

Niro Simone

Corso di
Basi di Dati



Prof.ssa Vittoria de Nitto Personè

Prof. Alessandro Pellegrini

CdL in Ingegneria Informatica - A.A. 2022/2023



INDICE

Sistema informativo e sistema informatico	1
Informazione, dato, DB e DBMS	1
<i>Modello dei dati, schema ed istanza</i>	1
Sistema di gestione di basi di dati (DBMS)	2
<i>Transazione</i>	2
<i>Architettura (semplificata) di un DBMS</i>	3
<i>Architettura standard a tre livelli per DBMS</i>	3
<i>Vantaggi e svantaggi dei DBMS</i>	3
I modelli logici dei dati	4
<i>Relazione: tre accezioni</i>	5
Il modello relazionale	5
<i>Tabelle e relazioni</i>	6
<i>Informazione incompleta e valore nullo</i>	6
Vincoli d'integrità	7
<i>Chiavi</i>	7
<i>Chiavi e valori nulli</i>	8
Algebra relazionale	10
<i>Operatori di base</i>	10
<i>Operatori insiemistici</i>	12
<i>Operatori di correlazione</i>	12
<i>Theta-join e equi-join</i>	14
<i>Equivalenza di espressioni</i>	15
<i>Viste (relazioni derivate)</i>	15
SQL – Concetti base	17
<i>Vincoli intrarelazionali</i>	17
<i>Vincoli interrelazionali e politica di reazione</i>	18
<i>Modifica degli schemi</i>	19
Interrogazioni in SQL	20
<i>Clausola select</i>	20
<i>Clausola from</i>	21
<i>Clausola where</i>	21

<i>Gestione dei valori nulli</i>	21
<i>Duplicati</i>	21
<i>Join interni ed esterni</i>	22
<i>Uso di variabili</i>	22
<i>Ordinamento</i>	23
<i>Interrogazioni di tipo insiemistico</i>	23
Interrogazioni nidificate	25
<i>Interrogazioni nidificate complesse</i>	25
Operatori aggregati	28
Interrogazioni con raggruppamento	29
<i>Predicati sui gruppi</i>	30
Modifica dei dati in SQL (insert, delete e update)	31
Introduzione alla progettazione	33
<i>Metodologie di progettazione</i>	33
Il modello Entità-Relazione	34
<i>Entità</i>	34
<i>Attributi</i>	36
<i>Relazioni (o associazioni)</i>	36
<i>Associazioni ricorsive e n-arie</i>	38
<i>Generalizzazioni</i>	38
Documentazione di schemi E-R	40
<i>Regole aziendali</i>	40
<i>Tecniche di documentazione</i>	41
Progettazione concettuale	42
<i>Raccolta e analisi dei requisiti</i>	42
<i>Progettazione concettuale</i>	42
<i>Rappresentazione concettuale dei dati</i>	45
<i>Pattern di progetto</i>	46
Strategie di progetto	51
<i>Strategia bottom-up</i>	51
<i>Strategia inside-out</i>	52
<i>Strategia mista</i>	52
<i>Qualità di uno schema concettuale</i>	53

Progettazione logica	54
<i>Analisi delle prestazioni su schemi E-R</i>	54
Ristrutturazione di schemi E-R	56
<i>Analisi delle ridondanze</i>	57
<i>Eliminazione delle generalizzazioni</i>	58
<i>Partizionamento/accorpamento di concetti</i>	59
<i>Scelta degli identificatori principali</i>	61
<i>Traduzione verso il modello relazionale</i>	62
Normalizzazione	66
<i>Dipendenze funzionali</i>	67
<i>Forma normale e normalizzazione</i>	68
<i>Proprietà delle decomposizioni</i>	68
<i>Verifiche di normalizzazione su entità</i>	69
<i>Verifiche di normalizzazione su associazioni</i>	70
Organizzazione fisica e gestione delle interrogazioni	72
<i>Gestore del buffer</i>	72
Gestione delle transazioni	74
<i>Controllo di affidabilità</i>	74
<i>Architettura del controllore dell'affidabilità</i>	75
<i>Organizzazione del log</i>	75
<i>Esecuzione delle transazioni e scrittura del log</i>	77
<i>Gestione dei guasti</i>	79
<i>Ripresa a freddo</i>	80
Controllo di concorrenza	82
<i>Teoria del controllo della concorrenza</i>	83
<i>View-equivalenza</i>	84
<i>Conflict-equivalenza</i>	85
<i>Controllo di concorrenza basato su locks</i>	86
<i>Controllo di concorrenza basato su timestamp</i>	89
<i>Lock management</i>	91

Sistema informativo e sistema informatico

Un **sistema informativo** di un'organizzazione è una combinazione di risorse (umane e materiali) e di procedure organizzate per la raccolta, l'archiviazione, l'elaborazione e lo scambio delle informazioni, necessarie alle attività:

- operative (informazioni di servizio),
- di programmazione e controllo (informazioni di gestione),
- di pianificazione strategica (informazioni di governo).

Il *sistema informativo automatizzato* è quella parte del sistema informativo in cui le informazioni sono raccolte, elaborate, archiviate e scambiate usando un sistema informatico.

Un **sistema informatico** è l'insieme delle tecnologie informatiche e della comunicazione a supporto delle attività di un'organizzazione.

Informazione, dato, DB e DBMS

Qual è la differenza tra *informazione* e *dato*? Per definizione, un **dato** è una *rappresentazione originaria, non interpretata*, di un evento o di un fenomeno effettuata attraverso dei *simboli*, o di un'altra forma di rappresentazione espressiva, legati ad un supporto. Invece, l'**informazione** è considerabile come la “chiave di lettura” dei dati.

Una **base di dati (DB)** rappresenta un aspetto del mondo reale, di interesse per qualche specifico scopo, ed è un insieme di dati coerenti e correlati, con un significato preciso per un particolare insieme di utenti.

All'interno di un DB, potrebbero presentarsi diversi problemi, come quello della **ridondanza** (che può tuttavia essere controllata) o dell'**inconsistenza dei dati** (se esistono varie copie degli stessi dati, è possibile che esse, in qualche momento, non siano uguali). Questi problemi vengono ridotti tramite la *condivisione dei dati*.

Un **DBMS (Data Base Management System)** è un insieme di programmi software che permette la creazione di un DB, la *manipolazione*, l'*interrogazione* e la *condivisione* dei dati, garantendo:

- affidabilità,
- efficienza,
- privatezza,
- efficacia.

Modello dei dati, schema ed istanza

Un **modello dei dati** è un tipo di astrazione dei dati che permette di definire le *proprietà* degli oggetti di interesse e le *relazioni* fra questi nascondendo i dettagli dell'implementazione. Inoltre, si possono distinguere i modelli in *concettuali* e *logici*.

Nelle basi di dati esiste una parte sostanzialmente invariante nel tempo, detta **schema** della base di dati, costituita dalle caratteristiche dei dati (e.g. titolo), e una parte variabile nel tempo, detta **istanza** della base di dati, costituita dai valori effettivi.

Sistema di gestione di basi di dati (DBMS)

Precisiamo le caratteristiche dei DBMS e delle basi di dati su cui si fondano le definizioni date in precedenza. Le basi di dati sono

- *grandi*, nel senso che possono avere anche dimensioni enormi e comunque in generale molto maggiori della memoria centrale disponibile. Di conseguenza, i DBMS devono prevedere una gestione dei dati in memoria secondaria.
- *persistenti*, cioè hanno un tempo di vita che non è limitato a quello delle singole esecuzioni dei programmi che le utilizzano.
- *condivise*, nel senso che applicazioni e utenti diversi possono poter accedere, secondo opportune modalità, a dati comuni. Per garantire l'accesso condiviso ai dati da parte di molti utenti che operano contemporaneamente, il DBMS dispone di un meccanismo apposito, detto *controllo di concorrenza*.

I DBMS:

- garantiscono la **privatezza** dei dati. Ciascun utente, opportunamente riconosciuto, viene abilitato a svolgere solo determinate azioni sui dati, attraverso meccanismi di *autorizzazione*.
- garantiscono la **affidabilità**, cioè la capacità del sistema di conservare sostanzialmente intatto il contenuto della base di dati (o almeno di permetterne la ricostruzione) in caso di malfunzionamenti hardware e software.
- sono *efficienti*, cioè capaci di svolgere le operazioni utilizzando un insieme di risorse (tempo e spazio) che sia accettabile per gli utenti.
- sono *efficaci*, in quanto sono capaci di rendere produttive, in ogni senso, le attività dei loro utenti.

Transazione

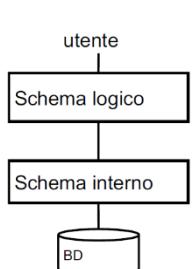
Una **transazione** consiste in un insieme di operazioni da considerare indivisibili (*operazioni atomiche*). Quindi, in caso di malfunzionamento l'operazione torna al punto di partenza, annullando tutto ciò che era stato già svolto riguardo quella specifica operazione atomica (ad esempio in caso di errori durante il versamento di denaro presso uno sportello bancario).

Il sistema deve poter garantire l'atomicità, al fine di garantire l'affidabilità e la consistenza dei dati. Tendenzialmente, le transazioni terminano con un'operazione nota come **commit** (impegno): nel momento in cui la transazione effettua il *commit* ed il DBMS glielo consente, è come se il sistema si stesse assumendo l'impegno che quel dato debba essere mantenuto, anche in presenza di guasti o malfunzionamenti.

Più nel dettaglio, le transazioni hanno due accezioni:

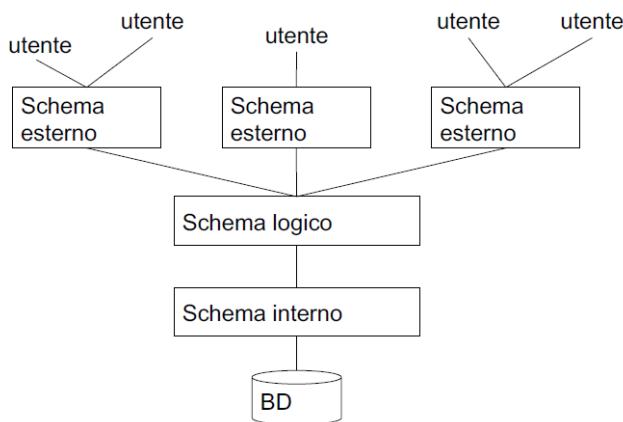
- per l'utente: programma a disposizione, da eseguire per realizzare una funzione di interesse;
- per il sistema: sequenza indivisibile di operazioni.

Architettura (semplificata) di un DBMS



Con l'architettura a livelli è resa possibile l'indipendenza dell'organizzazione fisica dei dati dal suo schema logico, consentendo quindi di cambiare l'organizzazione fisica mantenendo lo stesso schema logico.

Architettura standard a tre livelli per DBMS



Vantaggi e svantaggi dei DBMS

Andiamo a riassumere quali sono i pro e i contro per i DBMS. Tra i *pro*:

- dati come risorsa comune, base di dati come modello della realtà;
- gestione centralizzata con possibilità di standardizzazione ed “economia di scala” (fenomeno di riduzione dei costi e dell'aumento dell'efficienza legato ad un maggiore volume di produzione);
- disponibilità di servizi integrati;
- riduzione di ridondanze ed inconsistenze;
- indipendenza dei dati (favorisce lo sviluppo e la manutenzione delle applicazioni);

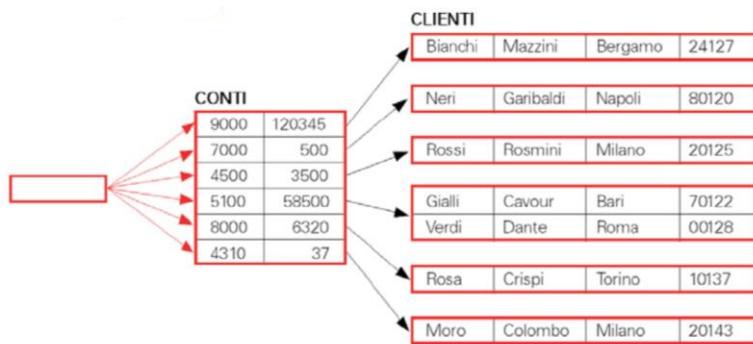
mentre tra i *contro* abbiamo:

- costo dei prodotti e della transizione verso di essi;
- impossibilità di scorporare le funzionalità (con conseguente riduzione di efficienza).

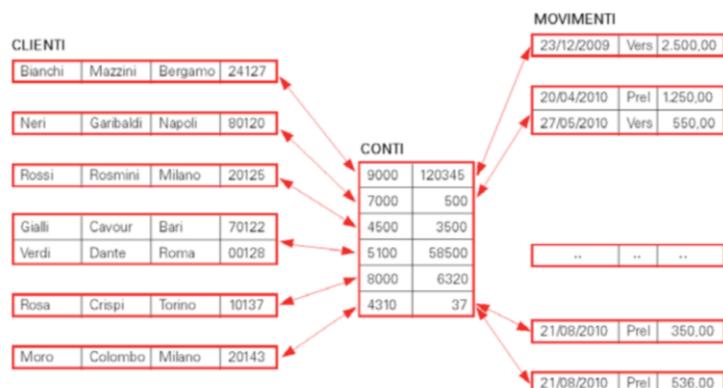
I modelli logici dei dati

Abbiamo a disposizione tre tipi di modelli logici tradizionali: gerarchico, reticolare e relazionale.

Il **modello gerarchico**, come ci suggerisce stesso il nome, prevede una struttura gerarchica dei dati caratterizzata da un nodo radice e dai suoi “figli”.



Nel **modello reticolare** l’albero del modello gerarchico viene sostituito da un grafo.



Infine, nel **modello relazionale** il riferimento tra strutture dati collegate tra di loro avviene tramite valori e non più tramite riferimenti fisici. In breve, si basa sul concetto matematico di relazione, che ha naturale rappresentazione per mezzo di tabelle.

studenti	Matricola	Cognome	Nome	Data di nascita
	6554	Rossi	Mario	05/12/1978
	8765	Neri	Paolo	03/11/1976
	9283	Verdi	Luisa	12/11/1979
	3456	Rossi	Maria	01/02/1978

esami	Studente	Voto	Corso
	3456	30	04
	3456	24	02
	9283	28	01
	6554	26	01

corsi	Codice	Titolo	Docente
	01	Analisi	Mario
	02	Chimica	Bruni
	04	Chimica	Verdi

Relazione: tre accezioni

Il termine “relazione” viene utilizzato in tre accezioni che, nei dettagli, differiscono in modo importante:

- *relazione matematica*, secondo la definizione normalmente data nella teoria degli insiemi elementare;
- *relazione* secondo la definizione del modello relazionale dei dati;
- *relazione*, come traduzione di “relationship”, costrutto del modello concettuale *Entità-Relazione* (in inglese “Entity-Relationship”) utilizzato per descrivere legami tra entità del mondo reale. A livello concettuale, questo tipo di relazione viene espressa in questo modo



Il modello relazionale

Supponiamo di avere l’insieme M delle persone che suonano uno strumento ($M = \{\text{Maria, Alberto, Alessandra}\}$) e l’insieme S degli strumenti musicali ($S = \{\text{Piano, Sax, Chitarra}\}$) e consideriamo il prodotto cartesiano $M \times S = \{(\text{Maria, Piano}), (\text{Alberto, Sax}), (\text{Alessandra, Chitarra})\}$.

Il **grado di una relazione** è il numero di domini che partecipano ad essa; quindi, ad esempio, se definissimo $r = \{(\text{Maria, Piano}), (\text{Alberto, Sax}), (\text{Alessandra, Chitarra})\}$ avremmo che il suo grado è due perché sono due i domini coinvolti.

Un **attributo** è una coppia *(nome, tipo di dato)* e una **tupla** è una sequenza ordinata di valori di attributi. Volendo fare un esempio, “Alberto” e “Sax” sono i valori degli attributi che potremmo definire come *(Nome, stringa)* e *(Strumento, stringa)*. Qualora volessimo indicare il valore dell’attributo A nella tupla t , possiamo utilizzare la notazione $t[A]$.

Lo **schema di relazione** lo possiamo definire in generale con la scrittura $R(A_1, A_2, \dots, A_m)$, dove tra parentesi troviamo un insieme di attributi (spesso indicato con $X \rightarrow R(X)$).

L’**istanza di relazione** (su $R(X)$) è un insieme di tuple (su X), come $r = \{t_1, \dots, t_i, \dots, t_n\}$, dove la generica tupla i -esima è una sequenza di m valori, ovvero $t_i = < v_1, \dots, v_j, \dots, v_m >$. Il singolo valore deve necessariamente appartenere al dominio di quell’attributo, ovvero $v_j \in \text{dom}(A_j)$.

La **cardinalità di un’istanza** è il numero di tuple che compongono l’istanza ad un dato istante di tempo.

L’**istanza di una base di dati** su un insieme di schemi $R = \{R_1(X_1), \dots, R_p(X_p)\}$ è a sua volta un insieme di relazioni $r = \{r_1, \dots, r_p\}$, con r_i relazione su R_i .

In generale, uno schema può essere rappresentato come segue

R	A ₁₁	...	A _j	...	A _m
t ₁	v ₁₁	...	v _{1j}	...	v _{1m}
...
t _i	v _{i1}	...	v _{ij}	...	v _{im}
...
t _n	v _{n1}	...	v _{nj}	...	v _{nm}

Tabelle e relazioni

Nel modello che vediamo in figura, ciascuno dei domini ha due **ruoli** diversi, distinguibili attraverso la posizione: la struttura è **posizionale!** Nel modello relazionale invece ciò non accade: a ciascun dominio è associato un nome *unico*, che ne descrive il “ruolo”.

Partite \subseteq string \times string \times int \times int

Juve	Lazio	3	1
Lazio	Milan	2	0
Juve	Roma	0	2
Roma	Milan	0	1

In una tabella che rappresenta una relazione l’ordinamento tra le righe e le colonne è irrilevante. Nel dettaglio, una tabella rappresenta una relazione se:

- ✓ le righe sono diverse tra loro;
- ✓ le intestazioni delle colonne sono diverse tra loro;
- ✓ i valori di ogni colonna sono fra loro omogenei.

Informazione incompleta e valore nullo

La struttura del modello relazionale è indubbiamente molto semplice e potente. Al tempo stesso, essa impone però un certo grado di rigidità:

- le informazioni sono rappresentate per mezzo di tuple;
- solo alcuni formati di tuple sono ammessi, ovvero quelli che corrispondono agli schemi di relazione.

In molti casi, i dati disponibili possono non corrispondere esattamente al formato previsto.

Per rappresentare in modo semplice la non disponibilità di valori, il concetto di relazione viene esteso prevedendo che una tupla possa assumere, su ciascun attributo, o un valore del dominio come visto finora, oppure un valore speciale, detto **valore nullo**, che denota appunto l’assenza di informazione. Modificheremo quindi la precedente scrittura $v_j \in \text{dom}(A_j)$ in $v_j \in \text{dom}(A_j) \cup \{\text{NULL}\}$.

I valori nulli possono essere dovuti a tre diverse motivazioni:

- *valore sconosciuto*: il valore nullo sostituisce un valore ordinario non noto alla base di dati in quell’istante di tempo;
- *valore inesistente*: il valore nullo denota l’inapplicabilità dell’attributo o l’inesistenza del valore;
- *valore senza informazione*: non sappiamo se il valore è inesistente oppure sconosciuto.

Tuttavia, la presenza di molteplici valori nulli in una relazione può addirittura generare dubbi sull'effettiva significatività e identità delle tuple! È quindi necessario moderare opportunamente la presenza dei valori nulli nelle nostre relazioni; in genere, quando si definisce una relazione, è possibile specificare che i valori nulli sono ammessi soltanto su alcuni attributi e non su altri.

Vincoli d'integrità

I **vincoli d'integrità** sono delle proprietà che devono essere soddisfatte dalle istanze che rappresentano informazioni corrette per l'applicazione. Ogni vincolo può essere visto come un *predicato* (una funzione booleana) che associa ad ogni istanza il valore vero o falso: se il predicato assume il valore *vero*, allora diciamo che l'istanza *soddisfa* il vincolo.

È possibile classificare i vincoli a seconda degli elementi di una base di dati che ne sono coinvolti. Distinguiamo due categorie:

- un vincolo è *intrarelazionale* se il suo soddisfacimento è definito rispetto a singole relazioni della base di dati; talvolta, il coinvolgimento riguarda le tuple (o addirittura i valori) separatamente le une dalle altre
 - un *vincolo di tupla* è un vincolo che può essere valutato su ciascuna tupla indipendentemente dalle altre;
 - un *vincolo di dominio* è un vincolo definito con riferimento a singoli valori (impone quindi una restrizione sul dominio dell'attributo).
- un vincolo è *interrelazionale* se coinvolge più relazioni.

Se un vincolo non viene rispettato parliamo di **violazione**. Di seguito abbiamo un esempio.

Esami	Studente	Voto	Lode	CORSO
	276545	32		01
	276545	30	e lode	02
	787643	27	e lode	03
	739430	24		04

Studenti	Matricola	Cognome	Nome
	276545	Rossi	Mario
	787643	Neri	Piero
	787643	Bianchi	Luca

Chiavi

Intuitivamente, potremmo dire che una chiave è un insieme di attributi utilizzato per identificare univocamente le tuple di una relazione. Per formalizzare la definizione, procediamo in due passi:

- una **superchiave** SK di uno schema $R(X)$ è un sottoinsieme di X tale che, $\forall t_i, t_l$ con $i \neq l$ e $i, l \in [1, n]$, $t_i[SK] \neq t_l[SK]$; in altre parole, i valori della SK in tuple diverse devono essere diversi. Essa è quindi in grado di identificare univocamente una tupla. Esiste sempre una superchiave banale che è l'insieme di tutti gli attributi, cioè X .

- una **chiave** K è una SK minimale, ovvero una SK dove non possono essere tolti degli attributi senza perdere la proprietà di univocità.

Nel seguente esempio

Matricola	Cognome	Nome	Nascita	Corso
4328	Rossi	Luigi	29/04/1994	Ing. Informatica
6328	Rossi	Dario	29/04/1994	Ing. Informatica
4766	Rossi	Luca	01/05/1995	Ing. Civile
4856	Neri	Luca	01/05/1995	Ing. Meccanica
5536	Neri	Luca	05/03/1993	Ing. Meccanica

possiamo notare che:

- l'insieme $\{Matricola\}$ è superchiave; è anche una superchiave minimale e, quindi, una chiave, in quanto contiene un solo attributo;
- l'insieme $\{Nome, Cognome, Nascita\}$ è superchiave; inoltre, nessuno dei suoi sottoinsiemi è superchiave: infatti esistono due tuple (la prima e la seconda) uguali su *Cognome* e *Nascita*, ecc; quindi, è un'altra chiave;
- l'insieme $\{Matricola, Corso\}$ è superchiave, ma non è una superchiave minimale, perché esiste un sottoinsieme proprio $\{Matricola\}$, esso stesso superchiave minimale e quindi $\{Matricola, Corso\}$ non è una chiave;
- l'insieme $\{Nome, Corso\}$ non è superchiave, perché sulla relazione compaiono due tuple, le ultime due, fra loro uguali sia su *Nome* che su *Corso*.

La superchiave va individuata ragionando sullo schema e non sull'istanza: potrebbe capitare che un insieme di attributi sia “casualmente” una chiave grazie a determinati valori dell'istanza! Tuttavia, a noi interessano le chiavi corrispondenti a vincoli di integrità, soddisfatti da tutte le relazioni lecite su un certo schema.

Su ciascuno schema di relazione può essere definita almeno una chiave e questo garantisce l'accessibilità a tutti i valori di una base di dati e la loro univoca identificabilità. Inoltre, permette di stabilire efficacemente quelle corrispondenze fra dati contenuti in relazioni diverse che caratterizzano il modello relazionale come “modello basato sui valori”.

Chiavi e valori nulli

In presenza di valori nulli, non è più vero che i valori delle chiavi permettono di identificare univocamente le tuple delle relazioni e di stabilire riferimenti fra tuple di relazioni diverse.

Esaminando la relazione seguente

Matricola	Cognome	Nome	Nascita	Corso
NULL	Rossi	Mario	NULL	Ing. Informatica
4766	Rossi	Luca	01/05/1995	Ing. Civile
4856	Neri	Luca	NULL	NULL
NULL	Neri	Luca	05/03/1993	Ing. Civile

notiamo problemi di due tipi. La prima tupla ha valori nulli su *Matricola* e *Nascita* e perciò almeno su un attributo di ciascuna chiave: questa tupla non è identificabile in nessun modo.

Inoltre, non è possibile, in altre relazioni della base di dati, fare riferimento a questa tupla, visto che ciò andrebbe fatto attraverso il valore di una chiave. Anche le ultime due tuple nella figura presentano un problema: nonostante ciascuna abbia una chiave completamente specificata, la presenza di valori nulli rende impossibile capire se le due tuple facciano riferimento allo stesso studente Luca Neri oppure a due studenti omonimi.

L'esempio ci suggerisce quindi la necessità di porre dei limiti alla presenza di valori nulli nelle chiavi delle relazioni. La soluzione che si adotta permette di garantire l'identificazione univoca di tutte le tuple e la possibilità di far riferimento a esse da parte di altre relazioni: su una delle chiavi, detta **chiave primaria**, si vieta la presenza di valori nulli; sulle altre, i valori nulli sono in genere (salvo necessità specifiche) ammessi. Gli attributi che costituiscono la chiave primaria vengono spesso evidenziati attraverso la sottolineatura.

Una **chiave esterna** FK (*foreign key*) di uno schema $R_1(X_1)$ che fa riferimento a $R_2(X_2)$ è un sottoinsieme proprio di X_1 ($\text{FK} \subset X_1$) tale per cui la cardinalità della chiave esterna è uguale alla cardinalità di K , dove K è chiave primaria di R_2 .

Il *vincolo di integrità referenziale* viene definito tra due relazioni al fine di mantenere coerenti i dati che fanno riferimento allo stesso concetto. Quindi, $t_1[\text{FK}] = t_2[K]$ oppure $t_1[\text{FK}] = \text{NULL}$. r_1 viene detta **tabella referente** e r_2 viene detta **tabella riferita**.

		Relazione referente		Vigile è FK in Infrazioni		
		Codice	Data	Vigile	Prov	Numero
		34321	1/2/95	3987	MI	39548K
		53524	4/3/95	3295	TO	E39548
		64521	5/4/96	3295	PR	839548
		73321	5/2/98	9345	PR	839548
Vigili		Matricola		Cognome	Nome	
Relazione riferita		3987		Rossi	Luca	
		3295		Neri	Piero	
		9345		Neri	Mario	
		7543		Mori	Gino	

Per poter mantenere la coerenza tra i dati potremmo eliminare una tupla, rischiando però di causare una violazione. Il comportamento standard sarebbe il rifiuto dell'operazione, ma abbiamo anche azioni compensative, come l'eliminazione in cascata e/o l'introduzione di valori nulli.

Algebra relazionale

Tipicamente, i linguaggi sulle basi di dati si distinguono in:

- DDL (*Data Definition Language*): consente di definire schemi logici, esterni, fisici e le autorizzazioni agli utenti;
- DML (*Data Manipulation Language*): consente di interrogare e aggiornare le istanze di BD.

Inoltre, i linguaggi si possono distinguere in *dichiarativi* (tendono a dire “che cosa” si vuole, non il come), che sono “result oriented”, e *procedurali* (tendono a dire “come” si vuole una cosa), che sono “task centered”. L'**algebra relazionale** è un linguaggio procedurale, basato su concetti di tipo algebrico. Sostanzialmente, esso è costituito da un insieme di operatori definiti su relazioni e che producono ancora relazioni come risultati.

Prima di passare alla definizione dei vari operatori, ci è utile specificare cosa si intende per interrogazione e aggiornamento. Per *aggiornamento* si intende una funzione che, data un’istanza di base di dati, produce un’altra istanza di base di dati sullo stesso schema. Per *interrogazione* si intende una funzione che, data un’istanza di base di dati, produce una relazione su un dato schema.

Nell’algebra relazionale troviamo

- operatori *di base*: selezione, proiezione, ridefinizione. I primi due svolgono operazioni che potremmo definire ortogonalì: la selezione produce un sottoinsieme delle tuple su tutti gli attributi, mentre la proiezione dà un risultato cui contribuiscono tutte le tuple, ma su un sottoinsieme degli attributi;
- operatori *insiemistici*: unione, intersezione, differenza;
- operatori *di correlazione*: prodotto cartesiano, join naturale, theta-join.

Operatori di base

L’operatore di **selezione** è denotato dal simbolo σ a pedice del quale viene indicata la “condizione di selezione” opportuna. Il risultato contiene le tuple dell’operando che soddisfano la condizione di selezione. In altre parole: la selezione $\sigma_F(r)$, in cui r è una relazione e F una *formula proposizionale* (ossia una formula ottenuta combinando, con i connettivi \vee , \wedge e \neg , condizioni atomiche), produce una relazione sugli stessi attributi di r che contiene le tuple di r su cui F è vera, ovvero

$$\sigma(r) = \{t \mid t \in r, F \text{ è vera su } t\}$$

L’operatore di **proiezione** è così definito: dati una relazione $r(X)$ e un sottoinsieme Y di X , la *proiezione* di r su Y (che si indica con $\pi_Y(r)$) è l’insieme di tuple su Y ottenute dalle tuple di r considerando solo i valori su Y , ovvero:

$$\pi_Y(r) = \{t[Y] \mid t \in r\}$$

In generale, possiamo dire che il risultato di una proiezione contiene al più tante tuple quante l’operando, ma può contenerne di meno. Notiamo anche che esiste un legame fra i vincoli di chiave e le proiezioni: $\pi_Y(r)$ contiene lo stesso numero di tuple di r se e solo se Y è superchiave per r . Infatti:

- se Y è superchiave, allora r non contiene tuple uguali su Y , quindi ogni tupla dà un contributo diverso alla proiezione;
- se la proiezione ha tante tuple quante l'operando, allora ciascuna tupla di r contribuisce alla proiezione con valori diversi, quindi r non contiene coppie di tuple uguali su Y (e questa è proprio la definizione di superchiave).

Vediamo un esempio di selezione ed un esempio di proiezione con meno tuple dell'operando.

