono attributi non presenti in F essi possono essere raggruppati in un solo schema ed eliminati. Se una dipendenza di F coinvolge tutti gli attributi di R , allora ritorna (R) , altrimenti ritorna la decomposizione fatta da tutti gli altri XA tali che $X \rightarrow A$ appartiene ad F .

Begin

$ND = T - \bigcup_{X \rightarrow A \in F} A$

$SD = \bigcup_{X \rightarrow A \in F} X \cap \bigcup_{X \rightarrow A \in F} A$

$Cand = [ND :: (SD)]$

$Keys = \phi$

while $Cand \neq \phi$ **do**

$X :: (Y) = First(Cand)$

$Cand = Rest(Cand)$

if $\nexists k \subset X \mid k \in Keys$ **then**

if $X^+ = T$ **then**

$Keys = Keys \cup \{X\}$

else

$A_1, \dots, A_n = Y - X^+$

for $i = 1 \dots n$ **do**

$cand = cand \cup XA_i :: (A_{i+1} \dots A_n)$

end for

end if

end if

end while

end

Gestione delle transazioni e Organizzazione Fisica

Una transazione è una parte di programma caratterizzata da un inizio (*start transaction*) e una fine, al cui interno deve essere eseguita una e una sola volta una delle seguenti operazioni terminali:

- **commit work**: per terminare correttamente e rendere permanenti le modifiche;
- **rollback work**: per abortire la transazione e annullare le modifiche.

Un **sistema transazionale (OLTP)** gestisce l'esecuzione di transazioni per conto di applicazioni concorrenti.

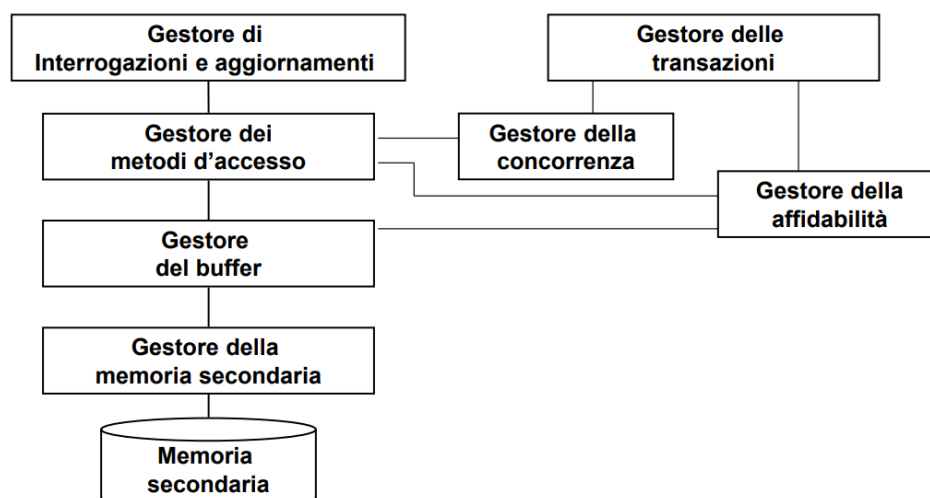
Le transazioni godono di 4 proprietà fondamentali, le proprietà ACID:

- **Atomicità**: la transazione è un'unità indivisibile, non può lasciare il database in uno stadio intermedio. In caso di guasto prima del commit si esegue l'**UNDO**; dopo il commit, se necessario, si esegue il **REDO**;
- **Consistenza**: la transazione deve rispettare i vincoli di integrità. Se lo stato iniziale è corretto, anche quello finale deve esserlo;
- **Isolamento**: la transazione non risente degli effetti delle altre transazioni concorrenti, l'esecuzione concorrente di una collezione di transazioni deve produrre un risultato che si potrebbe ottenere con una esecuzione sequenziale. Come conseguenza avremo che una transazione non espone i suoi stadi intermedi.
- **Durabilità (persistenza)**: gli effetti di una transazione andata in commit non vanno perduti, anche in presenza di guasti.

