1 Introduzione

Sistemi Informativi, informazioni e dati

Nello svolgimento di ogni attività sono necessari la disponibilità di informazioni e la capacità di gestirle in modo efficace; ogni organizzazione è dotata di un sistema informativo, che organizza e gestisce le informazioni necessarie per perseguire gli scopi dell'organizzazione stessa. Quasi sempre il sistema informativo è di supporto ad altri sottosistemi, più o meno integrati.
Per indicare la porzione automatizzata del sistema informativo viene di solito utilizzato il termine "sistema informatico". Nei sistemi informatici, per ragioni che in parte sono tecnologiche e in parte sono legate alla semplicità dei meccanismi di gestione, il concetto di rappresentazione e codifica viene portato all'estremo: le informazioni sono rappresentate per mezzo di dati, che hanno bisogno di essere interpretati per fornire informazioni.
Possiamo dire che i dati da soli non hanno alcun significato, ma, una volta interpretati e correlati opportunamente, essi forniscono informazioni.
Informazione: notizia, dato o elemento che consente di avere conoscenza più o meno esatta di fatti, situazioni, modi di essere;
Dato: ciò che è immediatamente presente alla conoscenza, prima di ogni elaborazione; elementi di informazione costituiti da simboli che debbono essere elaborati.
Una base di dati è una collezione di dati, utilizzati per rappresentare le informazioni di interesse per un sistema informativo. I dati hanno caratteristiche più stabili rispetto a quelle delle procedure che operano su di essi. (I dati relativi alle applicazioni bancarie hanno una struttura sostanzialmente invariata da decenni, mentre le procedure che agiscono su di essi variano con una certa frequenza).

Basi di dati e sistemi di gestione di basi di dati

In assenza di un software specifico, la gestione dei dati è affidata ai linguaggi di programmazione tradizionali. L'approccio "convenzionale" alla gestione dei dati sfrutta la presenza di archivi o file per memorizzare i dati in modo persistente sulla memoria di massa. Le procedure scritte in un linguaggio di programmazione sono completamente autonome; eventuali dati di interesse per più programmi sono replicati tante volte quanti sono i programmi che li utilizzano, con evidente ridondanza e possibilità di incoerenza. Le basi di dati sono state concepite in buona misura per superare questo tipo di inconvenienti.
Un sistema di gestione di basi di dati (Data Base Management Syste, DBMS) è un sistema software in grado di gestire collezioni di dati che siano grandi, condivise, e persistenti, assicurando la loro affidabilità e privatezza.

Grandi

Nel senso che possono avere anche dimensioni enormi, prevedendo una gestione di dati in memoria secondaria.

Condivise

Applicazioni e utenti diversi debbono poter accedere a dati comuni. In questo modo si riduce la ridondanza dei dati poiché si evitano ripetizioni, e conseguentemente si riduce la possibilità di inconsistenze. Per garantire l'accesso condiviso il DBMS dispone di un meccanismo apposito chiamato controllo di concorrenza.

Persistenti

Hanno tempo di vita non limitato all'esecuzione dei programmi.

Affidabilità

Capacità del sistema di conservare il contenuto della base di dati in caso di malfunzionamenti Hardware o Software; i DBMS forniscono per questo scopo funzionalità di salvataggio e ripristino (backup e recovery).

Privatezza
Ciascun utente, riconosciuto in base ad un nome d'utente che viene abilitato a svolgere solo determinate azioni sui dati, attraverso meccanismi di autorizzazione.

Efficienza
Capacità di svolgere le operazioni utilizzando un insieme di risorse che sia accettabile per gli utenti.

Efficacia
Capacità della base di dati di rendere produttive, in ogni senso, le attività dei suoi utenti.

Database vs File system
La gestione di insiemi di dati grandi e persistenti è possibile anche attraverso sistemi più semplici quali gli ordinari file system dei sistemi operativi, che permettono di realizzare anche rudimentali forme di condivisione. I DBMS estendono le funzionalità dei file system, fornendo più servizi ed in maniera più integrata. I file system prevedono forme di condivisione, permettendo accessi contemporanei in lettura ed esclusivi in scrittura. Nei FBMS c'è maggiore flessibilità: si può accedere contemporaneamente a record diversi di uno stesso file o addirittura allo stesso record.
Nei programmi che accedono ai file, ogni programma contiene una descrizione della struttura del file stesso, con i conseguenti rischi di incoerenza fra le descrizioni e i file stessi.
Nei DBMS, esiste una porzione della base di dati che contiene una descrizione centralizzata dei dati, che può essere utilizzata dai vari programmi.

Modelli dei dati
Un modello dei dati è un insieme di concetti utilizzati per organizzare i dati di interesse e descriverne la struttura in modo che essa risulti comprensibile per un elaboratore. Ogni modello dei dati fornisce meccanismi di strutturazione, che permettono di definire nuovi tipi sulla base di tipo predefiniti e costruttori di tipo.
Il modello relazionale dei dati, attualmente il più diffuso, permette di definire nuovi tipo per mezzo del costruttore relazione, che consente di organizzare i dati in insiemi di record a struttura fissa.

DOCENZA

Corso	*Docente*
Basi di dati	Rossi
Impianti	Neri
Linguaggi	Verdi

Esempio di base di dati relazionale.