IMPIEGATI

Cognome	Nome	Età	Stipendio
Rossi	Mario	25	2.000,00
Neri	Luca	40	3.000,00
Verdi	Nico	36	4.500,00
Rossi	Marco	40	3.900,00

IMPIEGATI

Cognome	Nome	Reparto	Capo
Rossi	Mario	Vendite	Gatti
Neri	Luca	Vendite	Gatti
Verdi	Mario	Personale	Lupi
Rossi	Marco	Personale	Lupi

$$\sigma_{\text{Età} > 30 \wedge \text{Stipendio} > 4.000,00}(\text{IMPIEGATI})$$

Cognome	Nome	Età	Stipendio
Verdi	Nico	36	4.500,00

$$\pi_{\text{Reparto}, \text{Capo}}(\text{IMPIEGATI})$$

Reparto	Capo
Vendite	Gatti
Personale	Lupi

L'operatore di **ridenominazione** “cambia il nome degli attributi”, lasciando inalterato il contenuto delle relazioni. Definiamolo in forma generale. Sia r una relazione definita sull'insieme di attributi X e sia Y un (altro) insieme di attributi con la stessa cardinalità. Inoltre, siano $A_1A_2 \dots A_k$ e $B_1B_2 \dots B_k$ rispettivamente un ordinamento per gli attributi in X e un ordinamento per quelli in Y . Allora la ridenominazione

$$\rho_{B_1B_2 \dots B_k \leftarrow A_1A_2 \dots A_k}(r)$$

contiene una tupla t' per ciascuna tupla t in r , definita come segue: t' è una tupla su Y e $t'[B_i] = t[A_i]$, per $i = 1, \dots, k$. La definizione conferma che ciò che cambia sono i nomi degli attributi, mentre i valori rimangono inalterati e vengono associati ai “nuovi” attributi. Nelle due liste indichiamo solo gli attributi che vengono ridenominati.

PATERNITÀ

Padre	Figlio
Adamo	Caino
Adamo	Abele
Abramo	Isacco
Isacco	Giacobbe

$\rho_{\text{Genitore} \leftarrow \text{Padre}}(\text{PATERNITÀ})$

Genitore	Figlio
Adamo	Caino
Adamo	Abele
Abramo	Isacco
Isacco	Giacobbe

Operatori insiemistici

L'**unione** di due relazioni r_1 e r_2 definite sullo stesso insieme di attributi X è indicata con $r_1 \cup r_2$ ed è una relazione ancora su X contenente le tuple che appartengono a r_1 oppure a r_2 , oppure a entrambe.

L'**intersezione** di $r_1(X)$ e $r_2(X)$ è indicata con $r_1 \cap r_2$ ed è una relazione su X contenente le tuple che appartengono sia a r_1 , sia a r_2 .

La **differenza** di $r_1(X)$ e $r_2(X)$ è indicata con $r_1 - r_2$ ed è una relazione su X contenente le tuple che appartengono a r_1 e non appartengono a r_2 .

LAUREATI

Matricola	Cognome	Età
7274	Rossi	37
7432	Neri	39
9824	Verdi	38

DIRIGENTI

Matricola	Cognome	Età
9297	Neri	56
7432	Neri	39
9824	Verdi	38

LAUREATI \cup DIRIGENTI

Matricola	Cognome	Età
7274	Rossi	37
7432	Neri	39
9824	Verdi	38
9297	Neri	56

LAUREATI \cap DIRIGENTI

Matricola	Cognome	Età
7432	Neri	39
9824	Verdi	38

LAUREATI – DIRIGENTI

Matricola	Cognome	Età
7274	Rossi	37

È importante specificare che le operazioni insiemistiche possono essere svolte solo su schemi che hanno gli stessi attributi.

Operatori di correlazione

L'operatore di join è il più caratteristico dell'algebra relazionale, in quanto è l'operatore che permette di correlare dati contenuti in relazioni diverse, confrontando i valori contenuti in esse e utilizzando quindi la caratteristica fondamentale del modello, quella di essere basato sui valori. Esistono due versioni dell'operatore join, comunque riconducibili l'una all'altra: la prima (il **join naturale**) utile per riflessioni di tipo astratto e la seconda (il **theta-join**) più rilevante dal punto di vista pratico.

Il **join naturale** è un operatore che correla dati in relazioni diverse, sulla base di valori uguali in attributi con lo stesso nome. Il risultato del join è costituito da una relazione sull'unione degli insiemi di attributi degli operandi e le sue tuple sono ottenute combinando le tuple degli operandi con valori uguali sugli attributi comuni.

In generale, il *join naturale* $r_1 \bowtie r_2$ di $r_1(X_1)$ e $r_2(X_2)$ è una relazione definita su X_1X_2 (cioè sull'unione degli insiemi X_1 e X_2), come segue:

$$r_1 \bowtie r_2 = \{t \text{ su } X_1X_2 \mid \exists t_1 \in r_1, t_2 \in r_2 : t[X_1] = t_1 \text{ e } t[X_2] = t_2\}$$

Nel caso in cui ciascuna tupla di ciascuno degli operandi contribuisce ad almeno una tupla del risultato, il **join** si dice **completo**: per ogni tupla t_1 di r_1 esiste una tupla t in $r_1 \bowtie r_2$ tale che $t[X_1] = t_1$ (e analogamente per r_2).

Questa proprietà non è sempre verificata, perché richiede una corrispondenza fra le tuple delle due relazioni: alcune tuple degli operandi possono non contribuire al risultato perché l'altra relazione non contiene tuple con gli stessi valori sull'attributo comune. In inglese tali **tuple** vengono chiamate **dangling** (“dondolanti”). Come caso limite è ovviamente possibile che nessuna delle tuple degli operandi sia combinabile, e allora il risultato del join è la relazione vuota.

Di seguito, un join con tuple *dangling* ed un join vuoto.

r_1	<table border="1"> <thead> <tr> <th>Impiegato</th> <th>Reparto</th> </tr> </thead> <tbody> <tr> <td>Rossi Neri Bianchi</td> <td>vendite produzione produzione</td> </tr> </tbody> </table>	Impiegato	Reparto	Rossi Neri Bianchi	vendite produzione produzione	r_2	<table border="1"> <thead> <tr> <th>Reparto</th> <th>Capo</th> </tr> </thead> <tbody> <tr> <td>produzione acquisti</td> <td>Mori Bruni</td> </tr> </tbody> </table>	Reparto	Capo	produzione acquisti	Mori Bruni	r_1	<table border="1"> <thead> <tr> <th>Impiegato</th> <th>Reparto</th> </tr> </thead> <tbody> <tr> <td>Rossi Neri Bianchi</td> <td>vendite produzione produzione</td> </tr> </tbody> </table>	Impiegato	Reparto	Rossi Neri Bianchi	vendite produzione produzione	r_2	<table border="1"> <thead> <tr> <th>Reparto</th> <th>Capo</th> </tr> </thead> <tbody> <tr> <td>concorsi acquisti</td> <td>Mori Bruni</td> </tr> </tbody> </table>	Reparto	Capo	concorsi acquisti	Mori Bruni
Impiegato	Reparto																						
Rossi Neri Bianchi	vendite produzione produzione																						
Reparto	Capo																						
produzione acquisti	Mori Bruni																						
Impiegato	Reparto																						
Rossi Neri Bianchi	vendite produzione produzione																						
Reparto	Capo																						
concorsi acquisti	Mori Bruni																						
$r_1 \bowtie r_2$	<table border="1"> <thead> <tr> <th>Impiegato</th> <th>Reparto</th> <th>Capo</th> </tr> </thead> <tbody> <tr> <td>Neri Bianchi</td> <td>produzione produzione</td> <td>Mori Mori</td> </tr> </tbody> </table>	Impiegato	Reparto	Capo	Neri Bianchi	produzione produzione	Mori Mori			$r_1 \bowtie r_2$	<table border="1"> <thead> <tr> <th>Impiegato</th> <th>Reparto</th> <th>Capo</th> </tr> </thead> <tbody> <tr> <td></td> <td></td> <td></td> </tr> </tbody> </table>	Impiegato	Reparto	Capo									
Impiegato	Reparto	Capo																					
Neri Bianchi	produzione produzione	Mori Mori																					
Impiegato	Reparto	Capo																					

All'estremo opposto, è possibile che ciascuna delle tuple di ciascuno degli operandi sia combinabile con tutte le tuple dell'altro. In tal caso, il risultato contiene un numero di tuple pari al prodotto delle cardinalità degli operandi e cioè $|r_1| \cdot |r_2|$, come nell'esempio successivo.

r_1	<table border="1"> <thead> <tr> <th>Impiegato</th> <th>Progetto</th> </tr> </thead> <tbody> <tr> <td>Rossi Neri Bianchi</td> <td>A A A</td> </tr> </tbody> </table>	Impiegato	Progetto	Rossi Neri Bianchi	A A A	r_2	<table border="1"> <thead> <tr> <th>Progetto</th> <th>Capo</th> </tr> </thead> <tbody> <tr> <td>A A</td> <td>Mori Bruni</td> </tr> </tbody> </table>	Progetto	Capo	A A	Mori Bruni
Impiegato	Progetto										
Rossi Neri Bianchi	A A A										
Progetto	Capo										
A A	Mori Bruni										
$r_1 \bowtie r_2$	<table border="1"> <thead> <tr> <th>Impiegato</th> <th>Reparto</th> <th>Capo</th> </tr> </thead> <tbody> <tr> <td>Rossi Neri Bianchi Rossi Neri Bianchi</td> <td>A A A A A A</td> <td>Mori Mori Mori Bruni Bruni Bruni</td> </tr> </tbody> </table>	Impiegato	Reparto	Capo	Rossi Neri Bianchi Rossi Neri Bianchi	A A A A A A	Mori Mori Mori Bruni Bruni Bruni				
Impiegato	Reparto	Capo									
Rossi Neri Bianchi Rossi Neri Bianchi	A A A A A A	Mori Mori Mori Bruni Bruni Bruni									

Ricapitolando, possiamo dire che il join di r_1 e r_2 contiene un numero di tuple compreso fra 0 e $|r_1| \cdot |r_2|$. Inoltre:

- se il join di r_1 e r_2 è completo, allora contiene almeno un numero di tuple pari al massimo fra $|r_1|$ e $|r_2|$;
- se $X_1 \cap X_2$ contiene una chiave per r_2 , allora il join di $r_1(X_1)$ e $r_2(X_2)$ contiene al più $|r_1|$ tuple;
- se $X_1 \cap X_2$ coincide con una chiave per r_2 e sussiste il vincolo di riferimento fra $X_1 \cap X_2$ in r_1 e la chiave di r_2 , allora il join di $r_1(X_1)$ e $r_2(X_2)$ contiene esattamente $|r_1|$ tuple.

Esiste anche una variante dell'operatore join chiamata **join esterno** (in inglese *outer join*), che prevede che tutte le tuple diano un contributo al risultato, eventualmente estese con valori nulli ove non vi siano controparti opportune. Distinguiamo:

- il join esterno *sinistro*, che estende solo le tuple del primo operando,
- il join esterno *destro*, che estende solo le tuple del secondo operando,
- il join esterno *completo*, che le estende tutte.

r_1	Impiegato	Reparto	r_2	Reparto	Capo
Rossi	vendite			produzione	Mori
Neri	produzione			acquisti	Bruni
Bianchi	produzione				
$r_1 \bowtie_{LEFT} r_2$		Impiegato	Reparto	Capo	
		Rossi	vendite	NULL	
		Neri	produzione	Mori	
		Bianchi	produzione	Mori	
$r_1 \bowtie_{RIGHT} r_2$		Impiegato	Reparto	Capo	
		Neri	produzione	Mori	
		Bianchi	produzione	Mori	
		NULL	acquisti	Bruni	
$r_1 \bowtie_{FULL} r_2$		Impiegato	Reparto	Capo	
		Rossi	vendite	NULL	
		Neri	produzione	Mori	
		Bianchi	produzione	Mori	
		NULL	acquisti	Bruni	

Theta-join e equi-join

Un prodotto cartesiano ha di solito ben poca utilità, in quanto concatena tuple non necessariamente correlate dal punto di vista semantico. In effetti, il prodotto cartesiano viene spesso seguito da una selezione, che centra l'attenzione su tuple correlate secondo le esigenze. Per questa ragione, viene spesso definito un operatore derivato (cioè esprimibile per mezzo di altri operatori), il ***theta-join***, come prodotto cartesiano seguito da una selezione, nel modo seguente (dove F è una formula proposizionale utilizzabile in una selezione e le relazioni r_1 e r_2 non hanno attributi in comune):

$$r_1 \bowtie_F r_2 = \sigma_F(r_1 \bowtie r_2)$$

IMPIEGATI		PROGETTI	
Impiegato	Progetto	Codice	Nome
Rossi	A	A	Venere
Neri	A	B	Marte
Neri	B		

$\sigma_{\text{Progetto}=\text{Codice}}(\text{IMPIEGATI} \bowtie \text{PROGETTI})$

Impiegato	Progetto	Codice	Nome
Rossi	A	A	Venere
Neri	A	A	Venere
Neri	B	B	Marte

Un theta-join in cui la condizione di selezione F sia una congiunzione di atomi di uguaglianza, con un attributo della prima relazione e uno della seconda, viene chiamato ***equi-join***.

Vediamo un esercizio: trovare le matricole dei capi i cui impiegati guadagnano tutti più di 40.

Ogni quantificatore universale (“per ogni”, “tutti”, ...) richiede una doppia negazione per essere verificato e si definisce prima negando la condizione e poi sottraendo il risultato all’insieme iniziale; quindi, prima neghiamo la condizione

$\pi_{Capo}(\text{Supervisione} \bowtie_{\text{Impiegato} = \text{Matricola}} (\sigma_{\text{Stipendio} < 40}(\text{Impiegati})))$

e poi sottraiamo all'insieme iniziale

$\pi_{Capo}(\text{Supervisione}) = \pi_{Capo}(\text{Supervisione} \bowtie_{\text{Impiegato} = \text{Matricola}} (\sigma_{\text{Stipendio} < 40}(\text{Impiegati})))$.

Il risultato sarà

Impiegati	Matricola	Nome	Età	Stipendio
	7309	Rossi	34	45
	5998	Bianchi	37	38
	9553	Neri	42	35
	5698	Bruni	43	42
	4076	Mori	45	50
	8123	Lupi	46	60

Supervisione	Impiegato	Capo
	7309	5698
	5998	5698
	9553	4076
	5698	4076
	4076	8123

Equivalenza di espressioni

Due espressioni sono **equivalenti** se producono lo stesso risultato qualunque sia l'istanza attuale della base di dati. L'equivalenza è importante in pratica perché i DBMS cercano di eseguire espressioni equivalenti a quelle date, ma meno "costose". L'equivalenza può essere:

- **assoluta** e, quindi, valere per qualsiasi schema $(E_1 \equiv E_2 \forall R)$ come $\pi_{AB}(\sigma_{A>0}(R)) \equiv \sigma_{A>0}(\pi_{AB}(R))$;
- **relativa** e, quindi, dipendere dallo schema $(E_1 \equiv_R E_2 \text{ se } E_1(r) = E_2(r), \text{ per ogni istanza } r \text{ di } R)$. Ad esempio, la seguente equivalenza

$$\pi_{AB}(R_1) \bowtie \pi_{AC}(R_2) \equiv_R \pi_{ABC}(R_1 \bowtie R_2)$$

sussiste se e solo se nello schema R l'intersezione fra gli insiemi di attributi di R_1 e R_2 è pari ad A , ovvero solo se $X_1 \cap X_2 = A$. Infatti, se ci fossero anche altri attributi, il join opererebbe solo su A nella prima espressione e su A e tali altri attributi nella seconda, con risultati in genere diversi.

Viste (relazioni derivate)

Spesso può risultare utile mettere a disposizione degli utenti rappresentazioni diverse per gli stessi dati. Nel modello relazionale, la tecnica prevista a questo scopo è quella delle **relazioni derivate**, ossia relazioni il cui contenuto è funzione del contenuto di altre relazioni.

In linea di principio, possono esistere due tipi di relazioni derivate:

- *viste materializzate*: relazioni derivate effettivamente memorizzate nella base di dati;
- *relazioni virtuali* (chiamate anche **viste**, senza ulteriori specificazioni): relazioni definite per mezzo di funzioni (espressioni nel linguaggio di interrogazione), non memorizzate sulla base di dati, ma utilizzabili nelle interrogazioni come se lo fossero.

Le viste vengono definite nei sistemi relazionali per mezzo di espressioni del linguaggio di interrogazione. Eventuali interrogazioni che si riferiscono alle viste vengono risolte sostituendo alla vista la sua definizione, componendo cioè le due interrogazioni.

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

L'uso delle viste può risultare vantaggioso per diversi ordini di motivi:

- un utente interessato solo a una porzione di una base di dati può evitare di considerare le componenti non rilevanti;
- espressioni molto complesse possono essere definite tramite viste, con vantaggi rilevanti soprattutto nel caso di presenza di sottoespressioni ripetute;
- attraverso la definizione di autorizzazioni di accesso rispetto alle viste, è possibile introdurre meccanismi di protezione della *privatezza*;
- in occasione di ristrutturazioni di una base di dati, può risultare conveniente definire viste che corrispondano a relazioni sostituite da altre e perciò non più presenti dopo la ristrutturazione stessa, ma ricavabili dalle nuove relazioni. In questo modo, le applicazioni scritte con riferimento alla versione precedente dello schema possono essere utilizzate sul nuovo senza bisogno di modifiche.

Vediamo un esempio senza la vista: trovare gli impiegati che hanno lo stesso capo di Rossi.

Afferenza	Impiegato	Reparto	Direzione	Reparto	Capo
	Rossi	A			Mori
	Neri	B		A	
	Bianchi	B		B	Bruni

	Impiegato	Reparto	Capo
	Rossi	A	Mori
	Neri	B	Bruni
	Bianchi	B	Bruni

La prima cosa che facciamo è $* := \sigma_{\text{Impiegato}='Rossi'}(\text{Afferenza} \bowtie \text{Direzione})$. Successivamente, $Y := \rho_{\text{Impiegato}=\text{Rossi}, \text{Reparto}=\text{Rossi} \leftarrow \text{Impiegato}, \text{Reparto}}(*)$ (se non lo facessimo, nel join successivo perderemmo tutto in quanto ci sarebbero tre attributi con lo stesso nome di altri tre), ottenendo:

ImpR	RepR	Capo
Rossi	A	Mori

Infine, scriviamo $\pi_{\text{Impiegato}}((\text{Afferenza} \bowtie \text{Direzione}) \bowtie Y)$ ottenendo:

Impiegato	Reparto	Capo	ImpR	RepR
Rossi	A	Mori	Rossi	A

SQL – Concetti base

Una tabella (ossia una relazione) SQL è costituita da una collezione ordinata di attributi e da un insieme (eventualmente vuoto) di vincoli. La sintassi per la **definizione delle tabelle** è:

```
create table NomeTabella
( NomeAttributo Dominio [ValoreDiDefault] [ Vincoli ]
  { , NomeAttributo Dominio [ ValoreDiDefault ] [ Vincoli ] }
  AltriVincoli
)
```

Ogni tabella viene quindi definita associandole un nome ed elencando gli attributi che ne compongono lo schema. Per ogni attributo si definiscono un nome, un dominio ed eventualmente un insieme di vincoli che devono essere rispettati dai valori dell'attributo. Dopo aver definito gli attributi, si possono definire i vincoli che coinvolgono più attributi sulla tabella.

Vincoli intrarelazionali

I più semplici *vincoli intrarelazionali* sono:

- **not null**: indica che il valore *nullo* non è ammesso come valore dell'attributo; in tal caso, il valore dell'attributo deve sempre essere specificato, tipicamente in fase di inserimento. Il vincolo viene specificato facendo seguire alla definizione dell'attributo la dichiarazione `not null`: `Cognome varchar(20) not null;`
- **unique**: si applica a un attributo o a un insieme di attributi di una tabella ed impone che i valori dell'attributo (o le ennuple di valori sull'insieme degli attributi) siano una (super)chiave, cioè che righe differenti della tabella non possano avere gli stessi valori. Viene fatta eccezione per il valore nullo, il quale può comparire su diverse righe senza violare il vincolo, in quanto si assume che i valori nulli siano tutti diversi tra di loro. La definizione di questo vincolo può avvenire in due modi: la prima alternativa può essere usata unicamente quando bisogna definire il vincolo su un solo attributo e si fa seguire la specifica dell'attributo alla parola chiave `unique` (analogamente a quanto avviene per la specifica del vincolo `not null`), mentre la seconda alternativa è necessaria quando il vincolo opera su un insieme di attributi. Dopo aver definito gli attributi della tabella, si usa la sintassi

```
unique ( Attributo { , Attributo } )
```

- **primary key**: come il vincolo *unique*, il vincolo *primary key* può essere definito direttamente su di un singolo attributo, oppure essere definito elencando più attributi che costituiscono l'identificatore. Gli attributi che fanno parte della chiave primaria non possono assumere il valore nullo; pertanto, la definizione di *primary key* implica per tutti gli attributi della chiave primaria una definizione di `not null`, che può essere omessa.

Vincoli interrelazionali e politica di reazione

Per quel che riguarda i *vincoli interrelazionali*, i più significativi sono i *vincoli di integrità referenziale*. In SQL per la loro definizione si usa l'apposito vincolo **foreign key**, ovvero di chiave esterna.

Questo vincolo crea un legame tra i valori di un attributo della tabella su cui è definito (che chiameremo *interna*) e i valori di un attributo di un'altra tabella (che chiameremo *esterna*). Il vincolo impone che per ogni riga della tabella interna il valore dell'attributo specificato, se diverso dal valore nullo, sia presente nelle righe della tabella esterna tra i valori del corrispondente attributo. L'unico requisito che la sintassi impone è che l'attributo cui si fa riferimento nella tabella esterna sia soggetto a un vincolo *unique*, cioè sia un identificatore della tabella; tipicamente l'attributo della tabella esterna cui si fa riferimento rappresenta in effetti la chiave primaria della tabella.

Il vincolo può essere definito in due modi, come i vincoli *unique* e *primary key*:

- se c'è un solo attributo coinvolto, si può usare il costrutto sintattico **references**, con il quale si specificano la tabella esterna e l'attributo della tabella esterna al quale l'attributo in questione deve essere legato.
- una definizione alternativa, necessaria quando il legame è rappresentato da un insieme di attributi, fa uso del costrutto **foreign key**, posto al termine della definizione degli attributi. Il costrutto elenca gli attributi della tabella coinvolti nel legame, cui segue la definizione dei corrispondenti attributi della tabella esterna mediante il costrutto **references**.

```
foreign key (Nome, Cognome)
    references Anagrafica (Nome, Cognome)
```

La corrispondenza tra gli attributi locali e quelli esterni avviene in base all'ordine: al primo attributo argomento di `foreign key` corrisponde il primo attributo argomento di `references`, e via via gli altri attributi.

Per tutti gli altri vincoli visti fino ad ora, quando il sistema rileva una violazione, il comando di aggiornamento viene rifiutato, segnalando l'errore all'utente. Per i vincoli di integrità referenziale, invece, SQL permette di scegliere altre reazioni da adottare quando viene rilevata una violazione. Vengono infatti offerte diverse alternative per rispondere alle violazioni generate da modifiche sulla tabella esterna; le operazioni sulla tabella esterna che possono introdurre delle violazioni sono le modifiche del valore dell'attributo riferito e la cancellazione di righe.

In particolare, per le operazioni di modifica, è possibile reagire in uno dei seguenti modi:

- **cascade**: il nuovo valore dell'attributo della tabella esterna viene riportato su tutte le corrispondenti righe della tabella interna;
- **set null**: all'attributo referente viene assegnato il valore nullo al posto del valore modificato nella tabella esterna;
- **set default**: all'attributo referente viene assegnato il valore di default al posto del valore modificato nella tabella esterna;

- **no action**: l'azione di modifica non viene consentita e il sistema non ha quindi bisogno di riparare la violazione.

Per le violazioni prodotte dalla cancellazione di un elemento della tabella esterna si ha a disposizione lo stesso insieme di reazioni:

- **cascade**: tutte le righe della tabella interna corrispondenti alla riga cancellata vengono cancellate;
- **set null**: all'attributo referente viene assegnato il valore nullo al posto del valore cancellato nella tabella esterna;
- **set default**: all'attributo referente viene assegnato il valore di default al posto del valore cancellato nella tabella esterna;
- **no action**: la cancellazione non viene consentita.

È possibile associare politiche diverse ai diversi eventi (per esempio utilizzare una politica di **cascade** per le modifiche e una politica di **set null** per le cancellazioni). La **politica di reazione** viene specificata immediatamente dopo il vincolo di integrità, secondo la seguente sintassi:

```
on { delete | update }
    ( cascade | set null | set default | no action )
```

Modifica degli schemi

SQL fornisce primitive per la manipolazione degli schemi delle basi di dati, che permettono di modificare le definizioni di tabelle precedentemente introdotte. I comandi che vengono utilizzati a questo fine sono:

- **alter**: permette di modificare domini e schemi di tabelle. Il comando può assumere varie forme

```
alter domain NomeDominio { set default ValoreDefault |
    drop default |
    add constraint DefVincolo |
    drop constraint NomeVincolo }
alter table NomeTabella {
    alter column NomeAttributo { set default NuovoDefault |
        drop default } |
    add constraint DefVincolo |
    drop constraint NomeVincolo |
    add column DefAttributo |
    drop column NomeAttributo }
```

Tramite **alter domain** e **alter table** è possibile aggiungere e rimuovere i vincoli e modificare i valori di default associati ai domini e agli attributi; è inoltre possibile aggiungere ed eliminare attributi e vincoli sullo schema di una tabella.

- **drop**: permette di rimuovere dei componenti, siano essi schemi, domini, tabelle, viste o asserzioni (ossia vincoli che non sono associati ad una tabella in particolare). Il comando rispetta la sintassi

```
drop { schema | domain | table | view | assertion } NomeElemento
    [ restrict | cascade ]
```

L'opzione **restrict** specifica che il comando non deve essere eseguito in presenza di oggetti *non vuoti*: uno schema non è rimosso se contiene tabelle o altri oggetti, un dominio non è

rimosso se appare in qualche definizione di tabella, una tabella non è rimossa se possiede delle righe o se è presente in qualche definizione di tabella o vista; infine, una vista non è rimossa se è utilizzata nella definizione di altre tabelle o viste. È l'opzione di default.

Con l'opzione **cascade** invece, tutti gli oggetti specificati devono essere rimossi. Quando si rimuove uno schema non vuoto, anche tutti gli oggetti che fanno parte dello schema vengono eliminati. In generale, l'opzione **cascade** attiva una reazione a catena, per cui tutti gli elementi che dipendono da un elemento rimosso vengono rimossi, e questo fino a che non si giunge in una situazione in cui non esistono dipendenze non risolte, ossia non vi sono elementi nella cui definizione compaiono elementi che sono stati rimossi.

Interrogazioni in SQL

SQL esprime le interrogazioni in modo *dichiarativo*, ovvero si specifica l'obiettivo dell'interrogazione e non il modo in cui ottenerlo. L'interrogazione SQL per essere eseguita viene passata all'ottimizzatore di interrogazioni (**query optimizer**), un componente del DBMS il quale analizza l'interrogazione e formula a partire da questa un'interrogazione equivalente nel linguaggio procedurale interno del sistema di gestione di basi di dati.

Le operazioni di interrogazione in SQL vengono specificate per mezzo dell'istruzione **select**, che ha la seguente struttura:

```
select ListaAttributi
      from ListaTabelle
            [ where Condizione ]
```

Le tre parti di cui si compone un'istruzione **select** vengono spesso chiamate *clausola select* (detta anche **target list**), *clausola from* e *clausola where*. Una descrizione più precisa della stessa sintassi è la seguente:

```
select AttrExpr [ [as] Alias ] {, AttrExpr [ [as] Alias ] }
      from Tabella [ [as] Alias ] {, Tabella [ [as] Alias ] }
            [ where Condizione ]
```

L'interrogazione SQL seleziona, tra le righe che appartengono al prodotto cartesiano delle tabelle elencate nella clausola **from**, quelle che soddisfano le condizioni espresse nell'argomento della clausola **where**. Ogni colonna del risultato viene eventualmente ridenominata con l'*Alias*, se questo compare dopo l'espressione.

Clausola select

La clausola **select** specifica gli elementi dello schema della tabella risultato. Come argomento della clausola **select** può anche comparire il carattere speciale *****, che rappresenta la selezione di tutti gli attributi delle tabelle elencate nella clausola **from**. Inoltre, possono comparire generiche espressioni sul valore degli attributi di ciascuna riga selezionata.

Clausola from

Quando si desidera formulare un'interrogazione che coinvolge righe appartenenti a più di una tabella, si pone come argomento della clausola `from` l'insieme delle tabelle alle quali si vuole accedere. Sul prodotto cartesiano delle tabelle elencate verranno applicate le condizioni contenute nella clausola `where`. Quindi, un join può essere specificato indicando in modo esplicito le condizioni che esprimono il legame tra le diverse tabelle. Ad esempio:

```
select Impiegato.Nome, Impiegato.Cognome,  
       Dipartimento.Città  
  from Impiegato, Dipartimento  
 where Impiegato.Dipart = Dipartimento.Nome
```

Notiamo l'uso dell'**operatore punto** per identificare le tabelle da cui vengono estratti gli attributi. È necessario specificare il nome della tabella quando le tabelle presenti nella clausola `from` posseggono più attributi con lo stesso nome. Qualora non vi sia possibilità di ambiguità, è possibile specificare l'attributo senza dichiarare la tabella di appartenenza.

Clausola where

La clausola `where` ammette come argomento un'espressione booleana costruita combinando predicati semplici con gli operatori `and`, `or` e `not`. Ciascun predicato semplice usa gli operatori `=`, `<>`, `≤` e `≥` per confrontare da un lato un'espressione costruita a partire dai valori degli attributi per la riga, e dall'altro lato un valore costante o un'altra espressione. La sintassi assegna la precedenza nella valutazione all'operatore `not`, ma non definisce una relazione di precedenza tra gli operatori `and` e `or`. Se è necessario esprimere un'interrogazione che richieda l'uso sia di `and` sia di `or`, conviene esplicitare l'ordine di valutazione mediante parentesi.

Oltre ai normali predicati di confronto relazionali, SQL mette a disposizione un operatore **like** per il confronto di stringhe, che permette di effettuare confronti con stringhe in cui compaiono i caratteri speciali `_` e `%`. Il primo carattere speciale può rappresentare nel confronto un carattere arbitrario, il secondo una stringa di un numero arbitrario (eventualmente anche nullo) di caratteri arbitrari. Un confronto `like 'ab%ba_'` sarà perciò soddisfatto da una qualsiasi stringa di caratteri che inizia con ab e che ha la coppia di caratteri ba prima dell'ultima posizione (per esempio abcdebac, oppure abbaf).

Gestione dei valori nulli

Per selezionare i termini con valori nulli si usa il predicato **is null**, la cui sintassi è

Attributo is [not] null

Il predicato risulta vero solo se l'attributo ha valore *nullo*. Il predicato **is not null** è la sua negazione.

Duplicati

Una significativa differenza tra SQL e algebra relazionale è data dalla gestione dei duplicati: in SQL si possono avere in una tabella più righe uguali (dette **duplicati**), ovvero righe con gli stessi valori per tutti gli attributi.

Per emulare il comportamento dell'algebra relazionale, sarebbe necessario effettuare l'eliminazione dei duplicati tutte le volte in cui si eseguono operazioni di proiezione. L'operazione di rimozione di duplicati è però molto costosa e spesso non necessaria, in quanto in molti casi il risultato non contiene duplicati. Per questo in SQL si è stabilito di permettere la presenza di duplicati all'interno delle tabelle, lasciando a chi scrive l'interrogazione il compito di specificare esplicitamente quando l'operazione di rimozione di duplicati è necessaria.