Architettura interna di un DBMS

Gestore degli accessi e delle interrogazioni

Gestore delle transazioni



Gestore dell'affidabilità

Questo modulo assicura atomicità e durabilità gestendo il **log**, un file sequenziale su memoria stabile che registra tutte le operazioni.

Prima di vedere come è formata la struttura dei **record dei file di log**, dobbiamo definire delle variabili:

- T : indica la transazione specifica che sta svolgendo l'operazione.
- O : indica l'oggetto del database su cui si interviene (ad esempio una riga specifica di una tabella, una pagina, ecc.).
- BS (*Before State*): indica il valore che l'oggetto aveva *prima* della modifica, serve per l'operazione di UNDO.
Se una transazione genera un errore a metà esecuzione, il sistema non può lasciare dati a metà. Leggerà quindi il BS dal file di log e sovrascriverà il dato parziale, riportando il database allo stato coerente iniziale.
- AS (*After State*): indica il valore dell'oggetto *dopo* la modifica, serve per l'operazione di REDO.
Se il sistema subisce un crash (es. salta la corrente) subito dopo che una transazione è terminata con successo, ma prima che le modifiche in memoria RAM siano state fisicamente scritte sull'hard disk, il sistema al riavvio leggerà l' AS dal log e ri-applicherà la modifica per non perdere il lavoro fatto.

Ora possiamo vedere le firme delle 6 operazioni fondamentali scritte nel log durante la vita di una transazione:

- **Begin** $B(T)$: viene scritto per segnare l'inizio della transazione T .
- **Insert** $I(T, O, AS)$: la transazione T inserisce un nuovo oggetto O . Dato che l'oggetto è appena nato, non esiste alcuno stato precedente. Pertanto nel log viene registrato solo il nuovo valore AS .
- **Delete** $D(T, O, BS)$: la transazione T elimina l'oggetto O . In questo caso l'oggetto cessa di esistere quindi non ci sarà alcun valore "nuovo" e viene salvato solo il valore che aveva prima di essere cancellato BS .
- **Update** $U(T, O, BS, AS)$: la transazione T aggiorna un oggetto O esistente. Questa è l'operazione più completa: il log registra sia il vecchio valore BS per un eventuale undo, sia il nuovo valore AS per un eventuale redo.
- **Commit** $C(T)$: segna la terminazione con successo della transazione T . Scrivere questo record nel log significa che il DBMS prende l'impegno definitivo di rendere permanenti le modifiche di T .
- **Abort** $A(T)$: segna il fallimento o l'annullamento della transazione T . Al rilevamento di questo record il sistema sa che deve avviare le procedure di undo leggendo i vari BS associati a T .

Regole fondamentali

1. **Write-Ahead-Log (WAL)**: prima di poter sovrascrivere fisicamente un dato in un database è obbligatorio registrare il suo *before state* nel file di log, così da avere sempre a disposizione le informazioni necessarie per eseguire l'undo.

2. **Commit-Precedenza:** prima di poter confermare con successo la fine di una transazione (commit) è obbligatorio salvare il suo *after state* nel file di log, così da poter eseguire il redo ed evitare perdite di informazioni.

Checkpoint e Ripristino

Il Checkpoint (CK) è un'operazione che serve a ridurre il tempo di ripristino, registrando quali transazioni sono attive in un dato istante e portando le pagine "sporche" sul disco.

L'algoritmo di **ripresa a caldo (Warm Restart)** è la procedura cruciale eseguita quando si verifica un **guasto di sistema** (o *soft crash*), come ad esempio un blackout elettrico o un blocco del sistema operativo.

L'obiettivo è riportare il database a uno stato coerente e si basa sulle regole citate in precedenza. Per non dover rileggere l'intero log da capo a fine esso sfrutta i Checkpoint. Questo algoritmo si divide in 4 fasi:

1. Trovare l'ultimo checkpoint ripercorrendo il log a ritroso.
2. Costruire gli insiemi:
 - a. **UNDO:** transazioni da disfare;
 - b. **REDO:** transazioni da rifare;
3. Ripercorrere il log all'indietro disfacendo le azioni delle transazioni in UNDO.
4. Ripercorrere il log in avanti rifacendo le azioni delle transazioni in REDO.