L'**eliminazione dei duplicati** è specificata con la parola chiave **distinct**, da porre immediatamente dopo la parola chiave **select**. La sintassi prevede che si possa anche specificare la parola chiave **all** al posto di **distinct**, indicando che si intende mantenere tutti i duplicati. L'indicazione di **all** è opzionale in quanto, come già detto, il mantenimento dei duplicati costituisce l'opzione di default.

Join interni ed esterni

Una sintassi alternativa per la **specificità dei join** permette di distinguere, tra le condizioni che compaiono nell'interrogazione, quelle che rappresentano condizioni di join e quelle che rappresentano condizioni di selezione sulle righe. In tal modo si possono anche specificare le forme esterne dell'operatore di join. La sintassi proposta è la seguente:

```
select AttrEspr [ [as] Alias ] {, AttrEspr [ [as] Alias ] }
  from Tabella [ [ as ] Alias ]
    { [ TipoJoin ] join Tabella [ [ as ] Alias ] on CondizioneDiJoin }
      [ where AltraCondizione ]
```

Mediante questa sintassi la condizione di join non compare come argomento della clausola **where**, ma viene invece spostata nell'ambito della clausola **from**, associata alle tabelle che vengono coinvolte nel join. Vediamo un esempio:

```
select I.Nome, Cognome, D.Città
  from Impiegato I join Dipartimento D
    on Dipart = D.Nome
```

Il parametro **TipoJoin** specifica qual è il tipo di join da usare, e a esso si possono sostituire i termini **inner** (interno, valore di default che può essere omesso), **right outer**, **left outer** o **full outer** (il qualificatore **outer** è opzionale). L'**inner join** rappresenta il tradizionale theta-join dell'algebra relazionale.

Uso di variabili

Abbiamo già visto come nelle interrogazioni SQL sia possibile associare un nome alternativo, detto *alias*, alle tabelle che compaiono come argomento della clausola **from**. Il nome viene usato per far riferimento alla tabella nel contesto dell'interrogazione. Questa funzionalità può essere sfruttata per far riferimento a una tabella in modo compatto, ricorrendo a brevi alias ed evitando così di scrivere per esteso il nome della tabella tutte le volte che ne viene richiesto l'uso.

Tutte le volte che si introduce un alias per una tabella si dichiara in effetti una **variabile** che rappresenta le righe della tabella di cui è alias. Quando una tabella compare una sola volta in un'interrogazione, non c'è differenza tra l'interpretare l'alias come uno pseudonimo

o come una nuova variabile. Quando una tabella compare invece più volte, è necessario considerare l'alias come una nuova variabile.

Vediamo un'interrogazione in cui vogliamo estrarre tutti gli impiegati che hanno lo stesso cognome (ma diverso nome) di impiegati del dipartimento *Produzione*.

```
select I1.Cognome, I1.Nome  
from Impiegato I1, Impiegato I2  
where I1.Cognome = I2.Cognome and  
    I1.Nome <> I2.Nome and  
    I2.Dipart = 'Produzione'
```

Ordinamento

Mentre una relazione è costituita da un insieme non ordinato di tuple, nell'uso reale delle basi di dati sorge spesso il bisogno di costruire un ordine sulle righe delle tabelle. SQL permette di specificare un **ordinamento delle righe** del risultato di un'interrogazione tramite la clausola **order by**, con la quale si chiude l'interrogazione. La clausola rispetta la seguente sintassi:

```
order by AttrDiOrdinamento [ asc | desc ]  
       { , AttrDiOrdinamento [ asc | desc ] }
```

Per prima cosa, le righe vengono ordinate in base al primo attributo nell'elenco. Per righe che hanno lo stesso valore del primo attributo, si considerano i valori degli attributi successivi, in sequenza. L'ordine su ciascun attributo può essere ascendente o descendente, a seconda che si sia usato il qualificatore **asc** o **desc**. Se il qualificatore è omesso, si assume un ordinamento ascendente.

Interrogazioni di tipo insiemistico

SQL mette a disposizione anche degli operatori insiemistici, simili a quelli disponibili nell'algebra relazionale. Gli operatori disponibili sono gli operatori **union** (unione), **intersect** (intersezione) ed **except** (chiamato anche minus, differenza), di significato analogo ai corrispondenti operatori dell'algebra relazionale.

Si noti che ogni interrogazione che faccia uso degli operatori di intersezione e differenza può essere espressa utilizzando altri costrutti del linguaggio. Al contrario, l'operatore di unione arricchisce il potere espressivo di SQL e permette di scrivere interrogazioni altrimenti non formulabili.

La sintassi per l'uso degli operatori insiemistici è la seguente:

```
SelectSQL { ( union | intersect | except ) [ all ] SelectSQL }
```

Gli operatori insiemistici, al contrario del resto del linguaggio, assumono come default di eseguire una eliminazione dei duplicati. Qualora nell'interrogazione si voglia adottare una diversa interpretazione degli operatori e si vogliano utilizzare gli operatori insiemistici che preservano i duplicati, sarà sufficiente utilizzare l'operatore con la parola chiave **all**.

Un'altra osservazione è che SQL non richiede che gli schemi su cui vengono effettuate le operazioni insiemistiche siano identici (come è invece richiesto dall'algebra relazionale), ma

solo che gli attributi siano in pari numero e che abbiano domini compatibili. La corrispondenza tra gli attributi non si basa sul nome, ma sulla posizione degli attributi. Se gli attributi hanno nome diverso, il risultato normalmente usa i nomi del primo operando.

Di seguito un'interrogazione in cui vogliamo estrarre i nomi e i cognomi degli impiegati.

```
select Nome  
from Impiegato  
      union  
select Cognome  
from Impiegato
```

Di seguito un'interrogazione in cui vogliamo estrarre i cognomi di impiegati che sono anche nomi.

```
select Nome  
from Impiegato  
      intersect  
select Cognome  
from Impiegato
```

Di seguito un'interrogazione in cui vogliamo estrarre i nomi degli impiegati che non sono cognomi di qualche impiegato.

```
select Nome  
from Impiegato  
      except  
select Cognome  
from Impiegato
```

Interrogazioni nidificate

SQL ammette anche l'uso di predicati con una struttura più complessa, in cui si confronta un valore con il risultato dell'esecuzione di un'interrogazione SQL. L'interrogazione che viene usata per il confronto viene definita direttamente nel predicato interno alla clausola `where`. Si parla in questo caso di **interrogazioni nidificate**.

Nel caso più tipico, l'espressione che compare come primo membro del confronto è il semplice nome di un attributo. Se in un predicato si confronta un attributo con il risultato di un'interrogazione, sorge il problema di disomogeneità dei termini del confronto: da una parte abbiamo il risultato dell'esecuzione di un'interrogazione SQL (in generale un insieme di valori), mentre dall'altro abbiamo il valore dell'attributo per la particolare riga. La soluzione offerta da SQL consiste nell'estendere, con le parole chiave **all** o **any**, i normali operatori di confronto.

La parola chiave **any** specifica che la riga soddisfa la condizione se risulta vero il confronto (con l'operatore specificato) tra il valore dell'attributo per la riga e almeno uno degli elementi restituiti dall'interrogazione. La parola chiave **all** invece specifica che la riga soddisfa la condizione solo se tutti gli elementi restituiti dall'interrogazione nidificata rendono vero il confronto. La sintassi richiede la compatibilità di dominio tra l'attributo restituito dall'interrogazione nidificata e l'attributo con cui avviene il confronto.

Vediamo un esempio di interrogazione che seleziona le righe di *IMPIEGATO* per cui il valore dell'attributo *Dipart* è uguale ad almeno uno dei valori dell'attributo *Nome* delle righe di *DIPARTIMENTO*.

```
select *
from Impiegato
where Dipart = any (select Nome
                     from Dipartimento
                     where Città = 'Firenze')
```

Per rappresentare il controllo di appartenenza e di esclusione rispetto ad un insieme, SQL mette a disposizione due appositi operatori, **in** e **not in**, i quali risultano del tutto identici agli operatori `= any` e `<> all`.

Interrogazioni nidificate complesse

Un'interpretazione molto semplice e intuitiva delle interrogazioni nidificate consiste nell'assumere che l'interrogazione nidificata venga eseguita prima di analizzare le righe dell'interrogazione esterna.

Talvolta però l'interrogazione nidificata fa riferimento al contesto dell'interrogazione che la racchiude; tipicamente ciò accade tramite una variabile definita nell'ambito della query più esterna e usata nell'ambito della query più interna (si parla di un **passaggio di binding** da un contesto all'altro). La presenza del meccanismo di passaggio di binding arricchisce il potere espressivo di SQL e in questo caso l'interpretazione semplice data precedentemente alle query nidificate non vale più.

La nuova interpretazione è la seguente: per ogni riga della query esterna, valutiamo per prima cosa la query nidificata e poi calcoliamo il predicato a livello di riga sulla query esterna.

Per quanto riguarda la **visibilità** (o *scope*) delle variabili SQL, vale la restrizione che una variabile è usabile solo nell'ambito della query in cui è definita o nell'ambito di una query nidificata (a un qualsiasi livello) all'interno di essa. Se un'interrogazione possiede interrogazioni nidificate allo stesso livello (su predicati distinti), le variabili introdotte nella clausola `from` di una query non potranno essere usate nell'ambito dell'altra query. Una interrogazione come la seguente, per esempio, è scorretta, in quanto utilizza la variabile `D1` dove non è visibile:

```
select *
from Impiegato
where Dipart in (select Nome
                  from Dipartimento D1
                  where Nome = 'Produzione') or
      Dipart in (select Nome
                  from Dipartimento D2
                  where D1.Città = D2.Città)
```

Introduciamo ora l'operatore logico **`exists`**. Questo operatore ammette come parametro un'interrogazione nidificata e restituisce il valore vero solo se l'interrogazione fornisce un risultato non vuoto (corrisponde al quantificatore esistenziale della logica). Questo operatore può essere usato in modo significativo solo quando si ha un passaggio di binding tra l'interrogazione esterna e quella nidificata. In questo caso non risulta possibile eseguire l'interrogazione nidificata prima di valutare l'interrogazione più esterna: si richiede invece che venga prima valutata l'interrogazione esterna e per ogni singola riga esaminata nell'ambito dell'interrogazione esterna si deve valutare l'interrogazione nidificata.

Di seguito, un esempio in cui vogliamo estrarre le persone che hanno degli omonimi.

```
select *
from Persona P
where exists (select *
              from Persona P1
              where P1.Nome = P.Nome and
                    P1.Cognome = P.Cognome and
                    P1.CodFiscale <> P.CodFiscale)
```

Supponendo invece di voler estrarre le persone che *non* hanno degli omonimi, allora avremmo:

```
select *
from Persona P
where not exists (select *
                   from Persona P1
                   where P1.Nome = P.Nome and
                         P1.Cognome = P.Cognome and
                         P1.CodFiscale <> P.CodFiscale)
```

L'interpretazione è analoga a quella della precedente interrogazione, con l'unica che differenza che il predicato è soddisfatto nel caso che il risultato dell'interrogazione nidificata sia vuoto. Questa interrogazione poteva anche essere implementata con una differenza che sottraesse ai nomi e cognomi delle persone i nomi e cognomi delle persone che possiedono un omonimo, determinati tramite un join.

Si consideri una base di dati con una tabella *CANTANTE*(Nome, Canzone) e una tabella *AUTORE*(Nome, Canzone). Vogliamo estrarre i cantautori puri, ovvero i cantanti che hanno eseguito solo canzoni di cui erano anche autori.

```
select Nome
from Cantante
where Nome not in
  (select Nome
   from Cantante C
   where Nome not in
     (select Nome
      from Autore
      where Autore.Canzone=C.Canzone))
```

La prima interrogazione nidificata non ha alcun legame con l'interrogazione esterna, e può quindi essere eseguita in modo del tutto indipendente. L'interrogazione al livello successivo invece presenta un legame. L'esecuzione dell'interrogazione può così avvenire seguendo queste fasi:

1. L'interrogazione `select Nome from Cantante C ...` legge tutte le righe della tabella *CANTANTE*.
2. Per ognuna delle righe di *C* viene valutata l'interrogazione più interna, che restituisce i nomi degli autori della canzone il cui titolo compare nella riga di *C* che viene considerata. Se il nome del cantante non compare tra gli autori (e quindi il cantante non è un cantautore puro), allora il nome viene selezionato.
3. Dopo che l'interrogazione nidificata ha terminato di analizzare le righe di *C*, costruendo la tabella contenente i nomi dei cantanti che non sono cantautori puri, viene eseguita l'interrogazione più esterna, la quale restituirà tutti i nomi di cantanti che non compaiono nella tabella ottenuta come risultato dell'interrogazione nidificata.

La correttezza di questa esecuzione appare evidente se si considera che la query può essere espressa in modo equivalente tramite l'operatore insiemistico `except (differenza)`:

```
select Nome
from Cantante
except
select Nome
from Cantante C
where Nome not in (select Nome
                    from Autore
                    where Autore.Canzone=C.Canzone)
```

Di fatto, una differenza è sempre possibile esprimerla come un'interrogazione nidificata.

Operatori aggregati

Gli **operatori aggregati** costituiscono una delle più importanti estensioni di SQL rispetto all'algebra relazionale. In algebra relazionale, infatti, tutte le condizioni vengono valutate su una tupla alla volta, ma nei contesti reali spesso viene richiesto di valutare delle proprietà che dipendono da insiemi di tuple.

Gli operatori aggregati vengono gestiti come un'estensione delle normali interrogazioni. Prima viene normalmente eseguita l'interrogazione, considerando solo le parti `from` e `where`. L'operatore aggregato viene poi applicato alla tabella contenente il risultato dell'interrogazione.

Lo standard SQL prevede cinque operatori aggregati: `count`, `sum`, `max`, `min` e `avg`.

L'**operatore `count`** usa la seguente sintassi:

```
count ( ( * | [ distinct | all ] ListaAttributi ) )
```

La prima opzione (*) restituisce il numero di righe; l'opzione `distinct` restituisce il numero di diversi valori degli attributi in *ListaAttributi*; l'opzione `all` invece restituisce il numero di righe che possiedono valori diversi dal valore nullo per gli attributi in *ListaAttributi*. Se si specifica un attributo e si omette `distinct` o `all`, si assume `all` come default.

Ad esempio, per estrarre il numero di diversi valori dell'attributo *Stipendio* fra tutte le righe di *IMPIEGATO*:

```
select count(distinct Stipendio)
      from Impiegato
```

Per estrarre il numero di righe che possiedono un valore non nullo per l'attributo *Nome*:

```
select count(all Nome)
      from Impiegato
```

Gli altri quattro operatori aggregati invece ammettono come argomento un attributo o un'espressione, eventualmente preceduta dalle parole chiave `distinct` o `all`. Le funzioni aggregate `sum` e `avg` ammettono come argomento solo espressioni che rappresentano valori numerici o intervalli di tempo. Le funzioni `max` e `min` richiedono solamente che sull'espressione sia definito un ordinamento, per cui si possono applicare anche su stringhe di caratteri o su istanti di tempo.

```
( sum | max | min | avg ) ([ distinct | all ] AttrExpr )
```

Gli operatori si applicano sulle righe che soddisfano la condizione presente nella clausola `where` e hanno il seguente significato:

- **sum**: restituisce la somma dei valori posseduti dall'espressione;
- **max** e **min**: restituiscono rispettivamente il valore massimo e minimo;
- **avg**: restituisce la media dei valori (vale a dire, il risultato della divisione di `sum` per `count`).

Le parole chiave `distinct` e `all` hanno il significato che abbiamo già visto: `distinct` elimina i duplicati, mentre `all` trascura solo i valori nulli; l'uso di `distinct` o `all` con gli operatori `max` e `min` non ha effetto sul risultato.

Possiamo anche valutare diversi operatori aggregati nell'ambito della stessa interrogazione. Ad esempio, supponiamo di voler estrarre gli stipendi minimo, massimo e medio fra quelli di tutti gli impiegati:

```
select min(Stipendio), max(Stipendio), avg(Stipendio)
from Impiegato
```

Inoltre, la valutazione degli operatori aggregati può avvenire su una generica interrogazione, come sul risultato di un join.

Interrogazioni con raggruppamento

Molto spesso sorge l'esigenza di applicare l'operatore aggregato separatamente a sottoinsiemi di righe. Per poter utilizzare in questo modo l'operatore aggregato, SQL mette a disposizione la clausola `group by`, che permette di specificare come dividere le tabelle in sottoinsiemi. La clausola ammette come argomento un insieme di attributi e l'interrogazione raggrupperà le righe che possiedono gli stessi valori per questo insieme di attributi.

Analizziamo come viene eseguita un'interrogazione SQL che fa uso della clausola `group by`. Per prima cosa l'interrogazione viene eseguita come se la clausola `group by` non esistesse, selezionando gli attributi che appaiono come argomento della clausola `group by` o che compaiono all'interno dell'espressione argomento dell'operatore aggregato. La tabella ottenuta viene poi analizzata, dividendo le righe in insiemi caratterizzati dallo stesso valore degli attributi che compaiono come argomento della clausola `group by`.

Dopo che le righe sono state raggruppate in sottoinsiemi, l'operatore aggregato viene applicato separatamente su ogni sottoinsieme. Il risultato dell'interrogazione è costituito da una tabella con righe che contengono l'esito della valutazione dell'operatore aggregato affiancato al valore dell'attributo che è stato usato per l'aggregazione.

La sintassi SQL impone che, in un'interrogazione che fa uso della clausola `group by`, possa comparire come argomento della `select` solamente un sottoinsieme degli attributi usati nella clausola `group by`. Per questi attributi, infatti, ciascuna tupla del sottoinsieme sarà caratterizzata dallo stesso valore. Ad esempio, supponiamo di voler calcolare la media dei redditi dei figli:

```
select padre, avg(f.reddito), p.reddito
from persone f join paternita on figlio = f.nome join
     persone p on padre = p.nome
group by padre
```



```
select padre, avg(f.reddito), p.reddito
from persone f join paternita on figlio = f.nome join
     persone p on padre = p.nome
group by padre, p.reddito
```

Predicati sui gruppi

Abbiamo visto come tramite la clausola group by le righe possano venire raggruppate in sottoinsiemi. Un'applicazione può aver bisogno di considerare solo i sottoinsiemi che soddisfano certe condizioni. Se le condizioni che i sottoinsiemi devono soddisfare sono verificabili al livello delle singole righe, allora basta porre gli opportuni predicati come argomento della clausola where. Se invece le condizioni sono delle condizioni di tipo aggregato, sarà necessario utilizzare un nuovo costrutto, la **clausola having**.

La clausola having descrive le condizioni che si devono applicare al termine dell'esecuzione di un'interrogazione che fa uso della clausola group by. Ogni sottoinsieme di righe costruito dalla group by fa parte del risultato dell'interrogazione solo se il predicato argomento della having risulta soddisfatto.

Supponiamo di voler sapere quali sono i padri i cui figli hanno un reddito medio maggiore di 25 (e di mostrare padre e reddito medio dei figli): come viene eseguita l'interrogazione?

- 1 select padre, f.reddito
from persone f join paternita on figlio = nome
- 2 group by padre
- 3 select padre, avg(f.reddito)
from persone f join paternita on
figlio = nome
group by padre
having avg(f.reddito) > 25

Per sapere quali predicati di un'interrogazione che fa uso del raggruppamento vanno dati come argomento della clausola where e quali come argomenti della clausola having, basta rispettare il seguente criterio: solo i predicati in cui compaiono operatori aggregati devono essere argomento della clausola having.

Modifica dei dati in SQL (insert, delete e update)

Il **comando di inserimento** di righe nella base di dati presenta due sintassi alternative:

```
insert into NomeTabella [ ListaAttributi ]
    { values( ListaValori ) |
      SelectSQL }
```

La prima forma permette di inserire *singole* righe all'interno delle tabelle e l'argomento della clausola *values* rappresenta esplicitamente i valori degli attributi della singola riga. La seconda forma invece permette di aggiungere degli *insiemi* di righe, estratti dal contenuto della base di dati.

Se in un inserimento non vengono specificati i valori di tutti gli attributi della tabella, agli attributi mancanti viene assegnato il valore di default, o in assenza di questo il valore nullo. Si noti infine che la corrispondenza tra gli attributi della tabella e i valori da inserire è data dall'ordine in cui compaiono i termini della definizione della tabella.

Il **comando delete** elimina righe dalle tabelle della base di dati, seguendo la semplice sintassi:

```
delete from NomeTabella [ where Condizione ]
```

Quando la condizione argomento della clausola *where* non viene specificata, il comando cancella tutte le righe della tabella, altrimenti vengono rimosse solo le righe che soddisfano la condizione.

La condizione rispetta la sintassi della *select*, per cui possono comparire al suo interno anche interrogazioni nidificate che fanno riferimento ad altre tabelle. Un semplice esempio è il comando che elimina i dipartimenti senza impiegati:

```
delete from Dipartimento
    where Nome not in (select Dpart
        from Impiegato)
```

Il comando *drop* ha lo stesso effetto del comando *delete*, ma in più anche lo schema della base di dati viene modificato, eliminando dallo schema non solo la tabella riferita, ma anche tutte le viste e tabelle che nella loro definizione fanno riferimento ad essa. Invece, con il comando *delete* lo schema della base di dati rimane immutato e viene modificata solo l'istanza della base di dati.

Il **comando di update** presenta una sintassi leggermente più complicata:

```
update NomeTabella
    set Attributo = { Espressione | SelectSQL | null | default }
        {, Attributo = { Espressione | SelectSQL | null | default } }
    [ where Condizione ]
```

Questo comando permette di aggiornare uno o più attributi delle righe di *NomeTabella* che soddisfano l'eventuale *Condizione*. Se il comando non presenta la clausola *where*, come al

solito si suppone che la condizione sia soddisfatta e si esegue la modifica su tutte le righe.
Il nuovo valore cui viene posto l'attributo può essere:

1. il risultato della valutazione di un'espressione sugli attributi della tabella, che può anche far riferimento al valore corrente dell'attributo che verrà modificato dal comando;
2. il risultato di una generica interrogazione SQL;
3. il valore nullo;
4. il valore di default per il dominio.

Introduzione alla progettazione

La progettazione di una base di dati costituisce solo una delle componenti del processo di sviluppo di un sistema informativo complesso, e va quindi inquadrata in un contesto più ampio, quello del **ciclo di vita** dei sistemi informativi.

Il ciclo di vita di un sistema informativo comprende, generalmente, le seguenti attività:

- *studio di fattibilità*: serve a definire, in maniera per quanto possibile precisa, i costi delle varie alternative possibili e a stabilire le priorità di realizzazione delle varie componenti del sistema;
- *raccolta e analisi dei requisiti*: consiste nell'individuazione e nello studio delle proprietà e delle funzionalità che il sistema informativo dovrà avere. Questa fase produce una descrizione completa, ma generalmente informale, dei dati coinvolti e delle operazioni su di essi. Vengono inoltre stabiliti i requisiti software e hardware del sistema informativo.
- *progettazione*: si divide generalmente in progettazione dei dati e progettazione delle applicazioni. Nella prima si individua la struttura e l'organizzazione che i dati dovranno avere, nell'altra si definiscono le caratteristiche dei programmi applicativi.
- *implementazione*: consiste nella realizzazione del sistema informativo secondo la struttura e le caratteristiche definite nella fase di progettazione. Viene costruita e popolata la base di dati e viene prodotto il codice dei programmi.
- *validazione e collaudo*: serve a verificare il corretto funzionamento e la qualità del sistema informativo.
- *funzionamento*: in questa fase il sistema informativo diventa operativo ed esegue i compiti per i quali era stato originariamente progettato.

Va precisato che il processo non è quasi mai strettamente sequenziale, in quanto spesso, durante l'esecuzione di una delle attività citate, bisogna rivedere decisioni prese nell'attività precedente.

Metodologie di progettazione

Una **metodologia di progettazione** consiste in:

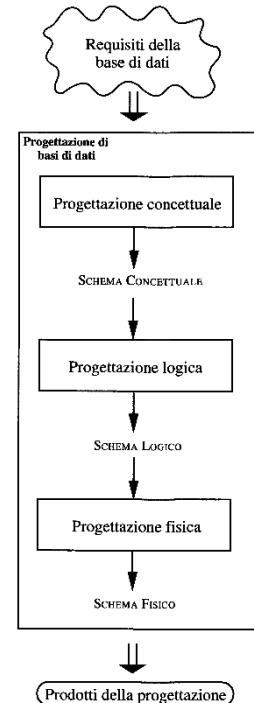
- una *decomposizione* dell'intera attività di progetto in passi successivi indipendenti tra loro;
- una serie di *strategie* da seguire nei vari passi e alcuni *criteri* per la scelta in caso di alternative;
- alcuni *modelli di riferimento* per descrivere i dati di ingresso e uscita delle varie fasi.

Le proprietà che una metodologia deve garantire sono principalmente:

- la *generalità* rispetto alle applicazioni e ai sistemi in gioco;
- la *qualità del prodotto* in termini di correttezza, completezza ed efficienza rispetto alle risorse impiegate;
- la *facilità d'uso* delle strategie e dei modelli di riferimento.

Negli anni si è consolidata una metodologia di progetto articolata in tre fasi principali da effettuare in cascata: vengono separate in maniera netta le decisioni relative a “cosa” rappresentare in una base di dati (prima fase), da quelle relative a “come” farlo (seconda e terza fase).

- **Progettazione concettuale:** ha come scopo quello di rappresentare le specifiche informali della realtà di interesse in termini di una descrizione formale e completa, ma indipendente dai criteri di rappresentazione utilizzati nei sistemi di gestione di basi di dati. Il prodotto in questa fase viene chiamato *schema concettuale* e fa riferimento a un *modello concettuale* dei dati.
- **Progettazione logica:** consiste nella traduzione dello schema concettuale definito nella fase precedente, in termini del modello di rappresentazione dei dati adottato dal sistema di gestione di basi di dati a disposizione. Il prodotto di questa fase viene chiamato *schema logico* della base di dati e fa riferimento a un *modello logico* dei dati.
- **Progettazione fisica:** in questa fase lo schema logico viene completato con la specifica dei parametri fisici di memorizzazione dei dati (organizzazione dei file e degli indici). Il prodotto di questa fase viene denominato *schema fisico* e fa riferimento a un *modello fisico* dei dati.



Quindi, a partire da requisiti rappresentati da documenti e moduli di vario genere, acquisiti anche attraverso l'interazione con gli utenti, viene costruito uno schema Entità-Relazione (rappresentato da un diagramma) che descrive a livello concettuale la base di dati. Questa rappresentazione viene poi tradotta in uno schema relazionale, costituito da una collezione di tabelle. Infine, i dati vengono descritti da un punto di vista fisico (tipo e dimensioni dei campi) e vengono specificate strutture ausiliarie, come gli indici, per l'accesso efficiente ai dati.

Il modello Entità-Relazione

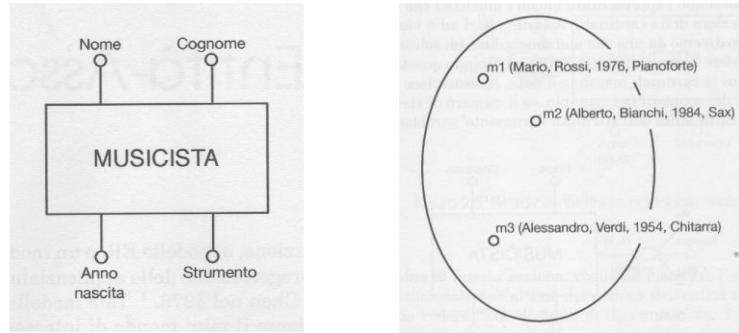
Come già sappiamo, il modello dei dati è un'astrazione dei dati che permette di definire le proprietà degli oggetti di interesse per la base di dati e le relazioni fra questi oggetti, dando la possibilità di nascondere i dettagli dell'implementazione. Il **modello E-R** è un modello *concettuale* dei dati e, come tale, fornisce una serie di strutture (dette *costrutti*) che vengono utilizzate per definire *schemi* che descrivono l'organizzazione e la struttura delle *occorrenze* dei dati.

Entità

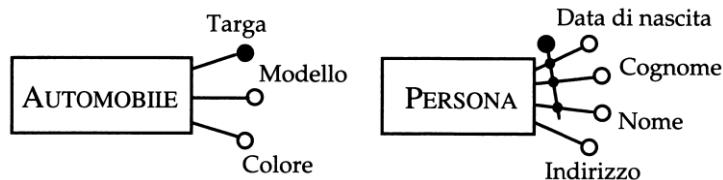
Le **entità** rappresentano classi di oggetti (per esempio fatti, cose, persone) della realtà di interesse che hanno proprietà comuni ed esistenza “autonoma”. Una occorrenza di entità ha un'esistenza (e un'identità) indipendente dalle proprietà a esso associate (un impiegato esiste indipendentemente dal fatto di avere un nome, un cognome, un'età, ecc.).

In uno schema, ogni entità ha un nome che la identifica univocamente e viene rappresentata graficamente mediante un rettangolo con il nome dell'entità all'interno.

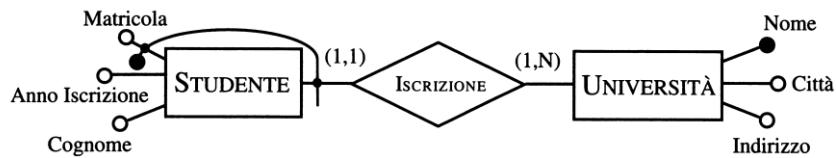
Possiamo distinguere una *rappresentazione intensionale* (che ci dice solo come saranno rappresentati i concetti del mondo che si sta cercando di modellare) e una *rappresentazione estensionale* (che dichiara quali sono i valori dei vari attributi).



Gli **identificatori delle entità** vengono specificati per ciascuna entità di uno schema e descrivono i concetti (attributi e/o entità) dello schema che permettono di identificare in maniera univoca le occorrenze delle entità. In molti casi, uno o più attributi di una entità sono sufficienti a individuare un identificatore: si parla in questo caso di *identificatore interno* (detto anche *chiave*).



Alcune volte però gli attributi di un'entità non sono sufficienti a identificare univocamente le sue occorrenze. Un'entità E può essere identificata da altre entità solo se tali entità sono coinvolte in una relazione a cui E partecipa con cardinalità $(1,1)$. Nei casi in cui l'identificazione di un'entità è ottenuta utilizzando altre entità si parla di *identificatore esterno*.



Sulla base di quanto detto sulle identificazioni, è possibile fare alcune considerazioni generali:

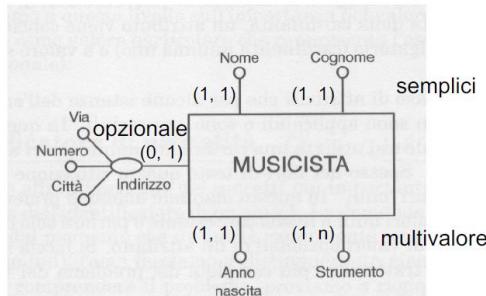
- un identificatore può coinvolgere uno o più attributi, ognuno dei quali deve avere cardinalità $(1,1)$;
- un'identificazione esterna può coinvolgere una o più entità, ognuna delle quali deve essere membro di una relazione alla quale l'entità da identificare partecipa con cardinalità $(1,1)$;

- un'identificazione esterna può coinvolgere un'entità che è a sua volta identificata esternamente, purché non vengano generati, in questa maniera, cicli di identificazioni esterne;
- ogni entità deve avere almeno un identificatore (interno o esterno), ma ne può avere in generale più di uno; nel caso di più identifieri, gli attributi e le entità coinvolte in alcune identificazioni (tranne una) possono essere opzionali (cardinalità minima uguale a zero).

Attributi

Un **attributo** è una proprietà elementare di un'entità o di una associazione di interesse ai fini dell'applicazione. Associa a ciascuna occorrenza di entità (o di relazione) un valore appartenente a un insieme, detto *dominio*, che contiene i valori ammissibili per l'attributo.

Può risultare comodo raggruppare attributi di una medesima entità o relazione che presentano affinità nel loro significato o uso: l'insieme di attributi che si ottiene in questa maniera viene detto **attributo composto**.

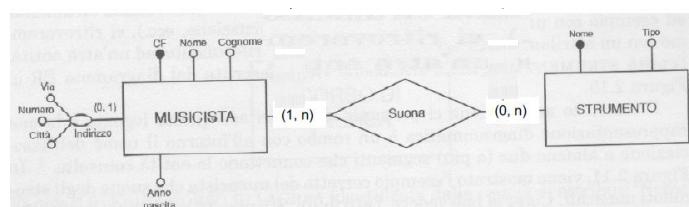


Le **cardinalità degli attributi** descrivono il numero minimo e massimo di valori dell'attributo associati a ogni occorrenza di entità o relazione. Sono quindi una coppia di valori (n, m) , i cui possibili valori possono essere 0 (attributo *opzionale*), 1 (attributo *semplice*, deve necessariamente esserci) oppure N (attributo *multivalue*). Nella maggior parte dei casi, la cardinalità di un attributo è pari a $(1, 1)$ e viene omessa: in questi casi l'attributo rappresenta sostanzialmente una funzione che associa a ogni occorrenza di entità un solo valore dell'attributo.

Relazioni (o associazioni)

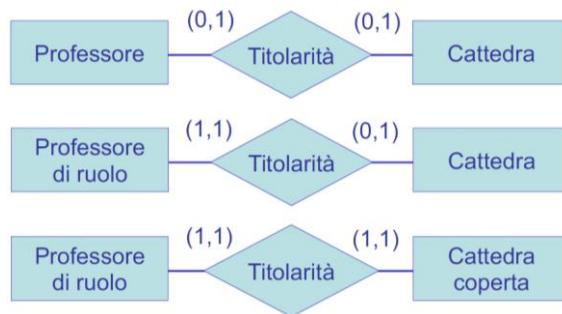
Le **relazioni** rappresentano legami logici, significativi per l'applicazione di interesse, tra due o più entità. Una occorrenza di relazione è un'ennupla (una coppia nel caso più frequente di relazione binaria) costituita da occorrenze di entità, una per ciascuna delle entità coinvolte.

In uno schema E-R, ogni relazione ha un nome che la identifica univocamente e viene rappresentata graficamente mediante un rombo, con il nome della relazione all'interno, e da linee che connettono la relazione con ciascuna delle sue componenti.

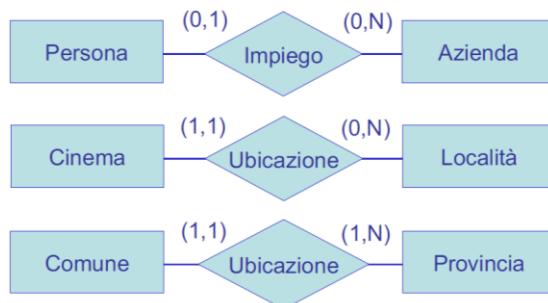


Le **cardinalità delle relazioni** (anche dette *vincoli di partecipazione*) dicono quante volte, in una relazione tra entità, un'occorrenza di una di queste entità può essere legata a occorrenze delle altre entità coinvolte. Sono quindi una coppia di valori (n, m) , i cui possibili valori possono essere 0 (la partecipazione dell'entità relativa è *opzionale*), 1 (la partecipazione è *obbligatoria*) oppure N (c'è una associazione con un numero arbitrario di occorrenze dell'altra entità).

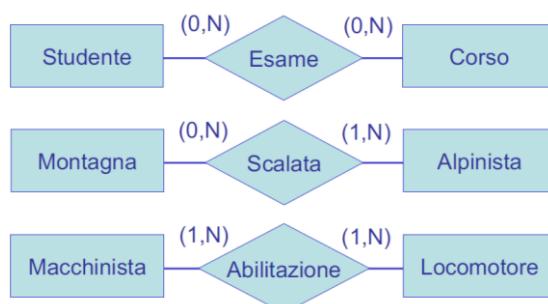
Osservando le cardinalità massime, è possibile classificare le relazioni binarie in base al tipo di corrispondenza che viene stabilita tra le occorrenze delle entità coinvolte. Le relazioni aventi cardinalità massima pari a uno per entrambe le entità coinvolte definiscono una corrispondenza uno-a-uno tra le occorrenze di tali entità e vengono quindi denominate **relazioni uno-a-uno**.



In maniera analoga, le relazioni aventi un'entità con cardinalità massima pari a uno e l'altra con cardinalità massima pari a N sono denominate **relazioni uno-a-molti**.



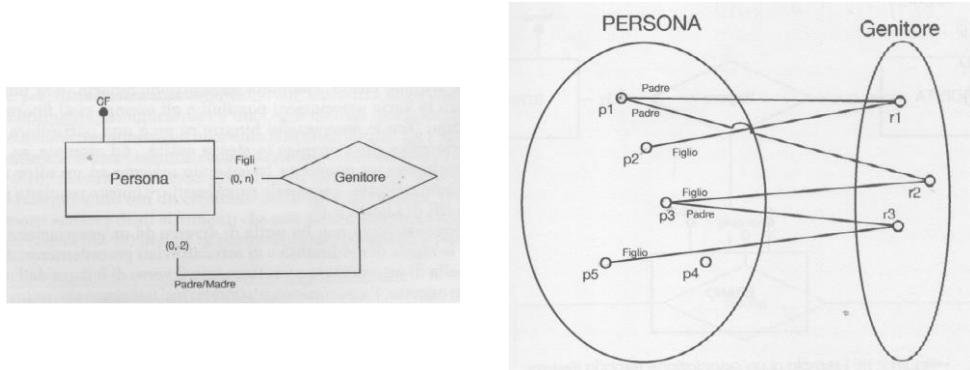
Infine, le relazioni aventi cardinalità massima pari a N per entrambe le entità coinvolte vengono denominate **relazioni molti a molti**.



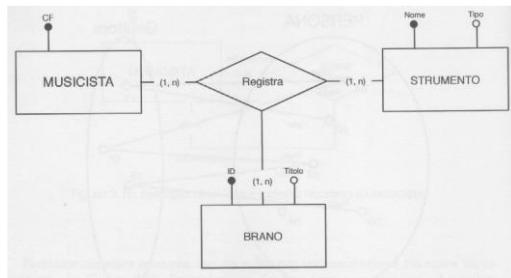
Associazioni ricorsive e n-arie

È anche possibile avere **relazioni ricorsive**, ovvero relazioni tra una entità e sé stessa. Se la relazione non è simmetrica, è necessario stabilire i due *ruoli* che l'entità coinvolta gioca nella relazione. Questo può essere fatto associando degli identificatori alle linee uscenti dalla relazione ricorsiva.

Di seguito, l'estensione di un esempio di associazione ricorsiva non simmetrica.



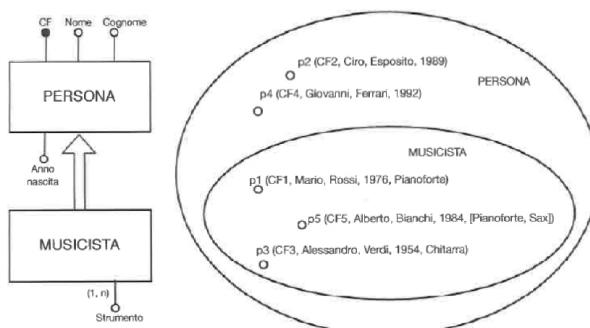
Infine, è possibile avere **relazioni n-arie**, ovvero relazioni che coinvolgono più di due entità.



Nel caso in cui un'entità partecipi a una relazione n-aria con cardinalità massima pari a uno, significa che ogni sua occorrenza può essere legata a una sola occorrenza della relazione, e quindi a un'unica ennupla di occorrenze delle altre entità coinvolte nella relazione. Questo significa che è possibile eliminare la relazione n-aria e legare direttamente tale entità con le altre entità, mediante delle relazioni binarie di tipo uno a molti.

Generalizzazioni

Le **generalizzazioni** rappresentano legami logici tra un'entità E , detta **entità genitore** (o super-insieme), e una o più entità E_1, \dots, E_n , dette **entità figlie** (o sotto-insieme), di cui E è più generale, nel senso che le comprende come caso particolare. Si dice in questo caso che E è *generalizzazione* di E_1, \dots, E_n e che le entità E_1, \dots, E_n sono *specializzazioni* dell'entità E .



Tra le entità coinvolte in una generalizzazione valgono le seguenti proprietà generali:

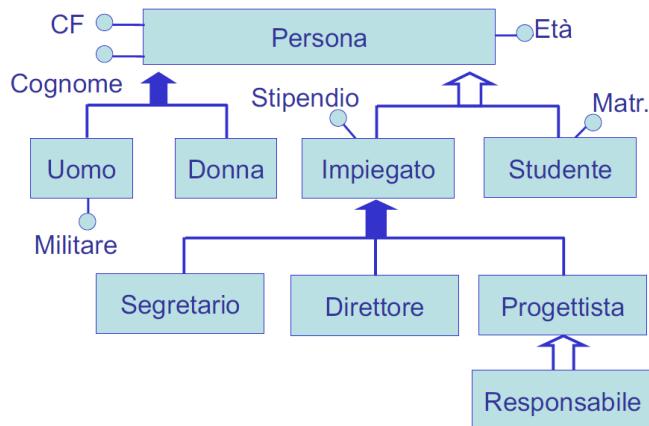
- ogni occorrenza di un'entità figlia è anche un'occorrenza dell'entità genitore;
- ogni proprietà dell'entità genitore (attributi, identificatori, relazioni e altre generalizzazioni) è anche una proprietà delle entità figlie.

Le generalizzazioni vengono rappresentate graficamente mediante delle frecce che congiungono le entità figlie con l'entità genitore. Si osservi che, per le entità figlie, le proprietà ereditate non vanno rappresentate esplicitamente.

Possono essere classificate sulla base di due proprietà tra loro ortogonali:

- una generalizzazione è **totale** (freccia nera) se ogni occorrenza dell'entità genitore è una occorrenza di almeno una delle entità figlie, altrimenti è **parziale** (freccia bianca).
- una generalizzazione è **esclusiva** se ogni occorrenza dell'entità genitore è al più un'occorrenza di una delle entità figlie (gli insiemi figli sono disgiunti), altrimenti è **sovrapposta** (gli insiemi figli non sono disgiunti).

Di seguito, un esempio riassuntivo (che è anche una *gerarchia* di generalizzazioni):



Documentazione di schemi E-R

Abbiamo visto come il modello E-R fornisca strumenti di modellazione molto espressivi che ci permettono di descrivere, con efficacia e facilità, situazioni anche molto complesse. Tuttavia, uno schema E-R non è quasi mai sufficiente, da solo, a rappresentare nel dettaglio tutti gli aspetti di un'applicazione, per varie ragioni.

Innanzitutto, in uno schema E-R compaiono solo i nomi dei vari concetti in esso presenti, ma questo può essere insufficiente per comprenderne il significato. Nel caso di schemi particolarmente complessi può accadere inoltre di non riuscire a rappresentare in maniera comprensibile ed esaustiva i vari concetti. Risulta infine, in certi casi, addirittura impossibile rappresentare alcune proprietà dei dati attraverso i costrutti che il modello E-R mette a disposizione.

In conclusione, risulta indispensabile corredare ogni schema E-R con una **documentazione di supporto**, che possa servire a facilitare l'interpretazione dello schema stesso e a descrivere proprietà dei dati rappresentati che non possono essere espressi direttamente dai costrutti del modello.

Regole aziendali

Una **regola aziendale** può essere:

1. la *descrizione di un concetto* rilevante per l'applicazione, ovvero la definizione precisa di un'entità, di un'attributo o di una relazione del modello E-R;
2. un *vincolo di integrità* sui dati dell'applicazione, sia esso la documentazione di un vincolo espresso con qualche costrutto del modello E-R (per esempio le cardinalità di una relazione) o la descrizione di un vincolo non esprimibile direttamente con i costrutti del modello;
3. una *derivazione*, ovvero un concetto che può essere ottenuto, attraverso un'inferenza o un calcolo aritmetico, da altri concetti dello schema.

Per le regole del primo tipo è chiaramente impossibile definire una sintassi precisa e si fa in genere ricorso a frasi in linguaggio naturale. Queste regole vengono tipicamente rappresentate sotto forma di **glossari**, raggruppando le descrizioni in maniera opportuna (per esempio, per entità e per relazione).

Le regole che descrivono vincoli di integrità possono essere espresse sotto forma di *asserzioni*, ovvero affermazioni che devono sempre essere verificate nella nostra base di dati. Una struttura predefinita per enunciare regole aziendali sotto forma di asserzioni è la seguente: <conceitto> deve/non deve <espressione sui concetti>, dove i concetti citati possono corrispondere o a concetti che compaiono nello schema E-R a cui si fa riferimento, oppure a concetti derivabili da essi.

Le regole aziendali che esprimono derivazioni possono essere espresse specificando le operazioni che permettono di ottenere il concetto derivato. Una possibile struttura è quindi: <conceitto> si ottiene <operazioni su concetti>.

Tecniche di documentazione

La documentazione dei vari concetti rappresentati in uno schema, ovvero le regole aziendali di tipo descrittivo, può essere prodotta facendo uso di un **dizionario dei dati**. Esso è composto da due tavole: *la prima descrive le entità* dello schema con il nome, una definizione informale in linguaggio naturale, l'elenco di tutti gli attributi (con eventuali descrizioni associate) e i possibili identificatori. *L'altra tabella descrive le relazioni* con il nome, una loro descrizione informale, l'elenco degli attributi (con eventuali descrizioni) e l'elenco delle entità coinvolte insieme alla loro cardinalità di partecipazione.

Entità	Descrizione	Attributi	Identificatore
Impiegato	Dipendente dell'azienda	Codice, Cognome, Stipendio	Codice
Progetto	Progetti aziendali	Nome, Budget	Nome
Dipartimento	Struttura aziendale	Nome, Telefono	Nome, Sede
Sede	Sede dell'azienda	Città, Indirizzo	Città

Associazione	Descrizione	Componenti	Attributi
Direzione	Direzione di un dipartimento	Impiegato, Dipartimento	
Afferenza	Afferenza a un dipartimento	Impiegato, Dipartimento	Data
Partecipazione	Partecipazione a un progetto	Impiegato, Progetto	
Composizione	Composizione dell'azienda	Dipartimento, Sede	

Per quel che riguarda le altre regole aziendali, si può far ricorso ancora a una tabella, nella quale vengono elencate le varie regole, specificando di volta in volta la loro tipologia.

Vincoli di integrità sui dati
(1) Il direttore di un dipartimento deve afferire a tale dipartimento
(2) Un impiegato non deve avere uno stipendio maggiore del direttore del dipartimento al quale afferisce
(3) Un dipartimento con sede a Roma deve essere diretto da un impiegato con più di dieci anni di anzianità
(4) Un impiegato che non afferisce a nessun dipartimento non deve partecipare a nessun progetto

Progettazione concettuale

La progettazione concettuale di una base di dati consiste nella costruzione di uno schema E-R in grado di descrivere al meglio le specifiche sui dati di un'applicazione. L'attività che precede la progettazione vera e propria, ossia la raccolta e l'analisi dei requisiti, non è completamente separata da quella della progettazione, ma procede (in molti casi) parallelamente a essa. Possiamo infatti iniziare a costruire uno schema E-R quando non abbiamo ancora terminato di raccogliere e analizzare tutti i requisiti, per poi arricchirlo progressivamente man mano che le informazioni in nostro possesso aumentano.

Raccolta e analisi dei requisiti

Per **raccolta dei requisiti** si intende la completa individuazione dei problemi che l'applicazione da realizzare deve risolvere e le caratteristiche che tale applicazione dovrà avere. Per caratteristiche del sistema si intendono sia gli aspetti statici (i dati) sia gli aspetti dinamici (le operazioni sui dati).

L'**analisi dei requisiti** consiste nel chiarimento e nell'organizzazione delle specifiche dei requisiti. Si tratta ovviamente di attività fortemente interconnesse: l'attività di analisi inizia con i primi requisiti ottenuti per poi procedere di pari passo con l'attività di raccolta. In molti casi è l'attività stessa di analisi dei requisiti che suggerisce successive attività di raccolta.

I requisiti di un'applicazione provengono, nella maggior parte dei casi, da fonti diverse. Le principali fonti di informazione sono, in genere, le seguenti.

- Gli *utenti dell'applicazione*. In questo caso le informazioni si acquisiscono mediante opportune interviste, anche ripetute, oppure attraverso una documentazione scritta che gli utenti possono aver predisposto appositamente per questo scopo.
- Tutta la *documentazione esistente* che ha qualche attinenza con il problema allo studio: moduli, regolamenti interni, procedure aziendali, normative. È richiesta, in questo caso, un'attività di raccolta e selezione che viene assistita dagli utenti, ma è a carico del progettista.
- Eventuali *realizzazioni preesistenti*, ovvero applicazioni che si devono rimpiazzare o che devono interagire in qualche maniera con il sistema da realizzare.

Progettazione concettuale

Durante l'interazione con gli utenti del sistema informativo può avvenire che utenti diversi forniscano informazioni diverse, spesso complementari ma qualche volta contraddittorie. In genere gli utenti a livello più alto possiedono una visione più ampia, ma meno dettagliata. Possono però indirizzare verso gli esperti dei singoli sottoproblemi.

Come criterio generale da seguire possiamo dire che, nel corso delle interviste, è opportuno effettuare con l'utente verifiche di comprensione e consistenza sulle informazioni che si stanno raccogliendo. Questo può essere fatto attraverso esempi (generali e relativi a casi limite) oppure richiedendo definizioni e classificazioni precise. È inoltre molto importante in questa fase cercare di individuare gli aspetti essenziali rispetto a quelli marginali e procedere per raffinamenti successivi.

È molto importante effettuare una profonda analisi del testo che descrive le specifiche per filtrare le eventuali inesattezze e i termini ambigui presenti. Proviamo a fissare alcune regole generali per ottenere una specifica dei requisiti più precisa e senza ambiguità (facendo anche riferimento a un semplice esempio).

Società di formazione	
1	<i>Si vuole realizzare una base di dati per una società che eroga corsi, di cui vogliamo rappresentare i dati dei partecipanti ai corsi e dei docenti.</i>
2	<i>Per i partecipanti (circa 5000), identificati da un codice, si vuole memorizzare il codice fiscale, il cognome, l'età, il sesso, il luogo di nascita,</i>
3	<i>il nome dei loro attuali datori di lavoro, i posti dove hanno lavorato in precedenza insieme al periodo, l'indirizzo e il numero di telefono, i corsi che hanno frequentato (i corsi sono in tutto circa 200) e il giudizio finale.</i>
4	<i>Rappresentiamo anche i seminari che stanno attualmente frequentando e, per ogni giorno, i luoghi e le ore dove sono tenute le lezioni. I corsi</i>
5	<i>hanno un codice, un titolo e possono avere varie edizioni con date di</i>
6	<i>inizio e fine e numero di partecipanti. Se gli studenti sono liberi professionisti, vogliamo conoscere l'area di interesse e, se lo possiedono, il titolo. Per quelli che lavorano alle dipendenze di altri, vogliamo conoscere invece il loro livello e la posizione ricoperta. Per gli insegnanti</i>
7	<i>(circa 300), rappresentiamo il cognome, l'età, il posto dove sono nati, il</i>
8	<i>nome del corso che insegnano, quelli che hanno insegnato nel passato e</i>
9	<i>quelli che possono insegnare. Rappresentiamo anche tutti i loro recapiti telefonici. I docenti possono essere dipendenti interni della società o</i>
10	<i>collaboratori esterni.</i>
11	
12	
13	
14	
15	
16	
17	
18	
19	

- **Scegliere il corretto livello di astrazione.** È bene evitare di utilizzare termini troppo generici o troppo specifici che rendono poco chiaro un concetto. Per esempio, nel nostro caso sono stati utilizzati i termini *titolo* (a riga 13), con riferimento ai partecipanti che sono liberi professionisti (che tra l'altro è utilizzato anche per indicare un concetto diverso a riga 10) e *giudizio* (riga 7), con riferimento alla valutazione dei corsi, che andrebbero specificati meglio (per esempio, come *titolo professionale e votazione in decimi*).
- **Standardizzare la struttura delle frasi.** Nella specifica dei requisiti è preferibile utilizzare sempre lo stesso stile sintattico. Per esempio, “*per <dato> rappresentiamo <insieme di proprietà>*”.
- **Evitare frasi contorte.** Le definizioni devono essere semplici e chiare. Per esempio, *lavoratori dipendenti* (o più semplicemente “dipendenti”) è da preferire a *quelli che lavorano alle dipendenze di altri* (riga 13).
- **Individuare sinonimi/omonimi e unificare i termini.** I *sinonimi* indicano termini con lo stesso significato (per esempio, *docente* a riga 2 e *insegnante* a riga 14, oppure *partecipante* a riga 2 e *studente* a riga 11); gli *omonimi* indicano termini uguali con diversi significati (per esempio *posto*, riferito a impiego a riga 5 e a città a riga 15, e *luogo*, riferito a città a riga 4 e ad aula a riga 9). Queste situazioni possono generare ambiguità e vanno chiarite: nel caso di sinonimi unificando i termini, nel caso di omonimi utilizzando termini diversi o specificandoli meglio.
- **Rendere esplicito il riferimento tra termini.** Può succedere che l'assenza di contesto di riferimento renda alcuni concetti ambigui: in questi casi bisogna esplicitare il riferimento tra termini. Per esempio, nella riga 6, non è chiaro se i termini *indirizzo* e *numero di telefono* sono relativi ai partecipanti o ai loro datori di lavoro; inoltre, a riga 13, nella frase *Per quelli che lavorano ...*, si deve chiarire

esplicitamente a chi ci stiamo riferendo (partecipanti, docenti?) per evitare confusione.

- **Costruire un glossario dei termini.** È molto utile, per la comprensione e la precisazione dei termini usati, definire un glossario che, per ogni termine, contenga: una breve descrizione, possibili sinonimi e altri termini contenuti nel glossario con i quali esiste un legame logico. Un breve glossario per il nostro esempio è il seguente:

Termino	Descrizione	Sinonimi	Collegamenti
Partecipante	Partecipante ai corsi. Può essere un dipendente o un professionista.	Studente	Corso, Datore
Docente	Docente dei corsi. Possono essere collaboratori esterni.	Insegnante	Corso
Corso	Corsi offerti. Possono avere varie edizioni.	Seminario	Docente, Partecipante
Datore	Datori di lavoro attuali e passati dei partecipanti ai corsi.	Posto	Partecipante

Dopo aver individuato le varie ambiguità e le imprecisioni, esse vanno eliminate sostituendo i termini non corretti con termini più adeguati. In caso di dubbio, è necessario intervistare nuovamente colui che ha fornito il dato o consultare la documentazione relativa.

Vediamo quali sono le principali modifiche da apportare al nostro testo.

- Come già detto, *luogo* di nascita dei partecipanti (riga 4) è un omonimo del luogo in cui si tengono le lezioni e va sostituito da *città* di nascita, così come *posto* (riga 5) che va sostituito con *datore di lavoro*.
- Va chiarito che a riga 6 l'*indirizzo* e il *numero di telefono* fanno riferimento ai datori di lavoro dei partecipanti.
- Il *giudizio* (riga 7) deve essere interpretato come *votazione in decimi*, mentre *periodo* (riga 6) va interpretato come *date di inizio e fine rapporto*.
- Bisogna specificare che i partecipanti frequentano o hanno frequentato specifiche *edizioni* di corsi. Per quanto riguarda gli altri termini che fanno riferimento ai corsi: *seminari* (riga 8) è un sinonimo e va sostituito da *edizione di corso*, *giorno* (riga 9), riferito alle lezioni, è troppo astratto, e va utilizzato *giorno della settimana* mentre *luogo* (riga 9) è un omonimo, che va sostituito da *aula*.
- Il termine *studente* (riga 11) va sostituito con *partecipante*. Per *titolo* (riga 13) di un partecipante che è libero professionista si intende il suo *titolo professionale*.
- Per quello che riguarda i docenti abbiamo che *insegnante* (riga 14) è sinonimo di *docente*, *posto* (riga 15) indica la *città* di nascita, il *nome* del corso che insegnano (riga 16) è un sinonimo di *titolo* del corso e il *recapito telefonico* (righe 17-18) è sinonimo di *numero di telefono*.

A questo punto possiamo riscrivere le nostre specifiche apportando le modifiche proposte. È molto utile, in questa fase, decomporre il testo in gruppi di frasi omogenee, relative cioè agli stessi concetti. Otteniamo così la strutturazione delle specifiche sui dati riportata di seguito:

Frasi di carattere generale
<i>Si vuole realizzare una base di dati per una società che eroga corsi, di cui vogliamo rappresentare i dati dei partecipanti ai corsi e dei docenti.</i>

Frasi relative ai partecipanti
<i>Per i partecipanti (circa 5000), identificati da un codice, rappresentiamo il codice fiscale, il cognome, l'età, il sesso, la città di nascita, i nomi dei loro attuali datori di lavoro e di quelli precedenti (insieme alle date di inizio e fine rapporto), le edizioni dei corsi che stanno attualmente frequentando e quelli che hanno frequentato in passato, con la relativa votazione finale in decimi.</i>
Frasi relative ai datori di lavoro
<i>Relativamente ai datori di lavoro presenti e passati dei partecipanti, rappresentiamo il nome, l'indirizzo e il numero di telefono.</i>
Frasi relative ai corsi
<i>Per i corsi (circa 200), rappresentiamo il titolo e il codice, le varie edizioni con date di inizio e fine e, per ogni edizione, rappresentiamo il numero di partecipanti e il giorno della settimana, le aule e le ore dove si sono tenute le lezioni.</i>
Frasi relative a tipi specifici di partecipanti
<i>Per i partecipanti che sono liberi professionisti, rappresentiamo l'area di interesse e, se lo possiedono, il titolo professionale. Per i partecipanti che sono dipendenti, rappresentiamo invece il loro livello e la posizione ricoperta.</i>
Frasi relative ai docenti
<i>Per i docenti (circa 300), rappresentiamo il cognome, l'età, la città di nascita, tutti i numeri di telefono, il titolo del corso che insegnano, di quelli che hanno insegnato in passato e di quelli che possono insegnare. I docenti possono essere dipendenti interni della società di formazione o collaboratori esterni.</i>

Naturalmente, accanto alle specifiche sui dati, vanno raccolte le specifiche sulle operazioni da effettuare su questi dati. Bisogna cercare di utilizzare la medesima terminologia usata per i dati (possiamo per questo far riferimento al glossario dei termini) e informarci anche sulla frequenza con la quale le varie operazioni vengono eseguite.

- Operazione 1:** inserisci un nuovo partecipante indicando tutti i suoi dati (operazione da effettuare in media 40 volte al giorno).
- Operazione 2:** assegna un partecipante a una edizione di corso (circa 50 volte al giorno).
- Operazione 3:** inserisci un nuovo docente indicando tutti i suoi dati e i corsi che può insegnare (2 volte al giorno).
- Operazione 4:** assegna un docente abilitato a una edizione di un corso (15 volte al giorno);
- Operazione 5:** stampa tutte le informazioni sulle edizioni passate di un corso con titolo, orari lezioni e numero partecipanti (10 volte al giorno);
- Operazione 6:** stampa tutti i corsi offerti, con informazioni sui docenti che possono insegnarli (20 volte al giorno);
- Operazione 7:** per ogni docente, trova i partecipanti a tutti i corsi da lui/lei insegnati (5 volte a settimana);
- Operazione 8:** effettua una statistica su tutti i partecipanti a un corso con tutte le informazioni su di essi, sull'edizione alla quale hanno partecipato e sulla rispettiva votazione (10 volte al mese).

Dopo questa strutturazione dei requisiti, siamo pronti ad avviare la prima fase della progettazione, che consiste nella costruzione di uno schema concettuale in grado di descrivere in maniera adeguata tutte le specifiche dei dati raccolte.

Rappresentazione concettuale dei dati

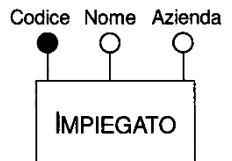
Non esiste una rappresentazione univoca di un insieme di specifiche, perché le stesse informazioni possono essere rappresentate in modi differenti e non comparabili. Nel caso della progettazione concettuale conviene seguire le “regole concettuali” del modello E-R.

- Se un concetto ha proprietà significative e/o descrive classi di oggetti con esistenza autonoma, è opportuno rappresentarlo con un'**entità** (come i concetti di *docente* e *corso*).
- Se un concetto ha una struttura semplice e non possiede proprietà rilevanti associate, è opportuno rappresentarlo con un **attributo** di un altro concetto a cui si riferisce (come i concetti di *età* e *città*, in quanto, oltre al nome, non è di interesse nessuna altra sua proprietà).
- Se sono state individuate due (o più) entità e nei requisiti compare un concetto che le associa, questo concetto può essere rappresentato da una **relazione** (come il concetto di *partecipazione a un corso*, rappresentabile da una relazione tra le entità che rappresentano i *partecipanti* e i *corsi*). È importante sottolineare il fatto che questo vale solo nel caso in cui il concetto in questione non abbia, esso stesso, le caratteristiche delle entità.
- Se uno o più concetti risultano essere casi particolari di un altro, è opportuno rappresentarli facendo uso di una **generalizzazione** (come i concetti di *professionista* e *dipendente* che costituiscono dei casi particolari del concetto di *partecipante*).

Pattern di progetto

Con il termine **design pattern** si intendono soluzioni progettuali a problemi comuni della progettazione concettuale dei dati.

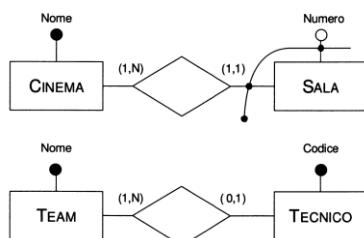
Cominciamo da un caso semplice: quello in cui si individua nelle specifiche un concetto autonomo con proprietà associate. Nel caso, per esempio, di un impiegato di cui sono di interesse un codice, il nome e l'azienda nel quale lavora, otteniamo il primo, semplice schema con una sola entità riportato qui a fianco.