Mentre la ripresa a caldo gestisce i guasti di sistema (in cui la memoria secondaria rimane intatta), la **ripresa a freddo** (nota anche come *Cold Restart* o *Media Recovery*) interviene in uno scenario molto più critico: il **guasto fisico ai supporti di memorizzazione** (un *hard crash*).

In questa situazione, il file di log da solo non è sufficiente. Per poter eseguire una ripresa a freddo, il Gestore dell'affidabilità si basa sul fatto che l'amministratore del database abbia predisposto due elementi fondamentali:

1. **Un Dump (Backup):** Una copia di sicurezza completa dell'intero database, realizzata periodicamente e salvata su un supporto fisico separato.
2. **L'integrità del Log:** Il file di log deve essere sopravvissuto al guasto. Per questo motivo, nei sistemi di produzione, i log non vengono mai salvati sullo stesso disco che ospita i dati, ma su architetture separate.

Questo algoritmo si sviluppa in due fasi:

1. Si ripristinano i dati dall'ultimo **Dump** disponibile.
2. Attraverso il file di log, si parte dall'istante in cui termina la copia e si eseguono tutte le operazioni di REDO.

Controllo di concorrenza

La concorrenza è fondamentale: decine o centinaia di transazioni al secondo non possono essere seriali. L'esecuzione concorrente non controllata può causare anomalie:

- **Perdita di aggiornamento:** $w_1(x)$ viene sovrascritta da $w_2(x)$ senza aver letto il valore aggiornato.
- **Lettura sporca:** leggere un dato scritto da una transazione poi abortita.
- **Lettura inconsistente:** leggere lo stesso dato due volte e ottenere due valori diversi.
- **Aggiornamento fantasma:** modifiche che violano vincoli a causa di inserimenti concorrenti.

Serializzabilità

Uno **schedule** è una sequenza di operazioni di I/O di transazioni concorrenti, ad esempio: $S_1: r_1(x) r_2(z) w_1(x) w_2(z)$.

L'obiettivo è di ottenere schedule **serializzabili**, ovvero che producano lo stesso risultato di uno schedule seriale dove le transazioni sono separate, una alla volta, ad esempio:

$$S_2: r_0(x) r_0(y) w_0(x) r_1(y) r_1(x) w_1(y) r_2(x) r_2(y) r_2(z) w_2(z)$$

View-Serializzabilità (VSR)

Per capire se due operazioni (chiamate schedule) sono equivalenti, dobbiamo prima definire due concetti chiave su come le transazioni interagiscono con i dati:

- **Relazione "legge-da":** questa relazione si verifica quando una transazione legge un valore appena prodotto da un'altra. L'operazione di lettura $r_i(x)$ della transazione i *legge da* l'operazione di scrittura $w_j(x)$ della transazione j all'interno di uno schedule S se si verificano due condizioni:
 - la scrittura $w_j(x)$ avviene **prima** della lettura $r_i(x)$.
 - in mezzo a queste due operazioni **non deve esserci nessun'altra scrittura** $w_k(x)$ sullo stesso dato x .
- **Scrittura finale:** un'operazione $w_j(x)$ è la scrittura finale se è l'ultima volta che il dato x viene modificato prima che lo schedule S termini.

Due schedule S_i e S_j si dicono **view-equivalenti** ($S_i \approx_v S_j$) se hanno la stessa relazione *legge da* e le stesse *scritture finali*. Uno schedule è **view-serializzabile** se è view-equivalente ad un qualche schedule seriale. L'insieme degli schedule view-serializzabili è detto **VSR**.