È importante comprendere che, con questa soluzione, non stiamo rappresentando anche il concetto di azienda: qui l'azienda è solo un attributo, ovvero niente di più che una stringa che assegniamo a un'occorrenza di impiegato. Per poter rappresentare esplicitamente il concetto di azienda dobbiamo **reificare l'attributo**, facendolo diventare un'entità. Otteniamo così lo schema sottostante.

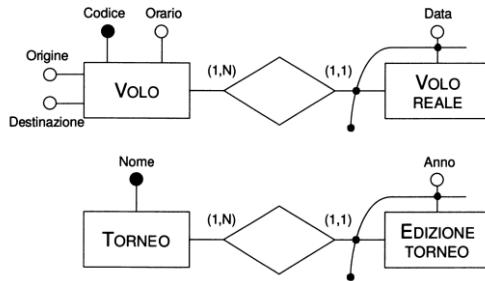


Passiamo ora a dei semplici pattern che coinvolgono le relazioni. Un caso piuttosto frequente di uso di questo costrutto è quello in cui si vuole rappresentare il fatto che un'entità è **parte di** (*part-of*) un'altra entità, come avviene negli schemi seguenti:



Queste relazioni sono tipicamente uno a molti e si presentano in due forme. Nel primo caso, l'esistenza di un'occorrenza dell'entità "parte" dipende dall'esistenza di un'occorrenza dell'entità che la contiene e richiede un'identificazione esterna. Nel secondo, l'entità contenuta nell'altra ha esistenza autonoma, come indicato dalla partecipazione opzionale alla relazione.

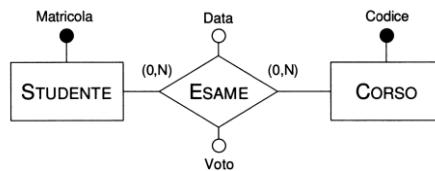
Un'altra situazione piuttosto comune è illustrata negli esempi della figura sottostante, nei quali le occorrenze di un'entità della relazione sono **istanze di** (*instance-of*) occorrenze dell'altra entità.



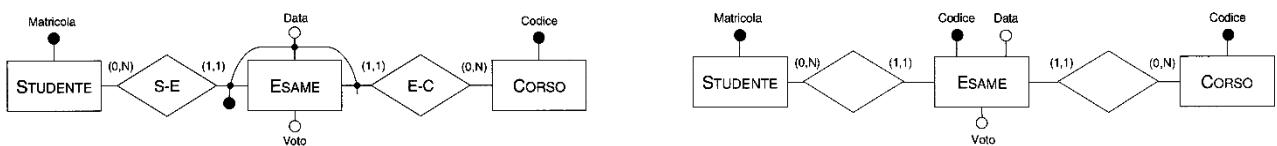
Nel primo caso abbiamo un'entità che descrive il concetto astratto di volo presente sull'orario di una compagnia aerea, con un codice, un'origine, una destinazione e un orario, e un'altra entità che rappresenta il volo "reale", vale a dire l'istanza di un certo volo in un certo giorno (e.g. il volo AZ610 del 15/12/2013). L'identificazione del volo reale avviene attraverso la data e, esternamente, il volo di cui è istanza (si assume quindi che lo stesso volo non possa essere ripetuto lo stesso giorno).

Un caso analogo è l'altro schema, con il quale viene rappresentato il concetto di torneo sportivo (per esempio gli internazionali italiani di tennis) e una sua edizione (per esempio quella del 2014).

Consideriamo ora il caso in cui si utilizza una relazione, tipicamente molti a molti, per descrivere un concetto che lega altri due concetti, come avviene nell'esempio seguente in cui l'esame è rappresentato da una relazione tra lo studente e il corso.

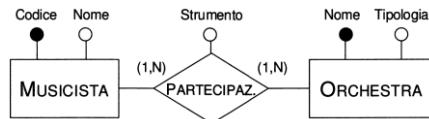


Il fatto che questo concetto abbia degli attributi associati non cambia la situazione, ci suggerisce piuttosto che, soprattutto nel caso in cui uno studente può sostenere più volte lo stesso esame, la soluzione corretta è lo schema nella figura sottostante, nel quale abbiamo **reificato la relazione** *ESAME* rappresentandola come entità. In questo caso, l'identificazione di un esame avviene attraverso lo studente, il corso e la data dell'esame.



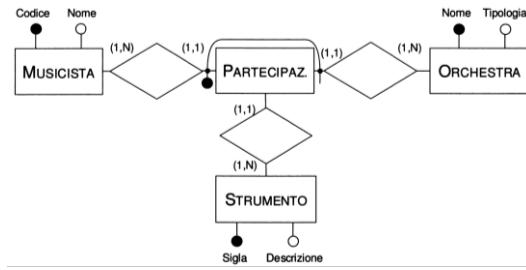
Una soluzione alternativa che non richiede un'identificazione esterna complessa è costituita dallo schema a destra, nel quale è stato introdotto un codice identificativo. Questa scelta semplifica le cose, ma bisogna tener conto del fatto che il codice è un concetto nuovo, non presente nelle specifiche e che quindi dovrà essere opportunamente gestito dal sistema informativo in via di sviluppo.

Consideriamo ora la relazione molti a molti che rappresenta la partecipazione di un musicista a un'orchestra con un certo strumento.

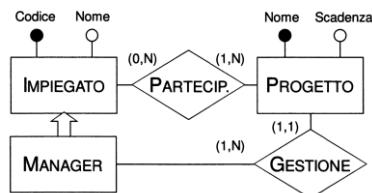


Il difetto di questo schema è semmai un altro e ha a che fare con quanto detto per la reificazione dell'attributo vista in precedenza: non stiamo rappresentando esplicitamente il concetto di strumento, che qui è solo una stringa. Se lo strumento è un concetto rilevante, dobbiamo reificare l'attributo della relazione e, per poterlo fare, dobbiamo reificare anche la relazione (**reificazione di associazione binaria e di un suo attributo**).

Pertanto, otteniamo lo schema seguente:



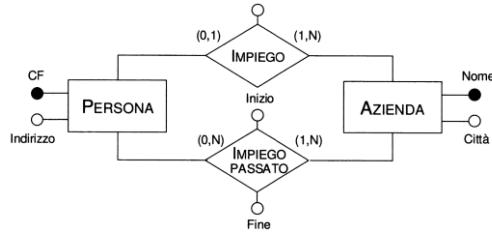
Passiamo ora ad alcuni pattern che coinvolgono le generalizzazioni. Un primo esempio di uso comune di questo costrutto è quello nel quale si vuole rappresentare un caso particolare di un altro (nell'esempio il sottoinsieme degli impiegati che sono dei manager). Si noti come sia possibile specializzare in questo modo i vari ruoli all'interno di un progetto (l'altra entità dello schema): la gestione è a carico solo dei manager.



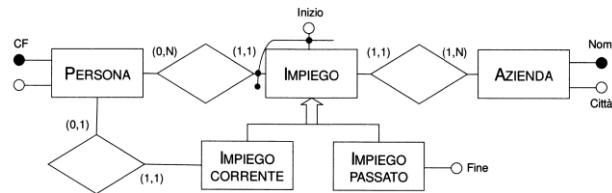
In questo schema è ragionevole assumere che un manager può gestire solo un progetto al quale partecipa. Questo implica che ogni coppia manager-progetto che compare tra le occorrenze della relazione *GESTIONE* deve comparire anche tra le occorrenze della relazione *PARTECIPAZIONE*. Questo vincolo però non può essere espresso direttamente sullo schema con un apposito costrutto e va quindi aggiunta una regola alla documentazione dello schema.

Lo schema successivo mostra un altro caso di uso comune del costrutto di generalizzazione. Si tratta di uno schema nel quale si vuole gestire la “**storicizzazione**” di un concetto

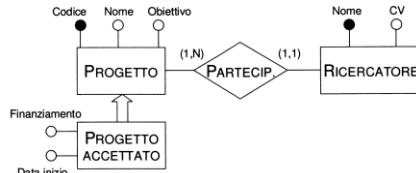
rappresentato da una relazione tra entità (nell'esempio gli impieghi presenti e passati di una persona).



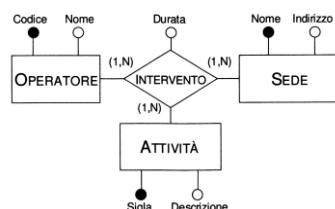
Una possibile soluzione consiste nel rappresentare separatamente i dati correnti e i dati storici e introdurre opportuni attributi per specificare gli intervalli di validità delle informazioni. Un'analisi attenta di questo schema ci fa comprendere che non possiamo rappresentare il fatto che una persona possa aver lavorato, in periodi diversi, per la stessa azienda. La soluzione in questo caso è, ancora una volta, la reificazione delle relazioni. Otteniamo così lo schema successivo, che consente di utilizzare una generalizzazione. Anche in questo caso risulta necessario l'inserimento di un vincolo esterno allo schema che impone che tutte le occorrenze della relazione *PERSONA* e *IMPIEGO CORRENTE* compaiano anche tra le occorrenze della relazione tra *PERSONA* e *IMPIEGO*.



L'ultimo esempio di uso comune del costrutto di generalizzazione è quello riportato di seguito. In questo schema vogliamo rappresentare il fatto che un certo concetto subisce un'evoluzione nel tempo che può essere diversa per le diverse occorrenze del concetto.

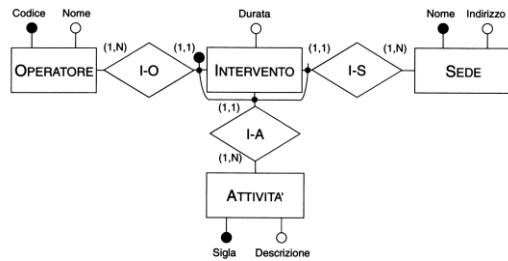


Consideriamo infine la relazione ternaria. Nell'esempio in figura è stata scelta una relazione ternaria perché si vuole modellare il caso in cui un operatore può effettuare operazioni che consistono in attività diverse svolte in sedi diverse. Inoltre, in ogni sede possono operare operatori diversi svolgendo attività diverse. Infine, le attività possono essere svolte da operatori diversi e in sedi diverse.



Anche questa relazione può essere reificata. Il nuovo schema modella esattamente la situazione dello schema originario perché la nuova entità risulta identificata da tutte le

entità originarie. Cambiando opportunamente l'identificazione siamo però in grado di modellare con questo pattern altre situazioni per le quali la relazione ternaria non sarebbe corretta.



Strategie di progetto

Nella **strategia top-down**, lo schema concettuale viene prodotto mediante una serie di raffinamenti successivi a partire da uno schema iniziale che descrive tutte le specifiche con pochi concetti molto astratti. Lo schema viene poi via via raffinato mediante opportune trasformazioni che aumentano il dettaglio dei vari concetti presenti.

Nel passaggio da un livello di raffinamento a un altro, lo schema viene modificato facendo uso di alcune trasformazioni elementari che vengono denominate *primitive di trasformazione top-down*. Esempi di primitive di trasformazione top-down sono:

- la definizione degli attributi di una entità o di una relazione;
- la reificazione di un attributo o di una entità;
- la decomposizione di una relazione in due relazioni distinte;
- la trasformazione di un'entità in una gerarchia di generalizzazione.

Il vantaggio della strategia top-down è che il progettista può descrivere inizialmente tutte le specifiche dei dati trascurandone i dettagli, per poi entrare nel merito di un concetto alla volta (si osservi infatti che le primitive di trasformazione agiscono su singoli concetti). Questo però è possibile solo quando si possiede, sin dall'inizio, una visione globale e astratta di *tutte* le componenti del sistema, ma ciò è estremamente difficile quando si ha a che fare con applicazioni di una certa complessità.

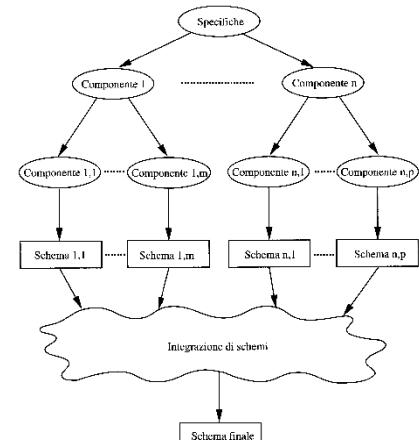
Strategia bottom-up

Nella **strategia bottom-up**, le specifiche iniziali sono suddivise in componenti via via sempre più piccole, fino a quando queste componenti descrivono un frammento elementare della realtà di interesse. A questo punto, le varie componenti vengono rappresentate da semplici schemi concettuali che possono consistere anche in singoli concetti. I vari schemi così ottenuti vengono poi fusi fino a giungere, attraverso una completa integrazione di tutte le componenti, allo schema concettuale finale.

A differenza della strategia top-down, con questa strategia i vari concetti presenti nello schema finale vengono via via introdotti durante le varie fasi.

Anche in questo caso, lo schema finale si ottiene attraverso alcune trasformazioni elementari che vengono denominate *primitive di trasformazione bottom-up* che introducono in uno schema nuovi concetti non presenti precedentemente e in grado di descrivere aspetti della realtà di interesse che non erano ancora stati rappresentati. Esempi di primitive di trasformazione bottom-up sono:

- l'introduzione di una nuova entità o di una relazione dall'analisi delle specifiche;
- l'individuazione nelle specifiche di un legame tra diverse entità riconducibile a una generalizzazione;
- l'aggregazione di una serie di attributi in una entità o in una relazione.

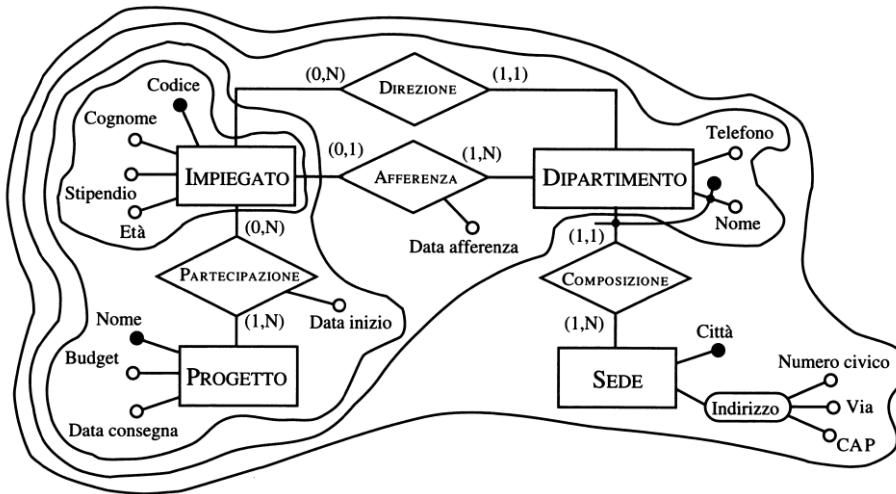


Il vantaggio della strategia bottom-up è che si adatta a una decomposizione del problema in componenti più semplici, facilmente individuabili, il cui progetto può essere affrontato anche da progettisti diversi. È quindi un tipo di strategia che si presta bene a lavori svolti in collaborazione o suddivisi all'interno di un gruppo. Lo svantaggio di questa strategia è invece il fatto che richiede delle operazioni di integrazione di schemi concettuali diversi che, nel caso di schemi complessi, presentano quasi sempre grosse difficoltà.

Strategia inside-out

Nella **strategia inside-out** (che può essere vista come un caso particolare della strategia bottom-up), si individuano inizialmente solo alcuni concetti importanti e poi si procede, a partire da questi, a “macchia d'olio”: si rappresentano prima i concetti in relazione con i concetti iniziali, per poi muoversi verso quelli più lontani attraverso una “navigazione” tra le specifiche.

Un esempio di sviluppo inside-out è quello mostrato di seguito, in cui le varie aree indicano un possibile sviluppo cronologico del progetto.



Questa strategia ha il vantaggio di non richiedere passi di integrazione. D'altro canto, è necessario, di volta in volta, esaminare tutte le specifiche per individuare concetti non ancora rappresentati e descrivere i nuovi concetti nel dettaglio (cosa non sempre possibile). Non è quindi possibile procedere per livelli di astrazione come avviene nella strategia top-down.

Strategia mista

Nella **strategia mista**, si cerca di combinare i vantaggi della strategia top-down con quelli della strategia bottom-up. Il progettista suddivide i requisiti in componenti separate, come nella strategia bottom-up, ma allo stesso tempo definisce uno *schema scheletro* contenente, a livello astratto, i concetti principali dell'applicazione. Questo schema scheletro fornisce una visione unitaria, sia pure astratta, dell'intero progetto e favorisce le fasi di integrazione degli schemi sviluppati separatamente.

La strategia mista è probabilmente la più flessibile tra le strategie viste, in quanto si adatta bene a esigenze contrapposte: quella di suddividere un problema complesso in sottoproblemi e quella di procedere per raffinamenti successivi. In effetti, questa strategia ingloba anche la strategia inside-out che, come abbiamo detto, è solo un caso particolare della strategia

bottom-up. È infatti abbastanza naturale, durante uno sviluppo bottom-up di una sottocomponente del progetto, procedere a macchia d'olio per rappresentare le specifiche della nostra base di dati non ancora rappresentate.

Qualità di uno schema concettuale

Nella costruzione di uno schema concettuale vanno comunque garantite alcune proprietà generali che uno schema concettuale di buona qualità deve possedere:

- **correttezza:** uno schema concettuale è *corretto* quando utilizza propriamente i costrutti messi a disposizione dal modello concettuale di riferimento. Gli errori possono essere *sintattici* o *semantici*. I primi riguardano un uso non ammesso di costrutto come, per esempio, una generalizzazione tra relazioni invece che tra entità. I secondi riguardano invece un uso di costrutti che non rispetta la loro definizione, come per esempio l'uso di una relazione per descrivere il fatto che una entità è specializzazione di un'altra.
- **completezza:** uno schema concettuale è *completo* quando rappresenta tutti i dati di interesse e quando tutte le operazioni possono essere eseguite a partire dai concetti descritti nello schema. La completezza di uno schema si può verificare controllando che tutte le specifiche sui dati siano rappresentate da qualche concetto presente nello schema che stiamo costruendo, e che tutti i concetti coinvolti in un'operazione presente nelle specifiche siano raggiungibili “navigando” attraverso lo schema.
- **leggibilità:** uno schema concettuale è *leggibile* quando rappresenta i requisiti in maniera naturale e facilmente comprensibile. Per garantire questa proprietà è necessario rendere lo schema autoesplicativo, per esempio, mediante una scelta opportuna dei nomi da dare ai concetti. La leggibilità di uno schema si può verificare facendo delle prove di comprensione con gli utenti.
- **minimalità:** uno schema è *minimale* quando tutte le specifiche sui dati sono rappresentate una sola volta nello schema. Uno schema, quindi, non è minimale quando esistono delle *ridondanze*. A differenza delle altre proprietà, non sempre una ridondanza è indesiderata, ma può nascere da precise scelte progettuali. In ogni caso però, queste situazioni vanno documentate. La minimalità di uno schema si può verificare per ispezione, controllando se esistono concetti che possono essere eliminati dallo schema che stiamo costruendo senza inficiare la sua completezza.

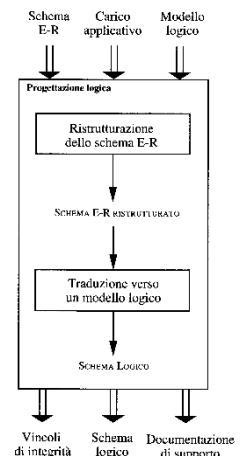
Progettazione logica

L'obiettivo della **progettazione logica** è quello di costruire uno schema logico in grado di descrivere, in maniera corretta ed efficiente, tutte le informazioni contenute nello schema E-R prodotto nella fase di progettazione concettuale. Tuttavia, non si tratta di una semplice traduzione da un modello a un altro perché, prima di passare allo schema logico, lo schema E-R va ristrutturato per soddisfare due esigenze: quella di “semplificare” la traduzione e quella di “ottimizzare” il progetto.

Le attività principali della progettazione logica sono la riorganizzazione dello schema concettuale e la traduzione in un modello logico:

- **ristrutturazione dello schema E-R**: è una fase indipendentemente dal modello logico scelto e si basa su criteri di ottimizzazione dello schema e di semplificazione della fase successiva.
- **traduzione verso il modello logico**: fa riferimento ad uno specifico modello logico (nel nostro caso il modello relazionale) e può includere una ulteriore ottimizzazione che si basa sulle caratteristiche del modello logico stesso.

I dati di ingresso della prima fase sono lo schema concettuale prodotto nella fase precedente e il *carico applicativo* previsto, in termini di dimensione dei dati e caratteristiche delle operazioni. Il risultato che si ottiene è uno schema E-R ristrutturato, che non è più uno schema concettuale nel senso stretto del termine, in quanto costituisce una rappresentazione dei dati che tiene conto degli aspetti realizzativi. Questo schema e il modello logico scelto costituiscono i dati di ingresso della seconda fase, che produce lo schema logico della nostra base di dati. Lo schema logico finale, i vincoli di integrità definiti su di esso e la relativa documentazione, costituiscono i prodotti finali della progettazione logica.



Analisi delle prestazioni su schemi E-R

È possibile effettuare studi di massima di due parametri che generalmente regolano le prestazioni dei sistemi software:

- **costo di un'operazione**: viene valutato in termini di numero di occorrenze di entità e associazioni che mediamente vanno visitate per rispondere ad un'operazione sulla base di dati;
- **occupazione di memoria**: viene valutato in termini dello spazio di memoria (misurato per esempio in numero di byte) necessario per memorizzare i dati descritti dallo schema.

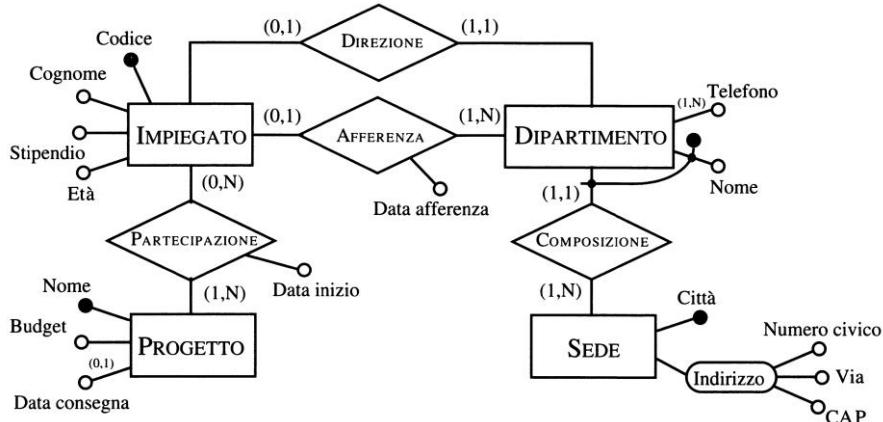
Per studiare questi parametri abbiamo bisogno di conoscere, oltre allo schema, le seguenti informazioni:

- **volume dei dati**, ossia
 - numero di occorrenze di ogni entità e associazione dello schema;
 - dimensioni di ciascun attributo (di entità o associazione).

- **caratteristiche delle operazioni:** ossia

- tipo dell'operazione;
 - frequenza (numero medio di esecuzioni in un certo intervallo di tempo);
 - dati coinvolti (entità e/o associazioni).

Per fare un esempio pratico, riprendiamo uno schema già incontrato e consideriamo le operazioni possibili che potrebbero essere eseguite.



Operazione 1: assegna un impiegato a un progetto.

Operazione 2: trova i dati di un impiegato, del dipartimento nel quale lavora e dei progetti ai quali partecipa.

Operazione 3: trova i dati di tutti gli impiegati di un certo dipartimento.

Operazione 4: per ogni sede, trova i suoi dipartimenti con il cognome del direttore e l'elenco degli impiegati del dipartimento.

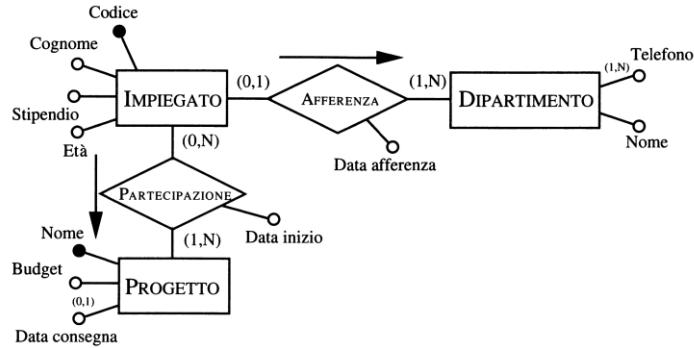
Sebbene un'analisi delle prestazioni che fa riferimento a un numero ristretto di operazioni possa sembrare riduttiva rispetto al reale carico della base di dati, va notato che le operazioni sulle basi di dati seguono la cosiddetta **regola “ottanta-venti”**: l'ottanta per cento del carico è generato dal venti per cento delle operazioni. Questo fatto ci consente di valutare adeguatamente il carico, concentrandoci solo sulle operazioni principali previste.

Nella **tavola dei volumi** vengono riportati tutti i concetti dello schema (entità e associazioni) con il volume previsto a regime. Il numero delle occorrenze delle associazioni dipende da due parametri: il numero di occorrenze delle entità coinvolte nelle associazioni e il numero (medio) di partecipazioni di una occorrenza di entità alle occorrenze di associazioni. Il secondo parametro dipende a sua volta dalle cardinalità delle associazioni. Ad esempio, nella tavola a seguire, assumendo che un impiegato partecipi in media a tre progetti, avremo $2000 \times 3 = 6000$ occorrenze per l'associazione *Partecipazione*.

Tavola 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

Inoltre, per ogni operazione, possiamo descrivere graficamente i dati coinvolti con uno **schema di operazione**, che consiste nel frammento dello schema E-R interessato dall'operazione, sul quale viene disegnato il “cammino logico” da percorrere per accedere alle informazioni d'interesse. Ad esempio, considerando l'operazione 2:



Avendo a disposizione queste informazioni, è possibile fare una stima del costo di un'operazione sulla base di dati contando il numero di accessi alle occorrenze di entità e associazioni necessario per eseguire l'operazione. Tutto questo può essere riassunto in una **tavola degli accessi**, dove nell'ultima colonna viene riportato il tipo di accesso: L per accesso in lettura e S per accesso in scrittura.

Tavola degli accessi

Concetto	Costrutto	Accessi	Tipo
Impiegato	Entità	1	L
Afferenza	Relazione	1	L
Dipartimento	Entità	1	L
Partecipazione	Relazione	3	L
Progetto	Entità	3	L

Ristrutturazione di schemi E-R

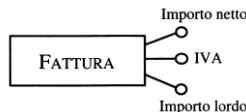
La fase di ristrutturazione di uno schema E-R si può suddividere in una serie di passi da effettuare in sequenza:

- **analisi delle ridondanze**: si decide se eliminare o mantenere eventuali ridondanze presenti nello schema;
- **eliminazione delle generalizzazioni**: tutte le generalizzazioni presenti nello schema vengono analizzate e sostituite da altri costrutti;
- **partizionamento/accorpamento di entità e associazioni**: si decide se è opportuno partizionare concetti dello schema (entità e/o associazioni) in più concetti o, viceversa, accorpare concetti separati in un unico concetto;
- **scelta degli identificatori principali**: si seleziona un identificatore per quelle entità che ne hanno più di uno.

Analisi delle ridondanze

Ricordiamo che una ridondanza in uno schema concettuale corrisponde alla presenza di un dato che può essere derivato (cioè ottenuto attraverso una serie di operazioni) da altri dati. In particolare, in uno schema E-R si possono presentare varie forme di ridondanza ed i più frequenti sono:

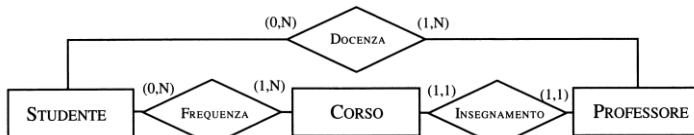
- attributi derivabili, occorrenza per occorrenza, da altri attributi della stessa entità (o associazione). Nell'esempio seguente, uno degli attributi (*Importo lordo*) è deducibile dagli altri attraverso una operazione di somma o differenza.



- attributi derivabili da attributi di altre entità (o associazioni), di solito attraverso funzioni aggregative. Nell'esempio a seguire, l'attributo *Importo totale* dell'entità *Acquisto* si può derivare, attraverso l'associazione *Composizione*, dall'attributo *Prezzo* dell'entità *Prodotto*, sommando i prezzi dei prodotti di cui un acquisto è composto.



- associazioni derivabili dalla composizione di altre associazioni in presenza di cicli. Nell'esempio a seguire, l'associazione *Docenza* tra studenti e professori può essere infatti derivata dalle associazioni *Frequenza* e *Insegnamento*. Va comunque precisato che la presenza di cicli non genera necessariamente ridondanze: se per esempio, al posto dell'associazione *Docenza*, ci fosse stata un'associazione *Tesi* rappresentante il legame tra studente e relatori (concetto indipendente dal fatto che un professore è un docente dello studente) allora lo schema non sarebbe stato ridondante.

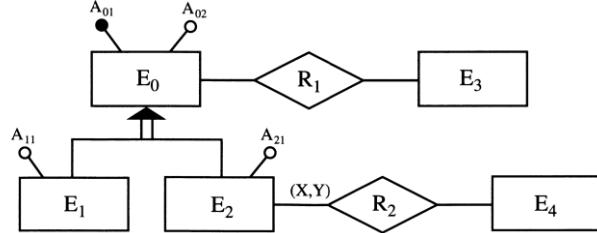


La presenza di un dato derivato presenta un vantaggio e alcuni svantaggi. Il vantaggio è una riduzione degli accessi necessari per calcolare il dato derivato, gli svantaggi sono una maggiore occupazione di memoria (che è comunque spesso un costo trascurabile) e la necessità di effettuare operazioni aggiuntive per mantenere il dato derivato aggiornato.

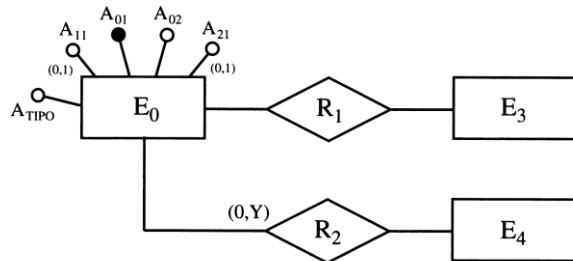
La decisione di mantenere o eliminare una ridondanza va quindi presa confrontando costo di esecuzione delle operazioni che coinvolgono il dato ridondante e relativa occupazione di memoria, nei casi di presenza e assenza della ridondanza.

Eliminazione delle generalizzazioni

Per rappresentare una generalizzazione mediante entità e associazioni abbiamo essenzialmente tre alternative possibili. Per presentare queste alternative, faremo riferimento al seguente schema E-R:

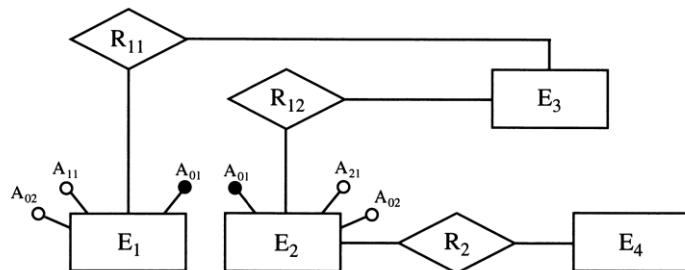


- Accorpamento delle figlie della generalizzazione nel genitore.** Le entità E_1 ed E_2 vengono eliminate e le loro proprietà (attributi e partecipazioni ed associazioni e generalizzazioni) vengono aggiunte all'entità genitore E_0 . A tale entità viene aggiunto un ulteriore attributo che serve a distinguere il “tipo” di un'occorrenza di E_0 , cioè se tale occorrenza apparteneva a E_1 , a E_2 o, nel caso di generalizzazione non totale, a nessuna di esse.



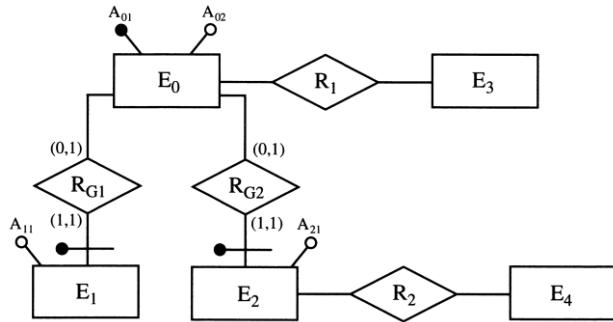
Si osservi che gli attributi A_{11} e A_{21} possono assumere valori nulli (perché non significativi) per alcune occorrenze di E_0 e che la relazione R_2 avrà, in ogni caso, una cardinalità minima pari a 0 sull'entità E_0 (perché le occorrenze di E_2 sono solo un sottoinsieme delle occorrenze di E_0).

- Accorpamento del genitore della generalizzazione nelle figlie.** L'entità genitore E_0 viene eliminata e, per la proprietà dell'ereditarietà, i suoi attributi, il suo identificatore e le relazioni a cui tale entità partecipava, vengono aggiunti alle entità figlie E_1 ed E_2 . Le relazioni R_{11} e R_{12} rappresentano rispettivamente la restrizione della relazione R_1 sulle occorrenze delle entità E_1 ed E_2 .



- Sostituzione della generalizzazione con associazioni.** La generalizzazione si trasforma in due associazioni uno a uno che legano rispettivamente l'entità genitore con le entità figlie E_1 ed E_2 . Non ci sono trasferimenti di attributi o associazioni e le entità E_1 ed E_2 sono identificate esternamente dall'entità E_0 . Nello schema ottenuto vanno aggiunti però dei vincoli: ogni occorrenza di E_0 non può partecipare

contemporaneamente a R_{G1} e R_{G2} ; inoltre, se la generalizzazione è totale, ogni occorrenza di E_0 deve partecipare o a un'occorrenza di R_{G1} oppure a un'occorrenza di R_{G2} .



La scelta tra le varie alternative può essere fatta in maniera analoga a quanto fatto per i dati derivati, considerando vantaggi e svantaggi di ognuna delle scelte possibili relativamente all'occupazione di memoria e al costo delle operazioni coinvolte. È possibile comunque stabilire alcune regole di carattere generale:

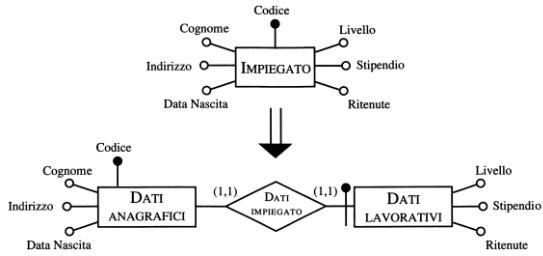
- l'alternativa (1) è conveniente quando le operazioni non fanno molta distinzione tra le occorrenze e tra gli attributi di E_0 , E_1 ed E_2 . In questo caso infatti, anche se abbiamo uno spreco di memoria per la presenza di valori nulli, la scelta ci assicura un numero minore di accessi rispetto alle altre nelle quali le occorrenze e gli attributi sono distribuiti tra le varie entità;
- l'alternativa (2) è possibile solo se la generalizzazione è totale, altrimenti le occorrenze di E_0 che non sono occorrenze né di E_1 né di E_2 non sarebbero rappresentate. È conveniente quando ci sono operazioni che si riferiscono solo ad occorrenze di E_1 oppure di E_2 , e dunque fanno delle distinzioni tra tali entità. In questo caso, abbiamo un risparmio di memoria rispetto alla scelta (1), perché, in linea di principio, gli attributi non assumono mai valori nulli. Inoltre, c'è una riduzione degli accessi rispetto alla scelta (3) perché non si deve visitare E_0 per accedere ad alcuni attributi di E_1 ed E_2 ;
- l'alternativa (3) è conveniente quando la generalizzazione non è totale (sebbene ciò non sia necessario) e ci sono operazioni che si riferiscono solo ad occorrenze di E_1 (E_2) oppure di E_0 , e dunque fanno delle distinzioni tra entità figlia ed entità genitore. In questo caso, abbiamo un risparmio di memoria rispetto alla scelta (1), per l'assenza di valori nulli, ma c'è un incremento degli accessi per mantenere la consistenza delle occorrenze rispetto ai vincoli introdotti.

Partizionamento/accorpamento di concetti

Entità e associazioni in uno schema E-R possono essere **partizionati** o **accorpati** per garantire una maggior efficienza delle operazioni in base al seguente principio: gli accessi si riducono separando attributi di uno stesso concetto che vengono acceduti da operazioni diverse e raggruppando attributi di concetti diversi che vengono acceduti dalle medesime operazioni.

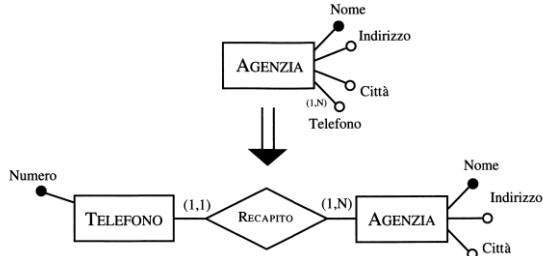
Un esempio di **partizionamento di entità** è quello mostrato nella figura seguente, in cui l'entità *Impiegato* viene sostituita da due entità, collegate da un'associazione uno a uno, che

descrivono rispettivamente i dati anagrafici degli impiegati e i dati relativi alla loro retribuzione.



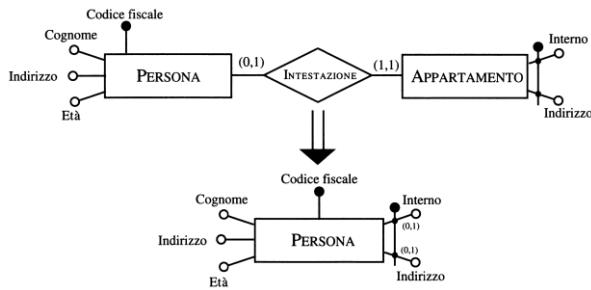
Un partizionamento di questo tipo è un esempio di **decomposizione verticale** di un'entità, nel senso che si suddivide il concetto operando sui suoi attributi. È comunque possibile effettuare anche delle *decomposizioni orizzontali* nelle quali la suddivisione avviene sulle occorrenze dell'entità (corrisponderebbero quindi all'introduzione di generalizzazioni a livello logico). I partizionamenti orizzontali hanno un effetto collaterale: dover duplicare tutte le associazioni a cui l'entità originaria partecipa.

Un particolare tipo di partizionamento è quello che riguarda l'**eliminazione di attributi multivalore**. Questa ristrutturazione si rende necessaria perché, come per le generalizzazioni, il modello relazionale non permette di rappresentare in maniera diretta questo tipo di attributo. Un esempio è il seguente:



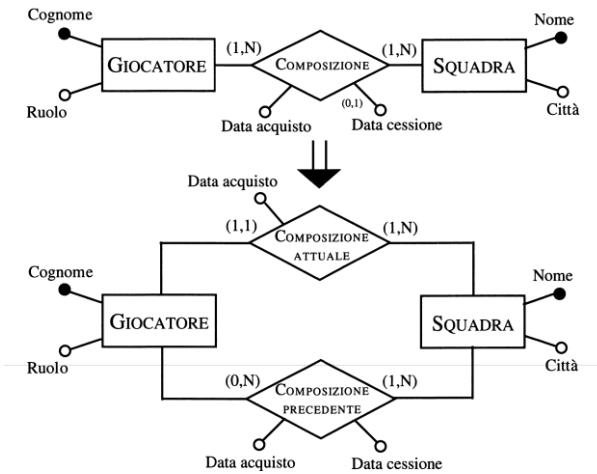
Qui, l'entità *Agenzia* avente l'attributo multivalore *Telefono* viene partizionata in due entità: un'entità con lo stesso nome e gli stessi attributi dell'entità originale eccetto l'attributo multivalore, e l'entità *Telefono*, con il solo attributo *Numero*, legata mediante un'associazione uno a molti con l'entità *Agenzia*.

Invece, un esempio di **accorpamento di entità** (operazione inversa del partizionamento) è quello della figura seguente, in cui la ristrutturazione può essere suggerita dal fatto che le operazioni più frequenti sull'entità *Persona* richiedono sempre i dati relativi all'appartamento che occupa e vogliamo quindi risparmiare gli accessi necessari per risalire a questi dati attraverso l'associazione che li lega.



Abbiamo parlato finora di partizionamento e accorpamento di entità, ma lo stesso discorso si può estendere alle associazioni: può convenire, in alcuni casi, **decomporre un'associazione** tra due entità in due (o più) associazioni tra le medesime entità, per separare occorrenze dell'associazione originale accedute sempre separatamente e, viceversa, **accoppare due (o più) associazioni** tra le medesime entità (che si riferiscono però a due aspetti dello stesso concetto) in un'unica associazione, quando le relative occorrenze vengono sempre accedute contemporaneamente.

Un esempio di partizionamento di associazioni è quello di seguito, in cui vengono distinti i giocatori che compongono *attualmente* una squadra da quelli che ne facevano parte nel passato.



Scelta degli identificatori principali

La **scelta degli identificatori principali** è essenziale nelle traduzioni verso il modello relazionale perché in questo modello le chiavi vengono usate per stabilire legami tra dati in relazioni diverse. Inoltre, i DBMS richiedono generalmente di specificare una *chiave primaria* sulla quale vengono costruite automaticamente delle strutture ausiliarie, dette *indici*, per il reperimento efficiente dei dati. Quindi, nei casi in cui esistono entità per le quali sono stati specificati più identificatori, bisogna decidere quale di questi verrà utilizzato come chiave primaria.

I criteri di decisione per questa scelta sono i seguenti:

- gli attributi con valori nulli non possono costituire identificatori principali;
- un identificatore composto da uno o da pochi attributi è da preferire a identificatori costituiti da molti attributi;
- un identificatore interno con pochi attributi è da preferire a un identificatore esterno, che magari coinvolge diverse entità;
- un identificatore che viene utilizzato da molte operazioni per accedere alle occorrenze di un'entità è da preferire rispetto agli altri.

A questo punto, se nessuno degli identificatori candidati soddisfa tali requisiti, è possibile pensare di introdurre un ulteriore attributo all'entità: questo attributo conterrà valori speciali (detti *codici*) generati appositamente per identificare le occorrenze delle entità.

Traduzione verso il modello relazionale

La seconda fase della progettazione logica corrisponde a una traduzione tra modelli di dati diversi: a partire da uno schema E-R ristrutturato si costruisce uno schema logico *equivalente*, in grado cioè di rappresentare le medesime informazioni. L'idea di base prevede che: (i) le entità diventano relazioni sugli stessi attributi, (ii) le relationship diventano relazioni sugli identificatori delle entità coinvolte (più gli attributi propri).

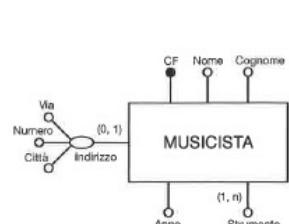
Per quanto riguarda la **traduzione di entità forte**, i passaggi sono:

1. creare uno schema relazionale il cui nome è il nome dell'entità;
2. gli attributi dello schema relazionale sono tutti gli attributi dell'entità;
3. l'attributo che identifica l'entità è la chiave primaria dello schema relazionale.

Per quanto riguarda la **traduzione di attributi multivaleore**, i passaggi sono:

1. creare uno schema relazionale (*sr*) il cui nome è il nome dell'attributo multivaleore (*am*);
2. gli attributi del nuovo schema sono: il nome dell'attributo e la chiave primaria dello *sr* relativo all'entità;
3. la chiave primaria dello schema generato sarà la composizione di tutti gli attributi;
4. aggiungere un vincolo di integrità referenziale (*vir*) tra lo *sr* dell'*am* e lo *sr* principale.

Un esempio è il seguente:



MUSICISTA	CF	Nome	Cognome	Anno nascita	Via	Numeri	Città
t ₁	cf123	Mario	Rossi	1980	Roma	7	Padova
t ₂	cf456	Alberto	Bianchi	1988	NULL	NULL	NULL
t ₃	cf678	Alessandro	Verdi	1973	Milano	8b	Firenze
t ₄	cf890	Stefano	Neri	1995	Verdi	140	Venezia
t ₅	cf098	Alessandro	Bianchi	1992	XX Settembre	57	Milano

STRUMENTO	Nome	Musicista
t ₁	Piano	cf123
t ₂	Sax	cf678
t ₃	Violino	cf890
t ₄	Violino	cf123

Per la **traduzione di un'associazione molti a molti**, i passaggi sono:

1. trasformare le due entità nei due schemi relazionali (*sr*) corrispondenti;
2. aggiungere uno *sr* corrispondente all'associazione che ha come attributi le chiavi primarie *kp* delle due entità partecipanti ed eventuali attributi dell'associazione;
3. la *kp* di questo nuovo schema sarà la composizione delle due *kp*;
4. aggiungere due *vir* tra gli attributi della chiave composta e le *kp*.

Un esempio è il seguente:



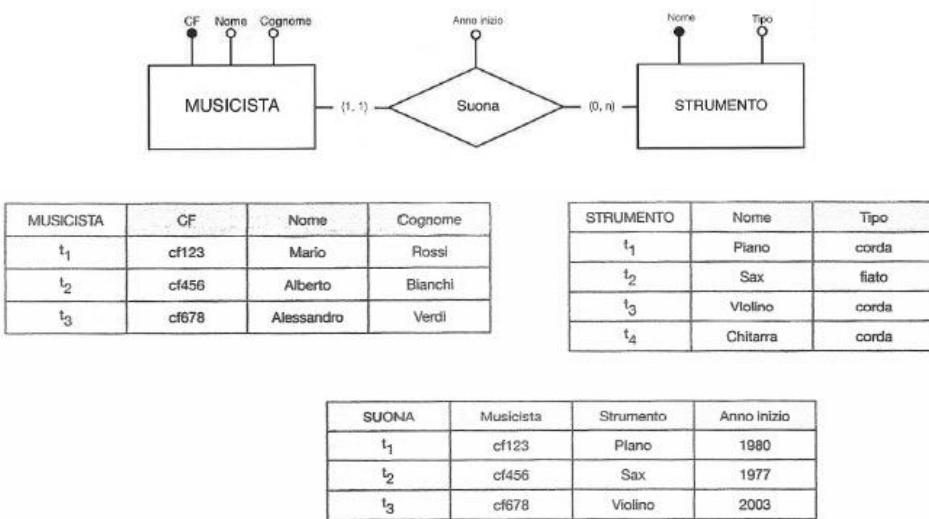
MUSICISTA	CF	Nome	Cognome	STRUMENTO	Nome	Tipo
t ₁	cf123	Mario	Rossi	t ₁	Piano	corda
t ₂	cf456	Alberto	Bianchi	t ₂	Sax	fisso
t ₃	cf678	Alessandro	Verdi	t ₃	Violino	corda

SUONA	Musicista	Strumento	Anno inizio
t ₁	cf123	Piano	1980
t ₂	cf123	Violino	1979
t ₃	cf456	Sax	1977
t ₄	cf678	Violino	2003

Per la **traduzione di un'associazione con cardinalità massima 1**, i passaggi sono:

1. trasformare le due entità nei due schemi *sr* corrispondenti;
2. aggiungere uno *sr* corrispondente all'associazione che ha come attributi le *kp* delle due entità partecipanti ed eventuali attributi dell'associazione;
3. la *kp* di questo nuovo schema è la *kp* dell'entità che partecipa con cardinalità massima pari ad 1;
4. aggiungere due *vir* tra gli attributi del nuovo schema e quelli derivati dalle entità forti.

Un esempio è il seguente:



Tuttavia, la soluzione proposta non è *ottima*, in quanto ci sono ridondanze sui dati dell'attributo *Musicista* nello schema relazionale *Suona*. Una soluzione ottima sarebbe la seguente:

MUSICISTA	CF	Nome	Cognome	Strumento	Anno inizio
t_1	cf123	Mario	Rossi	Piano	1980
t_2	cf456	Alberto	Bianchi	Sax	1977
t_3	cf678	Alessandro	Verdi	Violino	2003

STRUMENTO	Nome	Tipo
t_1	Piano	corda
t_2	Sax	fiamto
t_3	Violino	corda
t_4	Chitarra	corda

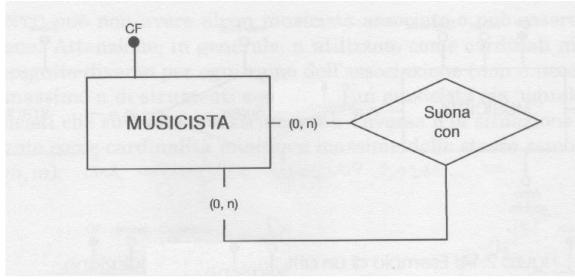
Per la **traduzione di un'associazione con partecipazione (1, 1)**, i passaggi sono:

1. trasformare l'entità che partecipa con cardinalità massima *n* nello *sr* corrispondente;
2. l'entità che partecipa con cardinalità massima 1 si trasforma allo stesso modo, ma ad essa vanno aggiunti sia gli attributi della *kp* dell'altro schema, sia gli eventuali attributi dell'associazione;
3. la *kp* di quest'ultimo schema è la *kp* dell'entità che partecipa con cardinalità massima pari ad 1;
4. aggiungeremo *vir* sugli attributi aggiunti dall'altra entità.

Per la **traduzione di un'associazione ricorsiva**, i passaggi sono:

1. trasformare l'entità forte nello *sr* corrispondente;
2. aggiungere uno *sr* corrispondente all'associazione che ha come attributi la *kp* dello schema che però deve essere ripetuta due volte, con nomi diversi, ed eventuali attributi dell'associazione;
3. la *kp* di questo nuovo schema sarà la coppia delle due *kp*;
4. aggiungere due *vir* tra gli attributi della chiave composta e la *kp*.

Un esempio è il seguente (non conta l'ordine) :

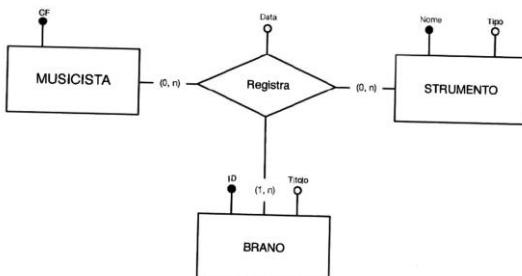


MUSICISTA	CF	Nome	Cognome
t ₁	cf123	Mario	Rossi
t ₂	cf456	Alberto	Bianchi
t ₃	cf678	Alessandro	Verdi
t ₃	cf890	Stefano	Neri

SUONA CON	Musicista1	Musicista2
t ₁	cf123	cf678
t ₂	cf123	cf456
t ₃	cf678	cf123
t ₄	cf456	cf123

Per la **traduzione di associazioni ternarie**, i passaggi sono simili a quelli della traduzione di associazioni binarie:

- trasformare le tre entità nei tre *sr* corrispondenti;
- aggiungere uno *sr* corrispondente all'associazione che ha come attributi le *kp* delle tre entità partecipanti ed eventuali attributi dell'associazione;
- la *kp* di questo nuovo schema sarà la composizione delle tre *kp*;
- aggiungere tre *vir* tra gli attributi della chiave composta e le *kp*.

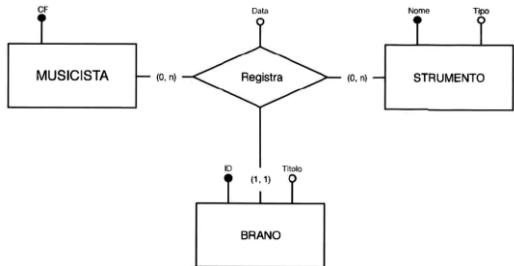


REGISTRA	Musicista	Strumento	Brano	Data
t ₁	cf123	Piano	1	2010
t ₂	cf123	Violino	1	2010
t ₃	cf456	Sax	2	1988
t ₄	cf123	Violino	3	2009

Per la **traduzione di associazioni ternarie con cardinalità massima 1**, ricordando che se la cardinalità è (1,1) possiamo portare le chiavi delle entità in quella con cardinalità (1,1) (*trasformazione ottima*), i passaggi sono:

- trasformare le due entità che partecipano con cardinalità massima pari a *n* nei due *sr* corrispondenti;
- trasformare l'entità che partecipa con cardinalità massima pari a 1 aggiungendo alla lista degli attributi le *kp* degli altri due schemi e gli eventuali attributi dell'associazione;

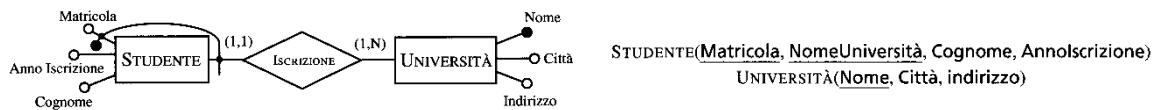
- aggiungere due *vir* tra gli attributi aggiunti a quest'ultimo schema e le *kp* dei due schemi corrispondenti.



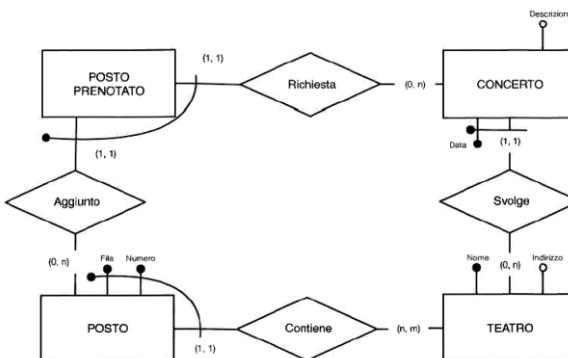
BRANO	ID	Titolo	Musicista	Strumento	Data
t ₁	1	Le onde	cf123	Piano	2010
t ₂	2	Lontano	cf456	Sax	1996
t ₃	3	OMBRE	cf123	Violino	2009

Per la **traduzione di un'entità debole**, i passaggi sono:

- trasformare l'entità forte nello *sr* corrispondente;
- trasformare l'entità debole aggiungendo alla lista degli attributi
 - gli attributi della *kp* dello schema relativo all'entità forte,
 - gli eventuali attributi dell'associazione;
- la *kp* di quest'ultimo schema è data dalla composizione degli attributi che identificano l'entità debole (che includono l'identificatore dell'entità forte);
- aggiungere il *vir* necessario.



Quello di seguito è invece un caso di entità deboli in cascata, dove ci troveremmo ad avere i gli schemi Teatro(Nome, Indirizzo), Concerto(Data, Teatro, Descrizione), Posto(Fila, Numero, Teatro), PostoPrenotato(PostoFila, PostoNumero, PostoTeatro, ConcertoData, ConcertoTeatro).



In generale, i passaggi per la **traduzione di uno schema E-R** sono:

- trasformare le entità forti che partecipano con cardinalità massima pari a *n* a tutte le associazioni ad esse collegate;
- trasformare le entità forti che partecipano con cardinalità (1, 1) ad almeno una delle associazioni ad esse collegate;
- trasformare le entità forti che partecipano con cardinalità (0, 1) ad almeno una delle associazioni ad esse collegate;
- trasformare le entità deboli;
- trasformare tutte le associazioni rimaste.

Normalizzazione

Introduciamo i primi concetti attraverso un esempio. Consideriamo la relazione in figura, che ha come chiave l'insieme costituito dagli attributi *Impiegato* e *Progetto*.

Impiegato	Stipendio	Progetto	Bilancio	Funzione
Rossi	20 000	Marte	2000	tecnico
Verdi	35 000	Giove	15 000	progettista
Verdi	35 000	Venere	15 000	progettista
Neri	55 000	Venere	15 000	direttore
Neri	55 000	Giove	15 000	consulente
Neri	55 000	Marte	2000	consulente
Mori	48 000	Marte	2000	direttore
Mori	48 000	Venere	15 000	progettista
Bianchi	48 000	Venere	15 000	progettista
Bianchi	48 000	Giove	15 000	direttore

Si può facilmente verificare che le tuple della relazione soddisfano le seguenti proprietà:

- lo stipendio di ciascun impiegato è unico ed è funzione del solo impiegato, indipendentemente dai progetti cui partecipa;
- il bilancio di ciascun progetto è unico e dipende dal solo progetto, indipendentemente dagli impiegati che vi partecipano.

Questi fatti hanno alcune conseguenze sul contenuto della relazione e sulle operazioni che si possono effettuare su di essa. Ad esempio:

- il valore dello stipendio di ciascun impiegato è ripetuto in tutte le tuple relative a esso: si ha quindi una **ridondanza**; se per esempio un impiegato partecipasse a 20 progetti, il suo stipendio verrebbe ripetuto 20 volte;
- se lo stipendio di un impiegato varia, è necessario andarne a modificare il valore in tutte le tuple corrispondenti affinché la dipendenza continui a valere; questo inconveniente, che comporta la necessità di effettuare più modifiche contemporaneamente, va sotto il nome di **anomalia di aggiornamento**;
- se un impiegato interrompe la partecipazione a tutti i progetti senza lasciare l'azienda, e quindi tutte le corrispondenti tuple vengono eliminate, non è possibile conservare traccia del suo nome e del suo stipendio (a meno di ammettere valori nulli sull'unica chiave, il che è inammissibile), che potrebbero rimanere di interesse; questo problema viene indicato come **anomalia di cancellazione**;
- analogamente, se si hanno informazioni su un nuovo impiegato, non è possibile inserirlo finché questi non viene assegnato a un progetto; in questo caso parliamo di **anomalia di inserimento**.

Una motivazione intuitiva della presenza di questi inconvenienti può essere la seguente: abbiamo usato un'unica relazione per rappresentare informazioni eterogenee.

Dipendenze funzionali

Consideriamo ancora la relazione rappresentata nella figura poco sopra. Abbiamo osservato che lo stipendio di ciascun impiegato è unico e quindi, ogni volta che in una tupla della relazione compare un certo impiegato, il valore del suo stipendio rimane sempre lo stesso. Possiamo cioè dire che il valore dell'attributo *Impiegato* **determina** il valore dell'attributo *Stipendio* o, in maniera più precisa, che esiste una funzione che associa a ogni elemento del dominio dell'attributo *Impiegato* che compare nella relazione un solo elemento del dominio dell'attributo *Stipendio*.

Un discorso analogo si può fare per il legame che intercorre tra gli attributi *Progetto* e *Bilancio* perché il valore del progetto determina il valore del bilancio del progetto stesso e quindi tutte le volte che nella relazione compare il nome di un progetto, il bilancio a esso associato sarà sempre lo stesso.

Questo concetto può essere formalizzato come segue. Data una relazione r su uno schema $R(X)$ e due sottoinsiemi di attributi non vuoti Y e Z , diremo che esiste su r una **dipendenza funzionale** tra Y e Z se, per ogni coppia di tuple t_1 e t_2 di r aventi gli stessi valori sugli attributi Y , risulta che t_1 e t_2 hanno gli stessi valori anche sugli attributi Z . Una dipendenza funzionale tra gli attributi Y e Z viene generalmente indicata con la notazione $Y \rightarrow Z$.

Tornando al nostro esempio possiamo dunque dire che sulla relazione della figura esistono le dipendenze funzionali *Impiegato* \rightarrow *Stipendio* e *Progetto* \rightarrow *Bilancio*.

Ci sono delle osservazioni da fare sulle FD (dipendenze funzionali):

1. se l'insieme Z è composto dagli attributi A_1, \dots, A_k , allora una relazione soddisfa $Y \rightarrow Z$ se e solo se essa soddisfa tutte le k dipendenze $Y \rightarrow A_1, \dots, Y \rightarrow A_k$.
2. Possiamo notare che, nella nostra relazione, è verificata anche la dipendenza funzionale *Impiegato* \rightarrow *Progetto*, in quanto due tuple con gli stessi valori sulla coppia di attributi *Impiegato* e *Progetto*, hanno ovviamente lo stesso valore sull'attributo *Progetto*, che è uno dei due.

Diremo quindi che, in generale, una dipendenza funzionale $Y \rightarrow A$ è *non banale* se A non compare tra gli attributi di Y .

Un'ultima osservazione sulle dipendenze funzionali riguarda il loro legame con il vincolo di chiave. Se prendiamo una chiave K di una relazione r , si può facilmente verificare che esiste una dipendenza funzionale tra K e ogni altro attributo dello schema di r . Questo perché, per definizione stessa di vincolo di chiave, non possono esistere due tuple con gli stessi valori su K e quindi una dipendenza funzionale che ha K al primo membro sarà sempre soddisfatta.

In particolare, esisterà una dipendenza funzionale tra una chiave di una relazione e tutti gli attributi dello schema della relazione. Nel nostro caso abbiamo cioè che: *Impiegato Progetto* \rightarrow *Stipendio Bilancio Funzione*.

Possiamo quindi concludere dicendo che il vincolo di dipendenza funzionale *generalizza* il vincolo di chiave. Più precisamente, possiamo dire che una dipendenza funzionale $Y \rightarrow Z$ su uno schema $R(X)$ degenera nel vincolo di chiave se $Y \cup Z = X$. In tal caso infatti, Y è (super)chiave per lo schema $R(X)$.

Forma normale e normalizzazione

Una **forma normale** è una proprietà di una base di dati relazionale che ne garantisce la “qualità”, cioè l’assenza di determinati difetti (anomalie). Quando una relazione non è normalizzata:

- presenta ridondanze;
- si presta a comportamenti poco desiderabili durante gli aggiornamenti.

Data una relazione che non soddisfa la forma normale è possibile, in molti casi, sostituirla con due o più relazioni normalizzate attraverso un processo detto di **normalizzazione**. Questo processo si fonda su un semplice criterio: se una relazione rappresenta più concetti indipendenti, allora va decomposta in relazioni più piccole, una per ogni concetto.