View serializzabilità: esempi

- $S_3 : w_0(x) r_2(x) r_1(x) w_2(x) w_2(z)$
 $S_4 : w_0(x) r_1(x) r_2(x) w_2(x) w_2(z)$
 $S_5 : w_0(x) r_1(x) w_1(x) r_2(x) w_1(z)$
 $S_6 : w_0(x) r_1(x) w_1(x) w_1(z) r_2(x)$

 - S_3 è view-equivalente allo schedule seriale S_4 (e quindi è view-serializzabile)
 - S_5 non è view-equivalente a S_4 , ma è view-equivalente allo schedule seriale S_6 , e quindi è view-serializzabile
- $S_7 : r_1(x) r_2(x) w_1(x) w_2(x)$ (perdita di aggiornamento)
 $S_8 : r_1(x) r_2(x) w_2(x) r_1(x)$ (letture inconsistenti)
 $S_9 : r_1(x) r_1(y) r_2(z) r_2(y) w_2(y) w_2(z) r_1(z)$ (aggiornamento fantasma)

 - S_7, S_8, S_9 non view-serializzabili

Conflict-serializzabilità (CSR)

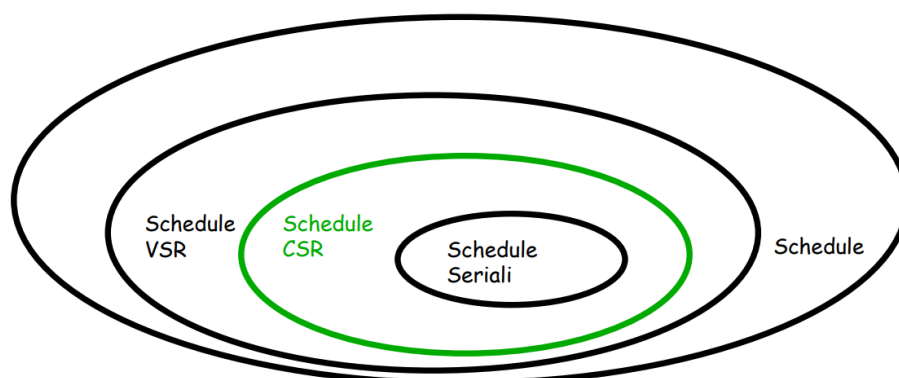
Il concetto base è capire quando due operazioni (schedule) entrano in conflitto. Un'azione a_i della transazione i è in **conflitto** con un'azione a_j della transazione j se si verificano contemporaneamente due condizioni: operano sullo **stesso oggetto** (stesso dato nel database) e **almeno una delle due** è un'operazione di **scrittura**.

Da questa regola derivano due casi in cui l'ordine in cui avvengono le operazioni è critico:

- **conflitto read-write** (rw o wr): una transazione legge e l'altra scrive. L'ordine è fondamentale perché determina se la lettura otterrà il dato vecchio o quello appena aggiornato.
- **conflitto write-write** (ww): entrambe le transazioni scrivono sullo stesso oggetto. L'ordine è fondamentale perché verrà salvato solo l'ultima scrittura.

Due schedule S_i e S_j sono **conflict-equivalenti** ($S_i \approx_c S_j$) se e solo se includono le *stesse operazioni* e ogni coppia di operazioni in conflitto compare nello *stesso ordine* in entrambi. Uno schedule è **conflict-serializzabile** se è *conflict-equivalente* ad un qualche schedule seriale. L'insieme degli schedule conflict-serializzabili è detto **CSR**.

Ogni schedule conflict-serializzabile è view-serializzabile ma non necessariamente viceversa.



È possibile stabilire se uno schedule è conflict-serializzabile per mezzo dei **grafi dei conflitti**:

- un nodo per ogni transazione t_i ;
- un arco orientato da t_i a t_j . Se c'è almeno un conflitto fra un'azione a_i e un'azione a_j tale che a_i precede a_j

Uno schedule è in CSR se e solo se il grafo è **aciclico**.

Lock

Per garantire la serializzabilità si usano i Lock:

- Tutte le letture sono precedute da *r_lock* (lock condiviso) e seguite da *unlock*.
- Tutte le scritture sono precedute da *w_lock* (lock esclusivo) e seguite da *unlock*.