Una relazione r è in **forma normale di Boyce e Codd** (BCFN) se per ogni dipendenza funzionale (non banale) $X \rightarrow A$ definita su di essa, X contiene una chiave K di r , cioè X è superchiave per r . Anomalie e ridondanze non si presentano per relazioni in BCFN, perché i concetti indipendenti sono separati, uno per relazione.

Proprietà delle decomposizioni

Esaminiamo più in dettaglio il concetto di decomposizione, notando come non tutte le decomposizioni siano desiderabili e individuando alcune proprietà essenziali che devono essere soddisfatte da una “buona” decomposizione.

Per discutere la prima proprietà, **decomposizione senza perdita**, esaminiamo la relazione seguente.

Impiegato	Progetto	Sede
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano

Tale relazione soddisfa le dipendenze funzionali $\text{Impiegato} \rightarrow \text{Sede}$ e $\text{Progetto} \rightarrow \text{Sede}$ che, sostanzialmente, specificano il fatto che ciascun impiegato opera presso un’unica sede e che ciascun progetto è sviluppato presso un’unica sede. Separando sulla base delle dipendenze, saremmo portati a decomporre la relazione in due parti:

- una relazione sugli attributi *Impiegato* e *Sede*, in corrispondenza alla prima dipendenza;
- l’altra sugli attributi *Progetto* e *Sede*, in corrispondenza alla seconda dipendenza funzionale.

Impiegato	Sede	Progetto	Sede
Rossi	Roma	Marte	Roma
Verdi	Milano	Giove	Milano
Neri	Milano	Saturno	Milano
		Venere	Milano

La ricostruzione della relazione originaria a partire dalle sue proiezioni (cioè la ricostruzione di tutte le sue tuple a partire dalle tuple nelle proiezioni) deve essere effettuata per mezzo di un’operazione di join naturale delle due proiezioni. Purtroppo, il join naturale delle due relazioni produce una relazione diversa da quella originaria.

La situazione nell’esempio corrisponde al caso generale: data una relazione r su un insieme di attributi X , se X_1 e X_2 sono due sottoinsiemi di X la cui unione sia pari a X stesso, allora

il join delle due relazioni ottenute per proiezione da r su X_1 e X_2 , rispettivamente, è una relazione che contiene tutte le tuple di r , più eventualmente altre, che possiamo chiamare “*spurie*”. Diciamo che r si **decomponе senza perdita** su X_1 e X_2 se il join delle due proiezioni è uguale a r stessa (cioè non contiene tuple spurie).

In altre parole, possiamo dire che r si decomponе senza perdita su due relazioni se l’insieme degli attributi comuni alle due relazioni è chiave per almeno una delle relazioni decomposte. Nell’esempio, possiamo vedere che l’intersezione degli insiemi di attributi su cui abbiamo effettuato le due proiezioni è costituita dall’attributo *Sede*, che non è il primo membro di alcuna dipendenza funzionale.

Per introdurre la seconda proprietà, **conservazione delle dipendenze**, possiamo esaminare di nuovo la relazione di prima. Volendo ancora rimuovere le anomalie, potremmo pensare di sfruttare solo la dipendenza *Impiegato* → *Sede* per ottenere una decomposizione senza perdita. Otteniamo in questa maniera due relazioni: una sugli attributi *Impiegato* e *Sede* e l’altra sugli attributi *Impiegato* e *Progetto*.

<i>Impiegato</i>	<i>Sede</i>	<i>Impiegato</i>	<i>Progetto</i>
Rossi	Roma	Rossi	Marte
Verdi	Milano	Verdi	Giove
Neri	Milano	Neri	Venere

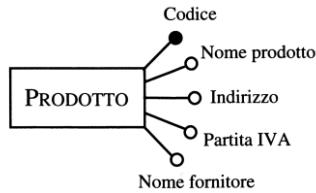
Il join di queste due relazioni produce effettivamente la relazione originaria, che possiamo quindi dire che si decomponе senza perdita. Questa decomposizione presenta però un altro inconveniente, che possiamo rilevare nel modo seguente. Supponiamo di voler inserire una nuova tupla che specifica la partecipazione dell’impiegato Neri, che opera a Milano, al progetto Marte. Sulla relazione originaria un tale aggiornamento verrebbe immediatamente individuato come illecito, perché porterebbe a una violazione della dipendenza *Progetto* → *Sede*. Sulle relazioni decomposte, invece, non è possibile rilevare alcuna violazione di dipendenze; possiamo quindi notare come non sia possibile effettuare alcuna verifica sulla dipendenza *Progetto* → *Sede*, perché i due attributi *Progetto* e *Sede* sono stati separati: uno in una relazione e l’altro nell’altra.

Generalizzando, possiamo quindi concludere che, in ogni decomposizione, ciascuna delle dipendenze funzionali dello schema originario dovrebbe coinvolgere attributi che compaiono tutti insieme in uno degli schemi composti. In questo modo, è possibile garantire, sullo schema decomposto, il soddisfacimento degli stessi vincoli il cui soddisfacimento è garantito dallo schema originario. Diremo che una decomposizione che soddisfa tale proprietà **conserva le dipendenze** dello schema originario.

Verifiche di normalizzazione su entità

È possibile considerare ciascuna entità come una relazione che ha per attributi gli attributi dell’entità (per le entità con identificatore esterno sono necessari ulteriori attributi in corrispondenza alle entità che partecipano all’identificazione). La verifica di normalizzazione può quindi procedere come visto finora: è sufficiente considerare le dipendenze funzionali che sussistono fra gli attributi dell’entità e verificare che ciascuna di esse abbia come primo membro l’identificatore (o lo contenga).

Per esempio, consideriamo la seguente entità:



Sussiste la dipendenza $\text{PartitaIVA} \rightarrow \text{NomeFornitore}$ Indirizzo . Inoltre, tutti gli attributi dipendono funzionalmente dall'attributo *Codice*, che costituisce quindi l'identificatore dell'entità. Poiché l'unico identificatore dell'entità è l'attributo *Codice*, possiamo concludere che l'entità viola la terza forma normale, in quanto la dipendenza $\text{PartitaIVA} \rightarrow \text{NomeFornitore}$ Indirizzo ha un primo membro che non contiene l'identificatore e un secondo membro composto da attributi che non fanno parte della chiave. In questi casi, la verifica di normalizzazione ci permette di segnalare il fatto che lo schema concettuale non è accurato e ci suggerisce di decomporre l'unità stessa.

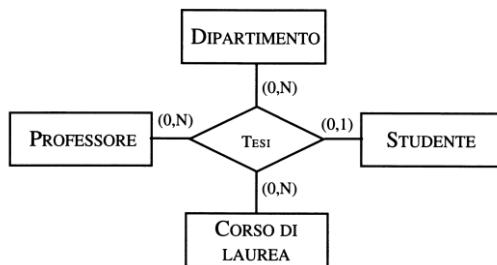
Nel nostro caso, possiamo dire che è opportuno modellare il concetto di fornitore per mezzo di una entità, con identificatore costituito dall'attributo *PartitaIVA* e ulteriori attributi *NomeFornitore* e *Indirizzo*.

Poiché nello schema originario gli attributi di *Prodotto* e *Fornitore* compaiono in una stessa entità, è evidente che se viceversa li separiamo in due entità è opportuno che tali entità siano correlate, cioè che esista un'associazione che le collega.



Verifiche di normalizzazione su associazioni

Consideriamo il seguente esempio:

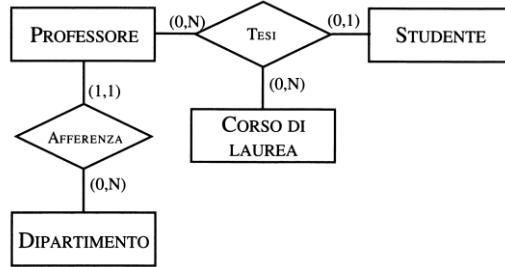


Esaminando in dettaglio l'associazione, possiamo arrivare alle seguenti conclusioni:

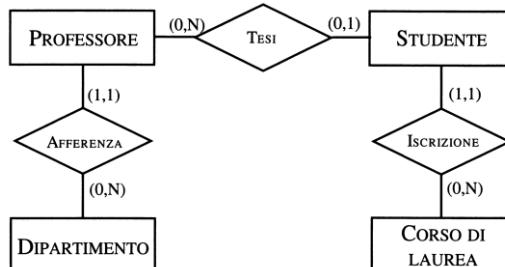
- ogni studente è iscritto a un solo corso di laurea;
- ogni studente svolge una tesi sotto la supervisione di un solo professore (che non è necessariamente legato al corso di laurea);
- ogni professore afferisce a un solo dipartimento e gli studenti sotto la sua supervisione svolgono la tesi presso tale dipartimento.

Possiamo quindi dire che le proprietà dell'applicazione di interesse sono descritte in modo esauriente dalle seguenti tre dipendenze funzionali: Studente → CorsoDiLaurea, Studente → Professore, Professore → Dipartimento.

La chiave (unica) della relazione risulta essere costituita da *Studente*: dato uno studente sono univocamente individuati il corso di laurea, il professore e il dipartimento. Ragionando come nei casi precedenti, possiamo concludere che l'associazione presenta aspetti indesiderabili e che va quindi decomposta, separando le dipendenze funzionali con primi membri diversi. In tal modo, otteniamo:



Attraverso le dipendenze possiamo notare che sarebbe opportuno decomporre ulteriormente l'associazione, ottenendo due associazioni.



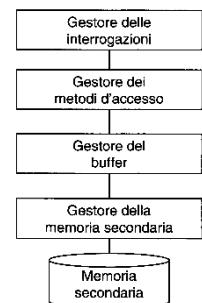
Generalizzando l'argomento appena sviluppato, possiamo arrivare alla conclusione che è opportuno decomporre le associazioni non binarie (anche già normalizzate) sulle quali sia definita qualche dipendenza il cui secondo membro contiene più di un'entità. In termini più semplici, poiché è raro incontrare associazioni che coinvolgano più di tre entità, possiamo affermare che è di solito opportuno decomporre un'associazione ternaria se su di essa è definita una dipendenza funzionale il cui primo membro è costituito da una entità e il secondo membro dalle altre due.

Organizzazione fisica e gestione delle interrogazioni

Nelle applicazioni per basi di dati, le operazioni vengono specificate in SQL e affidate a un modulo detto **gestore delle interrogazioni**, che traduce le interrogazioni in una forma interna, le trasforma al fine di renderle più efficienti (per questo il modulo viene spesso chiamato *ottimizzatore delle interrogazioni*) e le realizza in termini di operazioni di livello più basso (scansione, ordinamento, accesso diretto), che fanno riferimento alla struttura fisica dei file e sono eseguite da un modulo sottostante, chiamato **gestore dei metodi d'accesso**.

Quest'ultimo modulo, conoscendo i dettagli della struttura fisica dei file, trasforma le richieste in operazioni di lettura e scrittura di dati in memoria secondaria, che vengono però mediate da un modulo, detto **gestore del buffer**, che ha la responsabilità di mantenere temporaneamente porzioni della base di dati in memoria centrale, per favorire l'efficienza garantendo al tempo stesso l'affidabilità.

Il gestore del buffer invia poi al **gestore della memoria secondaria** le richieste effettive di lettura e scrittura fisica.



Gestore del buffer

L'interazione fra memoria centrale e memoria secondaria è realizzata nei DBMS attraverso l'utilizzo di un'apposita, grande zona di memoria centrale detta **buffer**, gestita dal DBMS in modo condiviso per tutte le applicazioni.

Il buffer è organizzato in *pagine*, che hanno dimensione pari a un numero intero di blocchi. Il gestore del buffer si occupa del caricamento e dello scaricamento (salvataggio) delle pagine dalla memoria centrale alla memoria di massa. Intuitivamente, possiamo pensare al gestore del buffer come a un modulo che riceve, dai programmi, richieste per la lettura e la scrittura di blocchi ed esegue le effettive letture o scritture sulla base di dati, secondo una tempistica che non coincide necessariamente con quella delle richieste ricevute; in particolare:

- in caso di lettura, se la pagina è già presente nel buffer, allora non è necessario effettuare la lettura fisica;
- in caso di scrittura, il gestore del buffer può decidere di differire la scrittura fisica, quando tale attesa è compatibile con le proprietà di affidabilità del sistema.

Si noti che entrambi i casi esposti consentono di ridurre il tempo di risposta di un'applicazione, evitando operazioni su memoria di massa.

Per la gestione dei buffer è necessario mantenere informazioni sull'uso delle pagine:

- un direttorio descrive il contenuto corrente del buffer indicando per ciascuna pagina quali sono il file fisico e il numero di blocco a essa corrispondente;
- per ogni pagina del buffer, il gestore mantiene alcune “variabili di stato”, fra cui un contatore, per indicare quanti programmi utilizzano la pagina, e un bit di stato, per indicare se la pagina è stata modificata (e quindi le modifiche vanno prima o poi riportate in memoria secondaria).

Un possibile insieme di operazioni fondamentali allo scopo è il seguente.

- La primitiva **fix** viene usata dalle transazioni per richiedere l'accesso a una pagina; essa restituisce al modulo chiamante il riferimento alla pagina del buffer, in modo che esso possa accedere effettivamente ai dati. Richiede una lettura solo se la pagina non è già nel buffer ed incrementa il valore del contatore associato alla pagina.
Cerca la pagina nel buffer: se c'è restituisce l'indirizzo, altrimenti cerca una pagina libera nel buffer (contatore a zero). Se la pagina viene trovata, opera lo "scambio", altrimenti si hanno due alternative: (i) "*steal*", selezione di una "vittima" i cui dati vengono scritti in memoria secondaria; (ii) "*no-steal*", l'operazione viene posta in attesa.
- La primitiva **setDirty** indica al gestore del buffer che una pagina è stata modificata; l'effetto è la modifica del bit di stato relativo.
- La primitiva **unfix** indica al gestore del buffer che il modulo chiamante ha terminato di usare la pagina; come effetto, viene decrementato il contatore di utilizzo della pagina.
- La primitiva **force** trascrive in memoria di massa, in modo sincrono, una pagina del buffer. Questa operazione viene richiesta dal gestore dell'affidabilità quando è necessario garantire che alcuni dati non vengano persi.

Gestione delle transazioni

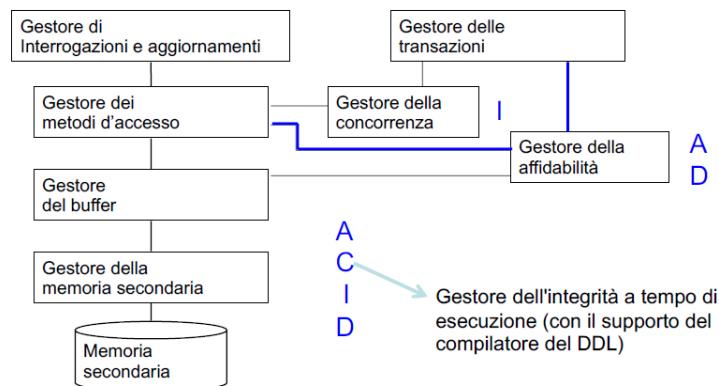
Una **transazione** è una parte di programma caratterizzata da un inizio (*begin-transaction*, *start transaction* in SQL), una fine (*end-transaction*, non esplicitata in SQL) e al cui interno deve essere eseguito una e una sola volta uno dei seguenti comandi:

1. **commit work**, per terminare correttamente;
2. **rollback work**, per abortire la transazione;

Un **sistema transazionale (OLTP)** è un sistema in grado di definire ed eseguire transazioni per conto di un certo numero di applicazioni concorrenti.

Le transazioni devono soddisfare le proprietà ACID:

- **Atomicità.** Una transazione è un'unità atomica di elaborazione: non può lasciare la base di dati in uno stato intermedio; quindi, un guasto o un errore prima del commit debbono causare l'annullamento (UNDO) delle operazioni eventualmente svolte e un guasto o un errore dopo il commit non deve avere conseguenze, ma se necessario vanno ripetute (REDO) le operazioni.
- **Consistenza.** La transazione rispetta i vincoli di integrità; quindi, se lo stato iniziale è corretto, anche lo stato finale è corretto.
- **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, una transazione non espone i suoi stati intermedi, altrimenti si potrebbe generare un "effetto domino".
- **Durabilità (persistenza).** Gli effetti di una transazione andata in commit non vanno perduti ("durano per sempre"), anche in presenza di guasti.



Controllo di affidabilità

Il controllo dell'affidabilità garantisce due proprietà fondamentali delle transazioni: l'atomicità e la persistenza. In pratica, esso garantisce che le transazioni non vengano lasciate incomplete, con alcune operazioni eseguite e le altre no, e che gli effetti di ciascuna transazione conclusa con un commit siano mantenuti in modo permanente.

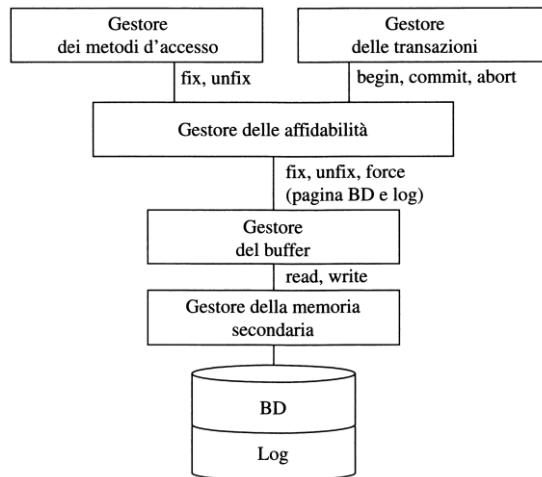
Il controllore dell'affidabilità svolge il proprio compito attraverso il **log**, un archivio persistente su cui registra le varie azioni svolte dal DBMS. Come vedremo, ogni azione di

scrittura sulla base di dati viene protetta tramite una azione sul log, in modo che sia possibile “disfare” (*undo*) le azioni a seguito di malfunzionamenti o guasti precedenti il commit, oppure “rifare” (*redo*) queste azioni qualora la loro buona riuscita sia incerta e le transazioni abbiano effettuato un commit.

Architettura del controllore dell'affidabilità

Il gestore delle affidabilità

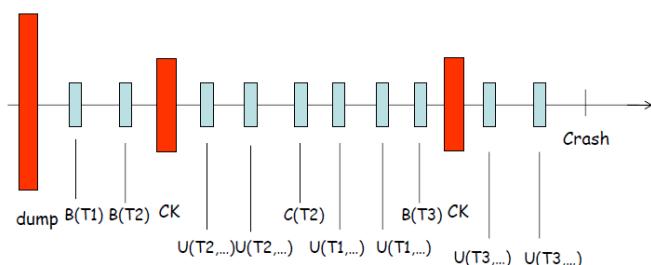
1. gestisce l'esecuzione dei comandi transazionali (*begin, end, commit, rollback*);
2. riceve le richieste di accesso alle pagine in lettura e in scrittura, che passa al gestore del buffer, e genera altre richieste di lettura e scrittura di pagine necessarie a garantire robustezza e resistenza ai guasti;
3. predisponde i dati essenziali per i meccanismi di ripristino dai guasti, in particolare azioni di *checkpoint* e di *dump*;
4. realizza le primitive di ripristino (recovery) dopo i malfunzionamenti/guasti, dette rispettivamente ripresa a caldo (*warm restart*) e ripresa a freddo (*cold restart*).



Abbiamo già visto che la memoria centrale è una memoria non persistente e che la memoria di massa è una memoria persistente, ma soggetta a danneggiamenti. Per poter operare, il controllore dell'affidabilità deve disporre di **memoria stabile**, cioè di memoria che risulti resistente ai guasti (ovviamente è un'astrazione).

Organizzazione del log

Il **log** è un file sequenziale gestito dal controllore dell'affidabilità, scritto in memoria stabile e contenente informazione ridondante che permette di ricostruire il contenuto della base di dati a seguito di guasti. Di seguito, la descrizione di un log:



I record del log sono di due tipi: di transazione e di sistema. I **record di transazione** registrano le attività svolte da ciascuna transazione, nell'ordine in cui esse vengono effettuate e troviamo:

- i record di `begin`, `commit` e `abort`, che contengono, oltre all'indicazione del tipo di record, anche l'identificativo T della transazione [$B(T)$, $C(T)$, $A(T)$];
- i record di `update`, che contengono l'identificativo T della transazione, l'identificativo O dell'oggetto su cui avviene l'update, e poi due valori BS e AS che descrivono rispettivamente il valore dell'oggetto O precedentemente alla modifica, *before state*, e successivamente alla modifica, *after state* [$U(T, O, BS, AS)$];
- i record di `insert` e di `delete` sono analoghi a quelli di `update`, da cui si differenziano per l'assenza nei primi del before state e nei secondi dell'after state [$I(T, O, AS)$ e $D(T, O, BS)$].

Questi record consentono di disfare e rifare le rispettive azioni sulla base di dati, attraverso operazioni specifiche, di competenza del gestore dell'affidabilità, chiamate *Undo* e *Redo* e realizzate nel modo seguente:

- *Undo*: per disfare un'azione su un oggetto O è sufficiente ricopiare nell'oggetto O il valore BS ; l'`insert` viene disfatto cancellando l'oggetto O ;
- *Redo*: per rifare un'azione su un oggetto O è sufficiente ricopiare nell'oggetto O il valore AS ; il `delete` viene rifatto cancellando l'oggetto O .

Dato che le primitive di *Undo* e *Redo* sono definite tramite un'azione di copiatura, vale per esse una proprietà essenziale, detta **idempotenza**, per la quale l'effettuazione di un numero arbitrario di undo e redo della stessa azione equivale allo svolgimento di tale azione una sola volta:

$$Undo(Undo(A)) = Undo(A) \quad Redo(Redo(A)) = Redo(A)$$

Questa proprietà è utile perché si potrebbero avere errori durante le operazioni di ripristino, che portano alla ripetizione delle operazioni di *Undo* e *Redo*.

I **record di sistema** indicano l'effettuazione di operazioni specifiche del controllore dell'affidabilità chiamate *dump* (abbastanza rare) e di *checkpoint* (più frequenti).

Un **checkpoint** è un'operazione che viene svolta periodicamente dal gestore dell'affidabilità (in stretto coordinamento con il buffer manager), con l'obiettivo di registrare quali transazioni sono attive. Esistono diverse versioni dell'operazione, ma vediamo la più semplice e intuitiva:

1. si sospende l'accettazione di operazioni di scrittura, `commit` o `abort`, da parte di ogni transazione;
2. si trasferiscono in memoria di massa (tramite *force*) tutte le pagine del buffer su cui sono state eseguite modifiche da parte di transazioni che hanno già effettuato il `commit`;
3. si scrive in modo sincrono (*force*) nel log un record di *checkpoint* che contiene gli identificatori delle transazioni attive;
4. si riprende l'accettazione delle operazioni sopra sospese.

Un ***dump*** è una copia incompleta e consistente della base di dati, che viene normalmente effettuata in mutua esclusione con tutte le altre transazioni quando il sistema non è operativo (per esempio, di notte oppure durante il fine settimana). La copia viene memorizzata su memoria stabile ed è anche chiamata *backup*. Dopo la conclusione dell'operazione di dump viene scritto nel log un *record di dump*, che segnala appunto la presenza di una copia fatta in un determinato istante; dopodiché il sistema può tornare al suo funzionamento normale.

Esecuzione delle transazioni e scrittura del log

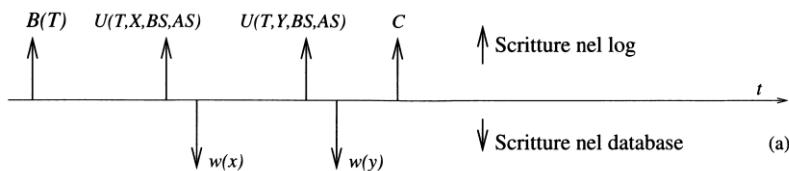
Durante il funzionamento normale delle transazioni, il controllore dell'affidabilità deve garantire che siano seguite due regole, che definiscono i requisiti minimi che consentono di ripristinare la correttezza della base di dati a fronte di guasti.

- La ***regola WAL*** (*Write Ahead Log*) impone che i record di log siano scritti prima dei corrispondenti record della base di dati. Questa regola consente di disfare le scritture già effettuate in memoria di massa da parte di una transazione che non ha ancora effettuato un commit, poiché per ogni aggiornamento viene reso disponibile in modo affidabile il valore precedente la scrittura.
- La ***regola di Commit-Precedenza*** impone che i record di log siano scritti prima della effettuazione dell'operazione di commit. Questa regola consente di rifare le scritture già decise da una transazione che ha effettuato il commit, ma le cui pagine modificate non sono ancora state trascritte dal buffer manager in memoria di massa.

La transazione sceglie, in modo atomico e indivisibile, l'esito di commit nel momento in cui scrive sul log, in modo sincrono (tramite *force*), il *record di commit*. Prima di questa azione, un eventuale guasto comporta l'*Undo* delle azioni effettuate, ricostruendo così lo stato iniziale della base di dati. Dopo l'azione di commit, un eventuale guasto comporta il *Redo* delle azioni effettuate, in modo da ricostruire con certezza lo stato finale della transazione.

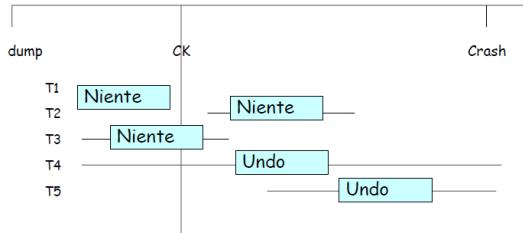
Le due regole WAL e di Commit-Precedenza impongono alcune restrizioni ai protocolli per la scrittura del log e della base di dati, ma lasciano aperte varie possibilità che vedremo a breve. Supponiamo per semplicità che le azioni svolte dalle transazioni siano *update*, ma non cambierebbe nulla nel caso di *insert* o *delete*.

Quella seguente è la **modalità immediata**.

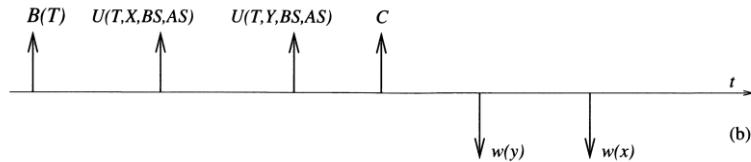


In questo schema, la transazione scrive inizialmente il record $B(T)$, quindi svolge le sue azioni di update scrivendo prima il record di log $U(T, O, BS, AS)$ e successivamente la pagina della base di dati, che così passa dal valore BS al valore AS . Tutte queste pagine sono effettivamente scritte (autonomamente dal gestore del buffer oppure con esplicite richieste di *force*) dal buffer manager prima del commit, il quale comporta una scrittura sincrona (*force*). In questo modo, al commit tutte le pagine della BD modificate dalla transazione sono certamente scritte in memoria di massa; questo schema non richiede operazioni di *Redo*.

Volendo vedere un esempio:

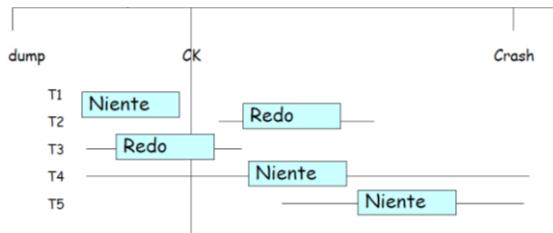


Quella seguente è invece la **modalità differita**.

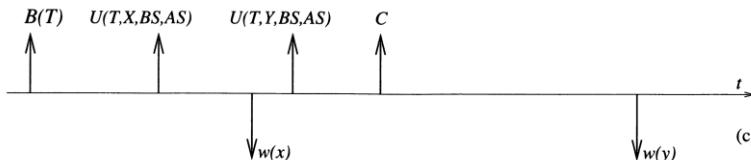


In questo schema, la scrittura dei record di log avviene prima delle scritture nella base di dati, che avvengono dopo la decisione di commit e la conseguente scrittura sincrona del record di commit sul log; questo schema non richiede operazioni di *Undo*.

Volendo vedere un esempio:

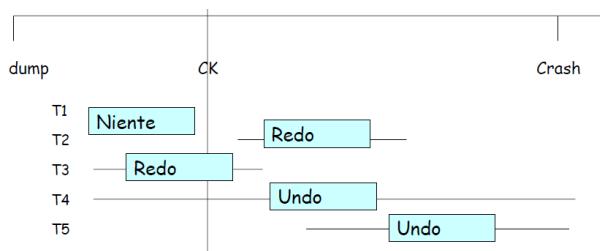


Infine, quella che segue è la **modalità mista**.



Secondo questo schema, le scritture nella base di dati, una volta protette dalle opportune scritture sul log, possono avvenire in un qualunque momento rispetto alla scrittura del record di commit sul log. Questo schema consente al buffer manager di ottimizzare le operazioni di flush relative ai suoi buffer, indipendentemente dal controllore dell'affidabilità; esso però richiede sia *Undo* che *Redo*.

Volendo vedere un esempio:



Si noti che tutti e tre i protocolli rispettano le due regole fondamentali (WAL e Commit-Precedenza) e scrivono il record di commit in modo sincrono; essi si differenziano solo per il momento in cui scrivono le pagine della base di dati.

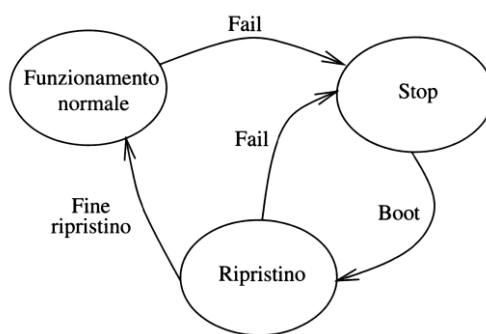
Il **costo** delle scritture del log è paragonabile al costo dell'aggiornamento della base di dati: l'uso di protocolli transazionali robusti introduce, quindi, un notevole sovraccarico per il sistema, ma è irrinunciabile per garantire le proprietà "acide" delle transazioni.

Gestione dei guasti

I guasti si suddividono in due classi:

- **guasti di sistema** ("soft"): sono guasti indotti da errori di programma, crash di sistema o cadute di tensione. Si traducono in una perdita del contenuto della memoria centrale (e quindi di tutti i buffer), mantenendo invece valido il contenuto della memoria di massa (e quindi della base di dati e del log).
- **Guasti di dispositivo** ("hard"): sono guasti relativi ai dispositivi di gestione della memoria di massa. Si traducono in una perdita del contenuto della base di dati, ma non del log.

Il modello ideale in cui ci poniamo è detto di *fail-stop*: quando il sistema individua un guasto (hard o soft), esso forza immediatamente un arresto completo delle transazioni e il successivo ripristino del corretto funzionamento del sistema operativo (*boot*). Quindi, viene attivata una procedura di ripresa, denominata **riprista a caldo** (*warm restart*) nel caso di guasto di sistema e **riprista a freddo** (*cold restart*) nel caso di guasto di dispositivo. Al termine delle procedure di ripresa, il sistema diventa nuovamente utilizzabile dalle transazioni; il buffer è completamente vuoto e può riprendere a caricare pagine della base di dati o del log.



La **riprista a caldo** è articolata in quattro fasi successive.

1. Si accede all'ultimo blocco del log, che era corrente all'istante del guasto, e si ripercorre all'indietro il log fino all'ultimo (cioè più recente) record di checkpoint;
2. Si costruiscono due insiemi, detti di *UNDO* e di *REDO*, contenenti identificativi di transazioni. Si inizializza l'insieme di *UNDO* con le transazioni attive al checkpoint e l'insieme di *REDO* con l'insieme vuoto. Si percorre poi il log in avanti, aggiungendo all'insieme di *UNDO* tutte le transazioni di cui è presente il record di begin, e spostando dall'insieme di *UNDO* all'insieme di *REDO* tutti gli identificativi delle transazioni di cui è presente il record di commit. Al termine di questa fase, gli insiemi

UNDO e *REDO* contengono rispettivamente tutti gli identificativi delle transazioni da disfare e di quelle da rifare;

3. Si ripercorre all'indietro il log disfacendo le transazioni nel set di *UNDO*, risalendo fino alla prima azione della transazione più “vecchia” nei due insiemi di *UNDO* e *REDO*;
4. Si applicano le azioni di *Redo* nell'ordine in cui sono registrate nel log. In questo modo, viene replicato esattamente il comportamento delle transazioni originali.

Vediamo un esempio di applicazione del protocollo. Si supponga che nel log vengano registrate le azioni:

$B(T1), B(T2), U(T2, O1, B1, A1), I(T1, O2, A2), B(T3), C(T1),$
 $B(T4), U(T3, O2, B3, A3), U(T4, O3, B4, A4), CK(T2, T3, T4),$
 $C(T4), B(T5), U(T3, O3, B5, A5), U(T5, O4, B6, A6), D(T3, O5, B7),$
 $A(T3), C(T5), I(T2, O6, A8).$

Successivamente, si verifica un guasto. Il protocollo opera come segue:

1. Si accede al record di checkpoint, $UNDO = \{T2, T3, T4\}$, $REDO = \{\}$;
2. Successivamente si percorre in avanti il record di log, e si aggiornano gli insiemi di *UNDO* e *REDO*:
 - (a) $C(T4)$: $UNDO = \{T2, T3\}$, $REDO = \{T4\}$
 - (b) $B(T5)$: $UNDO = \{T2, T3, T5\}$, $REDO = \{T4\}$
 - (c) $C(T5)$: $UNDO = \{T2, T3\}$, $REDO = \{T4, T5\}$
3. Successivamente, si ripercorre indietro il log fino all'azione $U(T2, O1, B1, A1)$, eseguendo le seguenti azioni di *Undo*:
 - (a) *Delete*($O6$)
 - (b) *Re-Insert*($O5 = B7$)
 - (c) $O3 = B5$
 - (d) $O2 = B3$
 - (e) $O1 = B1$
4. Infine, vengono svolte le azioni di *Redo*:
 - (a) $O3 = A4$ (nota: $A4 = B5!$)
 - (b) $O4 = A6$

Questo meccanismo garantisce l'atomicità e la persistenza delle transazioni. Per quanto concerne l'atomicità, viene garantito che le transazioni in corso all'istante del guasto lascino la base di dati nello stato iniziale oppure in quello finale; per quanto concerne la persistenza, viene garantito che le pagine nel buffer relative a transazioni completate ma non ancora trascritte in memoria di massa vengano effettivamente completate con una scrittura in memoria di massa.

Ripresa a freddo

La **ripresa a freddo** risponde a un guasto che provoca il deterioramento di una parte della base di dati; è articolata in tre fasi successive.

1. Durante la prima fase, si accede al *dump* e si ricopia selettivamente la parte deteriorata della base di dati. Si accede anche al più recente record di *dump* nel log.

2. Si ripercorre in avanti il log, applicando relativamente alla parte deteriorata della base di dati sia le azioni sulla base di dati sia le azioni di commit o abort e riportandosi così nella situazione precedente al guasto.
3. Infine, si svolge una ripresa a caldo.

Questo schema ricostruisce tutto il lavoro svolto sulla parte della base di dati soggetta al guasto e quindi, con una ripresa a caldo, garantisce la persistenza e atomicità relativamente all'istante del guasto.

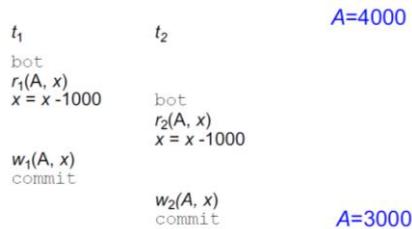
Controllo di concorrenza

Un DBMS deve spesso servire diverse applicazioni, e rispondere alle richieste provenienti da diversi utenti. Per questo motivo, è indispensabile che le transazioni di un DBMS vengano eseguite concorrentemente; è impensabile, infatti, una loro esecuzione seriale, in cui cioè le transazioni vengono eseguite una alla volta.

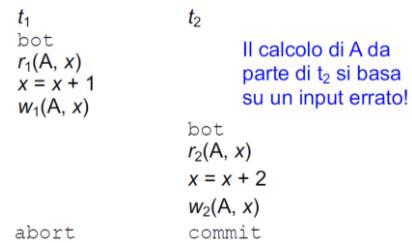
Possiamo pensare che il **controllore della concorrenza** riceva le richieste di accesso ai dati e decida se autorizzarle o meno, eventualmente riordinandole. Poiché in pratica esso stabilisce l'ordine degli accessi, il controllore della concorrenza viene anche chiamato **scheduler**.

L'esecuzione di varie transazioni può causare delle *anomalie*:

- **perdita di aggiornamento:** avviene quando si hanno due scritture sullo stesso dato. Nell'esecuzione concorrente, una delle due viene persa.



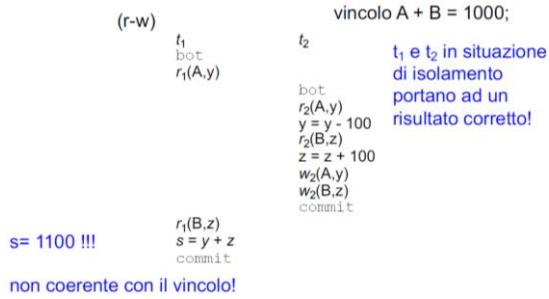
- **lettura sporca:** avviene quando una transazione legge un dato aggiornato da una transazione che successivamente andrà in abort.



- **lettura inconsistenti:** più letture dello stesso dato nel contesto della stessa transazione. Garantire che la transazione trovi esattamente lo stesso valore, non risenta dell'effetto di altre transazioni.



- **aggiornamento fantasma:** quando una transazione esegue operazioni di lettura su dati, in istanti successivi, coinvolti da un vincolo. (Nell'esempio, si trova solo B cambiato, non A)



- **inserimento fantasma:** valore aggregato da tutti gli elementi che soddisfano un predicato. Se il valore è calcolato due volte e tra il primo e il secondo calcolo viene inserito un elemento (che soddisfa il predicato) la transazione ottiene due risultati diversi!



Teoria del controllo della concorrenza

Definiamo una **transazione** come una sequenza di azioni di lettura e scrittura. Dato che le transazioni avvengono in modo concorrente, le operazioni di ingresso/uscita vengono richieste da varie transazioni in istanti successivi.

Uno **schedule** rappresenta la sequenza di operazioni di ingresso/uscita relative ad un *insieme* di transazioni; se una transazione termina prima che la successiva inizi, lo schedule è detto **seriale**.

Ci occuperemo di caratterizzare la correttezza degli schedule assumendo che le transazioni che compaiano in esse abbiano un esito noto; in questo modo, è possibile ignorare le transazioni che producono un abort, togliendo dallo schedule tutte le loro azioni, e concentrarsi solo sulle transazioni che producono un commit. Lo schedule si dice in tal caso una **commit-proiezione** dell'esecuzione delle operazioni in ingresso/uscita.

L'esecuzione di uno schedule S_i è corretta quando produce lo stesso risultato prodotto da un qualunque schedule seriale S_j delle stesse transazioni. In tal caso, diremo che S_i è **serializzabile**.

Vediamo alcuni esempi:

Schedule seriale s_1		A B		Schedule seriale s_2		A B		Schedule concorrente s_3 serializzabile		A B		Schedule s_4 non serializzabile		A B	
t_1	t_2	25	25	t_2	t_1	25	25	t_1	bot	25	25	t_1	bot	25	25
bot				bot				bot				bot			
$r_1(A,t)$				$r_2(A,s)$				$r_1(A,t)$				$r_2(A,s)$			
$t := t+100$				$s := s^2$				$t := t+100$				$s := s^2$			
$w_1(A,t)$				$w_2(A,s)$				$w_1(A,t)$				$w_2(A,s)$			
$r_1(B,t)$				$r_2(B,s)$				$r_1(B,t)$				$r_2(B,s)$			
$t := t+100$				$s := s^2$				$t := t+100$				$s := s^2$			
$w_1(B,t)$				$w_2(B,s)$				$w_1(B,t)$				$w_2(B,s)$			
commit				commit				commit				commit			
bot				bot				bot				bot			
$r_2(A,s)$				$r_1(A,t)$				$r_2(A,s)$				$r_1(A,t)$			
$s := s^2$				$t := t+100$				$s := s^2$				$t := t+100$			
$w_2(A,s)$				$w_1(A,t)$				$w_2(A,s)$				$w_1(A,t)$			
$r_2(B,s)$				$r_1(B,t)$				$r_1(B,t)$				$r_2(B,s)$			
$s := s^2$				$t := t+100$				$t := t+100$				$s := s^2$			
$w_2(B,s)$				$w_1(B,t)$				$w_1(B,t)$				$w_2(B,s)$			
commit				commit				commit				commit			

Per formalizzare che cosa si intende con la frase “due schedule producono lo stesso risultato” è necessario disporre di una definizione di equivalenza fra schedule.

View-equivalenza

Definiamo dapprima una relazione che lega coppie di operazioni di lettura e scrittura: un’operazione di lettura $r_i(x)$ **legge-da** una scrittura $w_j(x)$ quando $w_j(x)$ precede $r_i(x)$ e non vi è alcun $w_k(x)$ compreso tra le due operazioni. Un’operazione di scrittura $w_i(x)$ viene detta **scrittura finale** se è l’ultima scrittura dell’oggetto x che appare nello schedule.

Due schedule vengono detti **view-equivalenti** ($S_i \approx_v S_j$) se possiedono la stessa relazione “legge-da” e le stesse scritture finali. Uno schedule viene detto **view-serializzabile** se esiste uno schedule seriale view-equivalente ad esso. Indichiamo con **VSR** l’insieme degli schedule view-serializzabili.

Vediamo degli esempi. Si considerino gli schedule seguenti:

$$\begin{aligned} S_3 : & w_0(x) r_2(x) r_1(x) w_2(x) w_2(z) \\ S_4 : & w_0(x) r_1(x) r_2(x) w_2(x) w_2(z) \\ S_5 : & w_0(x) r_1(x) w_1(x) r_2(x) w_1(z) \\ S_6 : & w_0(x) r_1(x) w_1(x) w_1(z) r_2(x) \end{aligned}$$

S_3 è view-equivalente allo schedule seriale S_4 (quindi è view-serializzabile). Infatti, $LD_3 = \{w_0(x)r_2(x), w_0(x)r_1(x)\}$, $SF_3 = \{w_2(x), w_2(z)\}$ e $LD_4 = \{w_0(x)r_1(x), w_0(x)r_2(x)\}$, $SF_4 = \{w_2(x), w_2(z)\}$. S_5 non è invece view-equivalente a S_4 , ma è view-equivalente allo schedule seriale S_6 , e quindi risulta anch’esso view-serializzabile.

Notiamo che i seguenti schedule, corrispondenti alle anomalie di perdita di aggiornamento, di letture inconsistenti e di aggiornamento fantasma, non sono view-serializzabili:

$$\begin{aligned} S_7 : & r_1(x) r_2(x) w_2(x) w_1(x) \\ S_8 : & r_1(x) r_2(x) w_2(x) r_1(x) \\ S_9 : & r_1(x) r_1(y) r_2(z) r_2(y) w_2(y) w_2(z) r_1(z) \end{aligned}$$

Determinare la view-equivalenza di due schedule è un problema con complessità lineare. Determinare se uno schedule è view-equivalente a un qualsiasi schedule seriale è però un problema NP-difficile, perché può esistere un numero esponenziale di schedule seriali (tutte le permutazioni delle transazioni) con cui confrontare lo schedule dato, senza alcun indizio per scegliere quelli con cui confrontarlo. Questo risultato sulla complessità illustra che il concetto di view-equivalenza non può essere usato al fine di caratterizzare la serializzabilità.

Si preferisce quindi definire una condizione di equivalenza più ristretta, la quale non copra tutti i casi di equivalenza tra schedule, ma sia utilizzabile nella pratica, presentando una complessità inferiore.

Conflict-equivalenza

Date due azioni a_i e a_j , con $i \neq j$, si dice che a_i è in **conflitto** con a_j se esse operano sullo stesso oggetto e almeno una di esse è una scrittura. Possono quindi esistere conflitti lettura-scrittura (*rw* o *wr*) e conflitti scrittura-scrittura (*ww*).

Si dice che lo schedule S_i è **conflict-equivalente** allo schedule S_j ($S_i \approx_c S_j$) se i due schedule presentano le stesse operazioni e ogni coppia di operazioni in conflitto è nello stesso ordine nei due schedule. Uno schedule risulta quindi **conflict-serializzabile** se esiste uno schedule seriale a esso conflict-equivalente. Chiamiamo **CSR** l'insieme degli schedule conflict-serializzabili.

Vediamo un esempio di schedule S conflict-serializable (in quanto troveremo uno schedule seriale). Durante lo svolgimento, ci fermiamo alle prime due azioni che non sono in conflitto ($w_2(A)r_1(B)$) e le scambiamo (e così via).

$$S = r_1(A) w_1(A) r_2(A) w_2(A) r_1(B) w_1(B) r_2(B) w_2(B)$$

$r_1(A) w_1(A) r_2(A) w_2(A) r_1(B) w_1(B) r_2(B) w_2(B)$
 $r_1(A) w_1(A) r_2(A) r_1(B) w_2(A) w_1(B) r_2(B) w_2(B)$
 $r_1(A) w_1(A) r_1(B) r_2(A) w_2(A) w_1(B) r_2(B) w_2(B)$
 $r_1(A) w_1(A) r_1(B) r_2(A) w_1(B) w_2(A) r_2(B) w_2(B)$
 $r_1(A) w_1(A) r_1(B) w_1(B) r_2(A) w_2(A) r_2(B) w_2(B)$

È possibile dimostrare che la classe degli schedule CSR è strettamente inclusa in quella degli schedule VSR: esistono cioè schedule che appartengono a VSR ma non a CSR, mentre tutti gli schedule CSR appartengono a VSR. Quindi, la conflict-serializzabilità è condizione sufficiente, ma non necessaria, per la view-serializzabilità.

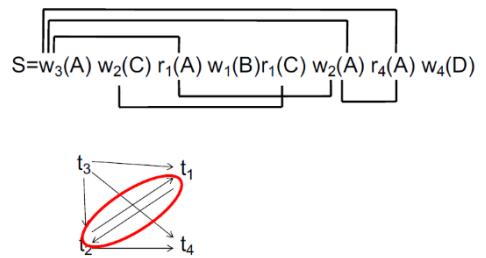
La verifica di conflict-serializzabilità avviene per mezzo del **grafo dei conflitti**, in cui troviamo:

- 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 dei conflitti è aciclico. Ad esempio, lo schedule di fianco non è in CSR.

La tecnica di CSR è inutilizzabile in pratica: sarebbe efficiente se potessimo conoscere il grado dall'inizio, ma così non è in quanto uno scheduler deve operare “incrementalmente”, cioè ad ogni richiesta di operazione decidere se eseguirla subito oppure fare altro. Inoltre, la tecnica si basa sull'ipotesi della commit-proiezione.

In pratica, si utilizzano tecniche che garantiscono la CSR senza dover costruire il grafo e che non richiedono l'ipotesi della commit-proiezione.



Controllo di concorrenza basato su locks

L'idea alla base della tecnica del **locking** è quella di proteggere le operazioni di lettura e scrittura tramite le esecuzioni di opportune primitive (*r_lock*, *w_lock* e *unlock*); lo scheduler (che viene in questo caso detto anche **lock manager**) riceve una sequenza di richieste di esecuzione di queste primitive da parte delle transazioni, e ne determina la correttezza con una semplice *ispezione* di un'opportuna struttura dati.

Nell'esecuzione di operazioni di lettura e scrittura si devono rispettare i seguenti vincoli:

1. ogni operazione di lettura deve essere preceduta da un *r_lock* e seguita da un *unlock*; il lock si dice in questo caso **condiviso**, perché su un dato possono essere contemporaneamente attivi più lock di questo tipo;
2. ogni operazione di scrittura deve essere preceduta da un *w_lock* e seguita da un *unlock*; il lock si dice in tal caso **esclusivo**, perché non può coesistere con altri lock (esclusivi o condivisi) sullo stesso dato.

Quando una transazione segue queste regole si dice **ben formata rispetto al locking**; si noti che le regole di precedenza illustrate non sono strette, e quindi l'operazione di lock di una risorsa può avvenire molto prima di un'azione di lettura o scrittura di quella risorsa. Se una transazione deve contemporaneamente leggere e scrivere, la transazione può richiedere solo un lock di tipo esclusivo, oppure passare al momento opportuno da un lock condiviso a uno esclusivo, “incrementando” il livello di lock (**lock upgrade**).

Quando una richiesta di lock è concessa, si dice che la corrispondente risorsa viene *acquisita* dalla transazione richiedente; all'atto dell'*unlock*, la risorsa viene *rilasciata*. Quando una richiesta di lock non viene concessa, la transazione richiedente viene messa in *stato di attesa*; l'attesa termina quando la risorsa viene sbloccata e diviene disponibile. I lock già concessi vengono memorizzati in **tabelle di lock**, gestite dal lock manager.

Ogni richiesta di lock che perviene al lock manager è caratterizzata solo dall'identificativo della transazione che fa la richiesta, e dalla risorsa per la quale la richiesta viene effettuata. La politica che viene seguita dal lock manager per concedere i lock è rappresentata nella **tabella dei conflitti**, in cui le righe identificano le richieste, le colonne lo stato della risorsa richiesta, il primo valore nella cella l'esito della richiesta e il secondo valore nella cella lo stato che verrà assunto dalla risorsa dopo l'esecuzione della primitiva.

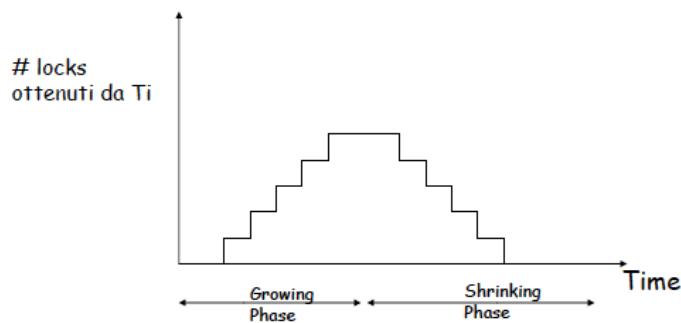
Richiesta	Stato risorsa		
	<i>libero</i>	<i>r_locked</i>	<i>w_locked</i>
<i>r_lock</i>	OK / <i>r_locked</i>	OK / <i>r_locked</i>	No / <i>w_locked</i>
<i>w_lock</i>	OK / <i>w_locked</i>	No / <i>r_locked</i>	No / <i>w_locked</i>
<i>unlock</i>	<i>error</i>	OK / <i>dipende</i>	OK / <i>libero</i>

I tre *No* presenti nella tabella rappresentano i conflitti che si possono presentare, quando si richiede una lettura o una scrittura su un oggetto già bloccato in scrittura, o una scrittura su un oggetto già bloccato in lettura. In pratica, solo quando un oggetto è bloccato in lettura è possibile dare risposta positiva a un'altra richiesta di lock in lettura. Nel caso di unlock di una risorsa bloccata in modo condiviso, la risorsa ritorna *libera* quando non ci sono altre

transazioni in lettura che operano su di essa; altrimenti, essa rimane bloccata in lettura. Per questo motivo, la corrispondente casella della matrice dei conflitti assume il valore *dipende*. Per tener conto del numero di lettori è necessario introdurre un contatore, che viene incrementato a ogni richiesta di *r_lock* concessa, e decrementato a ogni *unlock*.

Per avere la garanzia che le transazioni seguano uno schedule serializzabile è necessario porre una restrizione sull'ordinamento delle richieste di lock, che prende il nome di principio del ***locking a due fasi*** (2PL): una transazione, dopo aver rilasciato un lock, non può acquisirne altri.

Come conseguenza di tale principio si possono distinguere nell'esecuzione della transazione due diverse fasi: una prima fase in cui si acquisiscono i lock per le risorse cui si deve accedere (**fase crescente**), e una seconda fase in cui i lock acquisiti vengono rilasciati (**fase calante**).



Un sistema in cui le transazioni sono ben formate rispetto al locking (ovvero richiedono sempre un opportuno lock prima di accedere alle risorse e lo rilasciano prima del termine della transazione), con un lock manager che rispetta la politica descritta nella tabella, e in cui le transazioni seguono il principio del lock a due fasi, è un sistema transazionale caratterizzato dalla serializzabilità delle proprie transazioni. La classe 2PL contiene gli schedule che soddisfano queste condizioni.

Ogni schedule che rispetti i requisiti del protocollo di lock a due fasi risulta anche uno schedule serializzabile rispetto alla conflict-equivalenza, ovvero **la classe 2PL è strettamente contenuta nella classe CSR** (uno schedule può quindi essere in CSR e non in 2PL).

2PL risolve le anomalie di perdita di aggiornamento, aggiornamento fantasma e letture inconsistenti, ma non risolve quelle di lettura sporca e d'inserimento fantasma.

Si può procedere attraverso una restrizione del protocollo 2PL, che porta al cosiddetto **2PL stretto** (*strict 2PL*). Nel ***locking a due fasi stretto***, i lock di una transazione possono essere rilasciati solo dopo aver correttamente effettuato le operazioni di commit/abort.

In pratica, con questo vincolo i lock vengono rilasciati solo al termine della transazione, dopo che ciascun dato è stato portato nel suo stato finale. Tramite il 2PL stretto viene reso impossibile il verificarsi di letture sporche, perché viene impedito l'accesso (da parte di altre transazioni) a dati scritti da transazioni che ancora non hanno effettuato il commit, e si supera la necessità dell'ipotesi di commit-proiezione.

L'anomalia di inserimento fantasma viene risolta con il **lock di predicato**, in cui impediamo anche la scrittura di nuovi oggetti che soddisfino il predicato.

Concludiamo osservando come nel contesto del 2PL possono essere realizzati i diversi livelli di isolamento (per le transazioni di sola lettura):

- **read uncommitted**: la transazione non richiede lock e non osserva nemmeno i lock esclusivi posti da altre transazioni;
- **read committed**: richiede lock per le letture, rilasciandoli subito dopo, quindi senza 2PL; in questo modo si evitano le letture sporche, ma non altre anomalie tipiche delle letture;
- **repeatable read**: applica in 2PL stretto, ma applicando i lock a singole tuple; sono evitate tutte le anomalie, ma non l'inserimento fantasma (phantom), perché non è possibile impedire l'inserimento di nuove tuple;
- **serializable**: applica il 2PL stretto e i lock di predicato e quindi evita tutte le anomalie.

Controllo di concorrenza basato su timestamp

Questo metodo utilizza i **timestamp**, cioè identificatori associati ad ogni evento temporale che definiscono un ordinamento totale sugli eventi. Il controllo di concorrenza mediante timestamp (**metodo TS**) avviene nel seguente modo:

- a ogni transazione si associa un timestamp che rappresenta il momento di inizio della trasmissione;
- si accetta uno schedule solo se esso riflette l'ordinamento seriale delle transazioni in base al valore del timestamp di ciascuna transazione.

Le transazioni sono quindi naturalmente ordinate secondo l'ordine di arrivo ed eseguono liberamente. Ad ogni operazione r/w, lo scheduler controlla che i timestamps delle transazioni coinvolte non violino l'ordinamento seriale: se la violano, le uccide.

A ogni oggetto x vengono associati due indicatori, $WTM(x)$ e $RTM(x)$, che sono rispettivamente i timestamp della transazione che ha eseguito l'ultima scrittura e della transazione con t più grande che ha letto x . Allo scheduler arrivano le richieste di accesso agli oggetti del tipo $r_t(x)$ o $w_t(x)$, dove t rappresenta il timestamp della transazione che esegue la lettura o la scrittura. Lo scheduler non fa altro che permettere o no l'operazione, secondo la seguente politica:

- $read(x, ts)$: se $ts < WTM(x)$ la transazione viene uccisa, altrimenti la richiesta viene accettata; in tal caso $RTM(x) = \max\{RTM(x), ts\}$;
- $write(x, ts)$: se $ts < WTM(x)$ o $ts < RTM(x)$ la transazione viene uccisa, altrimenti la richiesta viene accettata; in tal caso $WTM(x) = ts$.

In pratica, ogni transazione non può leggere o scrivere un dato scritto da una transazione con timestamp superiore, e non può scrivere su un dato che è già stato letto da una transazione con timestamp superiore.

Vediamo un esempio. Si supponga che sia $RTM(x) = 7$ e $WTM(x) = 5$ (ovvero l'oggetto x è stato letto da transazioni con timestamp 7 o minore e scritto l'ultima volta dalla transazione con timestamp 5). Nel seguito, descriviamo la risposta dello scheduler alle richieste di lettura e scrittura ricevute:

Richieste	Risposte	Nuovi valori
$r_6(x)$	ok	
$r_7(x)$	ok	
$r_9(x)$	ok	$RTM(x) = 9$
$w_8(x)$	no	t_8 uccisa
$w_{11}(x)$	ok	$WTM(x) = 11$
$r_{10}(x)$	no	t_{10} uccisa

Il metodo TS comporta l'uccisione di un gran numero di transazioni; inoltre, questa versione del metodo è corretta sotto l'ipotesi di uso di commit-proiezioni. Per rimuovere questa ipotesi è necessario “bufferizzare” le scritture, cioè effettuarle in memoria e trascriverle in memoria di massa solo dopo il commit; ciò comporta che le letture da parte di altre transazioni dei dati bufferizzati nel buffer e in attesa di commit vengano a loro volta messe in attesa del

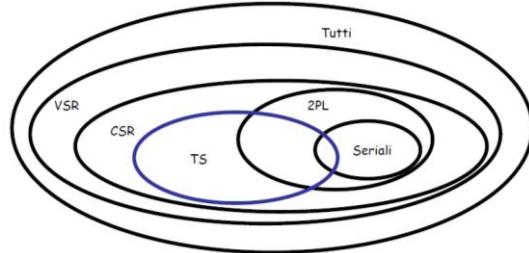
commit della transazione scrivente, in pratica introducendo meccanismi di attesa analoghi a quelli di locking.

Una modifica del metodo è l'uso delle ***multiversioni***, che consiste nel mantenere diverse copie degli oggetti della base di dati, per ogni transazione che modifica la base di dati. Ogni volta che una transazione scrive un oggetto, la vecchia copia non viene persa, ma una nuova N -esima copia viene creata, con un corrispondente $WTM_N(x)$. Si ha invece un solo $RTM(x)$ globale. Quindi, in un generico istante sono attive $N \geq 1$ copie di ciascun oggetto x ; con questo metodo, le richieste di lettura non vengono mai rifiutate, ma vengono dirette alla versione dei dati corretta rispetto al timestamp della transazione richiedente. Le copie vengono rilasciate quando sono divenute inutili, in quanto non esistono più transazioni in lettura interessate al loro valore. Le regole di comportamento diventano:

- $read(x, ts)$: una lettura è sempre accettata. Si legge un x_k siffatto: se $ts > WTM_N(x)$, allora $k = N$, altrimenti si prende i in modo che sia $WTM_i(x) < ts < WTM_{i+1}(x)$;
- $write(x, ts)$: se $ts < RTM(x)$ o $ts < WTM_N(x)$ la transazione viene uccisa, altrimenti si aggiunge una nuova versione del dato (N cresce di uno) con $WTM_{N+1}(x) = ts$.

Le tecniche 2PL e TS sono incomparabili! Infatti, ad esempio,

- Schedule in TS ma non in 2PL
 $r_1(x) w_2(x) r_3(x) r_1(y) w_2(y)$
- Schedule in 2PL ma non in TS
 $r_2(x) w_2(x) r_1(x) w_1(x)$
- Schedule in TS e in 2PL
 $r_1(x) r_2(y) w_2(y) w_1(x) r_2(x) w_2(x)$



Volendole comparare, abbiamo

	2PL	Metodo TS
Transazioni Rifiutate	Attesa	Uccise e riavviate
Ordine	Imposto dai conflitti	Imposto da TS
Attesa esito	Incremento tempo di blocco	Condizioni di attesa
Problemi	Deadlock	Restart molto lento (> tempo attesa 2PL)

Timestamp è superiore in genere quando le transazioni sono read only o è raro che transazioni concorrenti leggano e scrivano lo stesso elemento. Il locking è superiore nelle situazioni di alto conflitto perché certamente il locking ritarda le transazioni quando sono in attesa di lock e può anche portare a rollback in caso di deadlock, ma la probabilità di rollback è molto superiore nel caso di timestamp, dando luogo a ritardi medi molto superiori che nel locking.

Lock management

Un **lock manager** (LM) è un processo in grado di essere invocato da tutti i processi che intendono accedere alla base di dati. I processi per accedere alle risorse dovranno eseguire delle procedure di:

$$\begin{aligned}r_lock(T, x, \textit{errcode}, \textit{timeout}) \\w_lock(T, x, \textit{errcode}, \textit{timeout}) \\unlock(T, x)\end{aligned}$$

- T rappresenta l'identificativo della transazione;
- x l'elemento per il quale si richiede o si rilascia il lock;
- $\textit{errcode}$ rappresenta un valore restituito dal lock manager e vale zero qualora la richiesta sia stata soddisfatta, mentre assume un valore diverso da zero qualora la richiesta non sia stata soddisfatta;
- $\textit{timeout}$ rappresenta l'intervallo di massimo di tempo che la procedura chiamante è disposta ad aspettare per ottenere il lock sulla risorsa.

Quando la richiesta non può essere immediatamente soddisfatta, il sistema inserisce il processo richiedente in una coda associata alla risorsa; ciò comporta un'attesa arbitrariamente lunga, e quindi il processo associato alla transazione viene sospeso. Appena una risorsa viene rilasciata, il lock manager controlla se esistono dei processi in attesa della risorsa e nel caso prende il primo processo della coda e concede a esso la risorsa.

Quando scatta un timeout e la richiesta è insoddisfatta, la transazione richiedente può eseguire un *rollback*, cui generalmente seguirà una ripartenza della stessa transazione, oppure decidere di proseguire, richiedendo nuovamente il lock, in quanto un fallimento nella richiesta di lock non comporta un rilascio delle altre risorse acquisite dalla transazione in precedenza.