Quando una transazione prima legge e poi scrive un oggetto, può adottare due strategie:

- chiedere subito un lock esclusivo, in modo da poter leggere e scrivere senza rischiare interruzioni o attese intermedie.
- chiedere un lock condiviso per effettuare una lettura e successivamente chiedere di convertire questo blocco condiviso in uno esclusivo. Questa “promozione” viene detta **lock escalation**.

Il **lock manager** riceve queste richieste dalle transazioni e decide se *accoglierle* o *rifiutarle*, sulla base della tavola dei conflitti.

Protocollo 2PL

Il Locking a due fasi è il protocollo usato da quasi tutti i sistemi poiché garantisce a priori la conflict-serializzabilità ma può causare **Deadlock** (stallo), risolvibili tramite timeout o rilevamento di cicli nel grafo delle attese. Si basa su due regole:

1. Una transazione acquisisce un lock (*fase crescente*);
2. Una volta rilasciato un lock, non può acquisirne altri (*fase decrescente*).

Stallo (deadlock)

Accade quando due transazioni detengono ciascuna una risorsa e aspettano la risorsa detenuta dall'altra, ad esempio:

- $t_1: read(x), write(x)$
- $t_2: read(y), write(x)$
- $S: r_lock_1(x), r_lock_2(y), read_1(x), read_2(y), w_lock_1(y), w_lock_2(x)$

Organizzazione fisica dei dati

I DBMS gestiscono i dati in memoria secondaria (persistente ma lenta) usando un **Buffer Manager** in memoria centrale per ridurre gli accessi al disco.

Il **Buffer** è un'area di memoria centrale gestita dal DBMS e condivisa fra le transazioni. È organizzato in **pagine** di dimensioni pari o multiple di quelle dei blocchi di memoria secondaria (1KB-100KB) ed è importantissimo per via della grande differenza di tempo di accesso tra memoria centrale e memoria secondaria.

Gestione del buffer

Il Buffer Manager regola il transito e il salvataggio delle pagine dati tra la memoria RAM e il disco fisico avvalendosi di queste tre operazioni e politiche fondamentali:

- **Fix**: richiede una pagina (se non c'è, la carica applicando politiche come LRU).
- **Force**: forza la scrittura su disco (sincrona).
- **Steal/No-Steal**: politiche di sostituzione delle pagine modificate prima del commit.

DBMS e file system

Il file system è il componente del sistema operativo che gestisce la memoria secondaria. I DBMS ne utilizzano le funzionalità, ma in misura limitata, per creare ed eliminare file e per leggere e scrivere singoli blocchi o sequenze di blocchi contigui.

L'organizzazione dei file, sia in termini di distribuzione dei record nei blocchi sia relativamente alla struttura all'interno dei singoli blocchi è gestita direttamente dal DBMS. dati memorizzati su disco sono organizzati in **file di record**.

Esistono diverse tecniche di memorizzazione fisica dei record di file su un disco magnetico:

- **Heap file** (file non ordinato): posiziona i record sul disco in modo non ordinato. Efficiente per inserimenti, inefficiente per ricerche.
- **Sorted file** (file sequenziale): ordina i record secondo il valore di un particolare campo. Buoni per ricerche, costosi per inserimenti.
- **Hashed file**: usa una funzione di hash per determinare la posizione dei record sul disco. Ottimo per uguaglianze ($A = v$), pessimo per intervalli ($v_1 < A < v_2$).

Indici di file

Un indice è una struttura ausiliaria per l'accesso efficiente ai record di un file sulla base dei valori di un campo detto **chiave**, ad esempio l'indice analitico di un libro: lista di coppie (termine, pagina), ordinata alfabeticamente sui termini, posta in fondo al libro e separabile da esso.

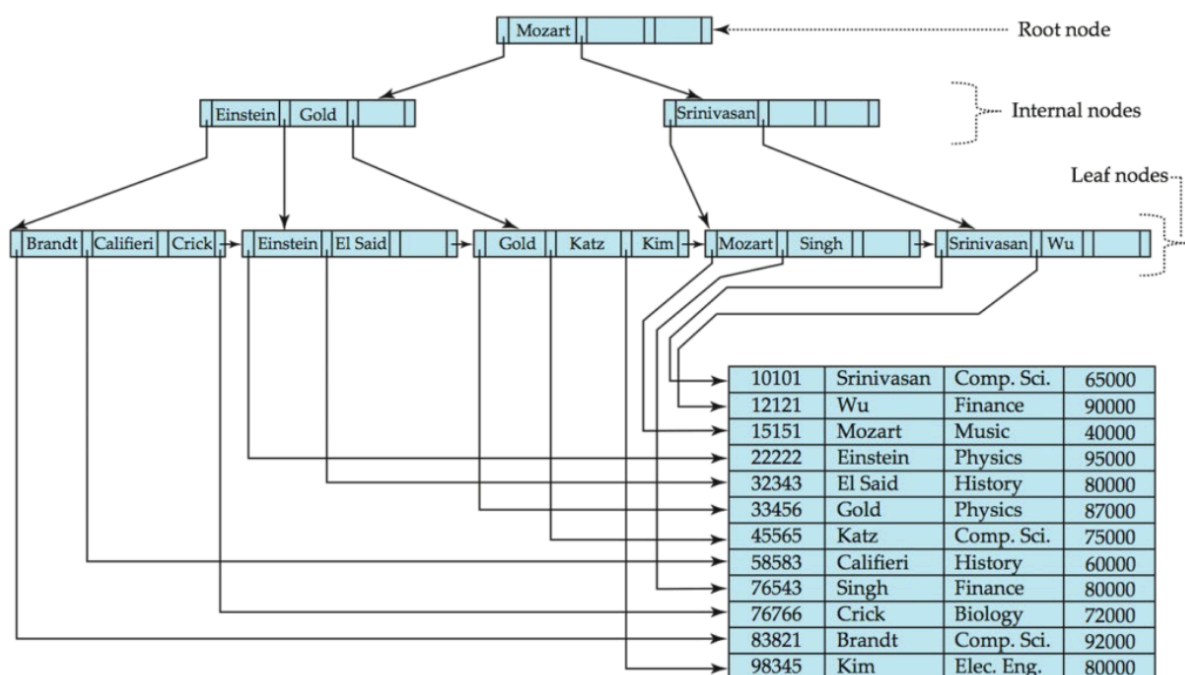
Esistono diversi tipi di indice:

- **Indice primario:** basato sul campo di ordinamento fisico del file. Solitamente **sparso** (una entry per blocco). Costo ricerca: $\log_2 n$ blocchi.
- **Indice secondario:** basato su campi non di ordinamento. Deve essere **denso** (una entry per record). Più costoso in spazio ma essenziale per ricerche su chiavi alternative.

Ogni file può avere al più un indice primario e un numero qualunque di indici secondari (su campi diversi).

Gli indici utilizzati dai DBMS sono più sofisticati: **indici dinamici multilivello**.

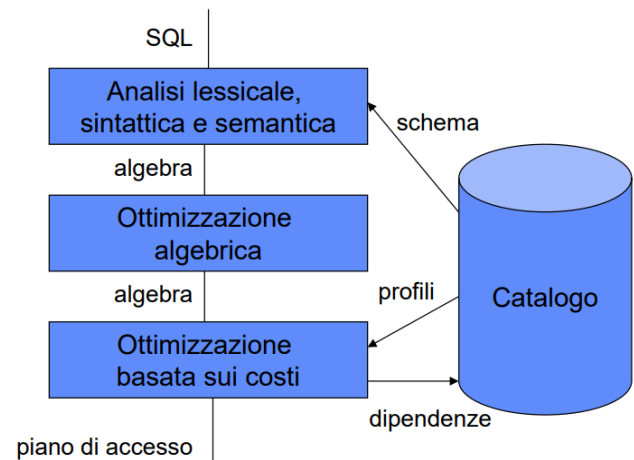
B-tree (alberi di ricerca bilanciati): Alberi di ricerca bilanciati pensati per la memoria secondaria. Ogni nodo corrisponde a una pagina. Riducono drasticamente l'altezza dell'albero e quindi gli accessi al disco ($t = \log_{fo} N$). Usati dalla maggior parte dei DBMS moderni (Oracle, SQL Server, MySQL InnoDB).



Ottimizzazione delle interrogazioni

Il **Query Optimizer** (o Ottimizzatore) traduce la query SQL in un piano di esecuzione: efficiente:

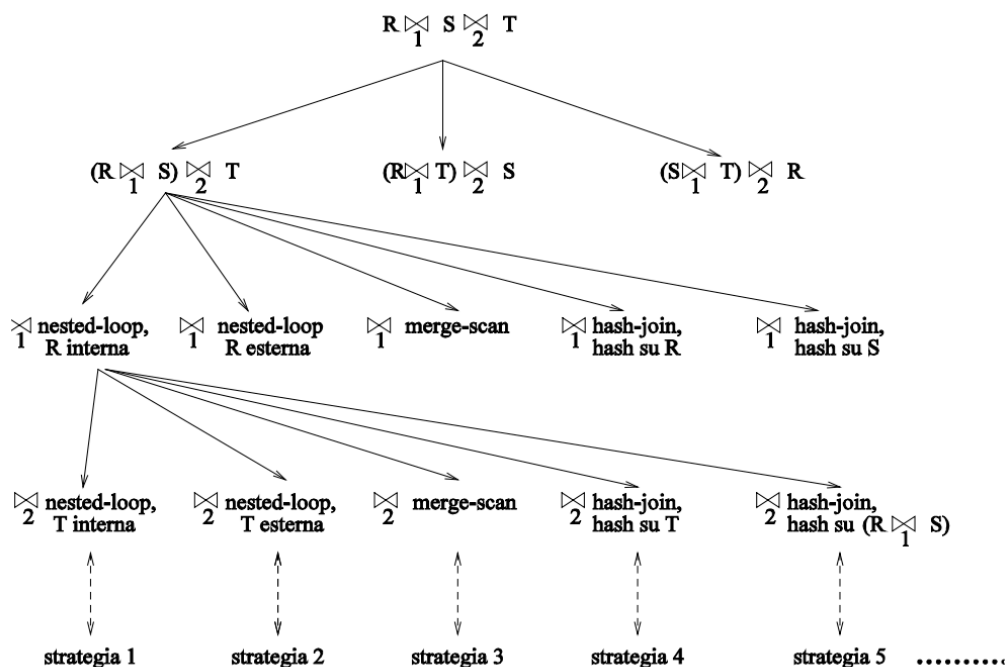
1. **Analisi:** lessicale, sintattica e semantica.
2. **Ottimizzazione algebrica:** trasformazioni logiche (es. anticipare le selezioni).
3. **Ottimizzazione basata sui costi:** scelta degli indici e algoritmi (es. *nested-loop* vs *hash-join*) basata su statistiche (cardinalità, distribuzione) mantenute nel catalogo.



Processo di ottimizzazione

Si costruisce un albero di decisione con le varie alternative, detti **piani di esecuzione**, si valuta il costo di ciascun piano e si sceglie il piano di costo minore. L'ottimizzatore trova di solito una "buona" soluzione, non necessariamente l'"ottimo".

Un albero di decisione



Progettazione fisica nel modello relazionale

La caratteristica comune dei DBMS relazionali è la **disponibilità degli indici**, la progettazione fisica spesso coincide con la scelta degli indici (oltre ai parametri strettamente dipendenti dal DBMS).

- Le **chiavi** (primarie) delle relazioni sono di solito coinvolte in **selezioni** e **join**: molti sistemi prevedono (oppure suggeriscono) di definire indici sulle chiavi primarie.
- Altri indici vengono definiti con riferimento ad altre selezioni o join *"importanti"*.

Se le prestazioni sono insoddisfacenti, si "tara" il sistema aggiungendo o eliminando indici

È utile verificare se e come gli indici sono utilizzati con il comando SQL **show plan (explain in MySQL)**.