Oltre al modello relazionale sono stati definiti altri tre tipi di modelli:
- il modello gerarchico, basato sull'uso di strutture ad albero;
- Il modello reticolare, basato sull'uso di grafi, sviluppato successivamente al modello gerarchico;
- il modello a oggetti, sviluppato negli anni Ottanta come evoluzione del modello relazionale, che estende alle basi di dati il paradigma di programmazione a oggetti.

Questi modelli sono detti modelli logici per sottolineare il fatto che le strutture utilizzate da questi modelli riflettono una particolare organizzazione (ad alberi, a grafi, a tabelle o a oggetti). Più recentemente sono stati introdotti altri modelli di dati, detti *concettuali*, utilizzati per descrivere i dati in maniera completamente indipendente dalla scelta del modello logico.

Schemi e istanze
Nelle basi di dati esiste una parte sostanzialmente invariata nel tempo, detto *schema* della base di dati, costituita dalle caratteristiche dei dati, e una parte variabile nel tempo, detta *istanza* o *stato* della base di dati, costituita dai valori effettivi.
Lo schema di una relazione è costituito dalla sua intestazione, cioè dal nome della relazione seguito dai nomi dei suoi attributi; ad esempio:

DOCENZA(Corso, NomeDocente)

L'istanza di una relazione è costituita dall'insieme, variante nel tempo, delle sue righe; nell'esempio abbiamo le tre coppie

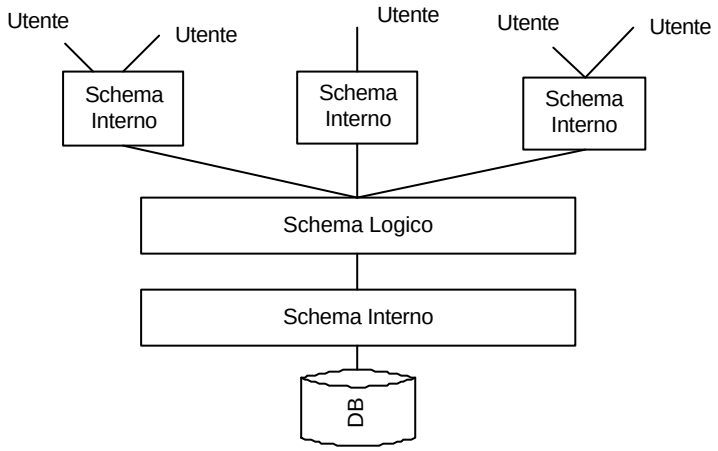
Basi di dati	Rossi
Impianti	Neri
Linguaggi	Verdi

Si dice anche che lo schema è la componente intensionale della base di dati e l'istanza la componente estensionale.

Livelli di astrazione nei DBMS

Esiste una proposta di architettura standardizzata per DBMS articolata su tre livelli, detti rispettivamente *esterno, logico* e *interno*; per ciascun livello esiste uno schema.

- Lo schema logico costituisce una descrizione dell'intera base di dati per mezzo del modello logico adottato dal DBMS;
- Lo schema interno costituisce la rappresentazione dello schema logico per mezzo di strutture fisiche di memorizzazione;
- Uno schema esterno costituisce la descrizione di una porzione della base di dati di interesse, per mezzo del modello logico.



Nei sistemi più moderni il livello esterno non è esplicitamente presente, ma è possibile definire relazioni derivate dette viste.

Indipendenza dei dati

L'architettura a livelli garantisce l'indipendenza dei dati, la principale proprietà dei DBMS. Questa proprietà consente a utenti e programmi applicativi che utilizzano una base di dati di interagire ad un elevato livello di astrazione.

L'indipendenza dei dati può essere caratterizzata ulteriormente come indipendenza fisica e logica.

- L'*indipendenza fisica* consente di interagire con il DBMS in modo indipendente dalla struttura fisica dei dati.
- L'*indipendenza logica* consente di interagire con il livello esterno della base di dati in modo indipendente dal livello logico. Ad esempio è possibile aggiungere ad uno schema esterno in base alle esigenze di un nuovo utente oppure modificare uno schema esterno senza dover modificare lo schema logico e perciò la sottostante organizzazione fisica dei dati.

E' importante sottolineare che gli accessi alla base di dati avvengono solo attraverso il livello esterno; è il DBMS che traduce le operazioni in termini dei livelli sottostanti.

Linguaggi e utenti delle basi di dati

I DBMS sono caratterizzati, da un lato, dalla presenza di molteplici linguaggi per la gestione dei dati; dall'altro dalla presenza di molteplici tipologie di utenti.

Linguaggi per basi di dati

I linguaggi per basi di dati si distinguono in due categorie:

- *linguaggi di definizione dei dati* (*Data Definition Language* – DDL), utilizzati per definire gli schemi logici, esterni e fisici e le autorizzazioni per l'accesso;
- *linguaggi di manipolazione dei dati* (*Data Manipulation language* – DML), utilizzati per l'interrogazione e l'aggiornamento delle istanze di basi di dati.

L'accesso ai dati può essere effettuato con varie modalità:

- tramite linguaggi testuali interattivi, ad esempio il linguaggio SQL;
- tramite comandi simili a quelli interattivi immersi in linguaggi di programmazione tradizionali, detti *linguaggi ospiti*;
- tramite comandi simili a quelli interattivi immersi in linguaggi di sviluppo *ad hoc*;
- tramite interfacce amichevoli che permettono di sintentizzare interrogazioni senza usare un linguaggio testuale.

Utenti e progettisti

Varie categorie di persone posso interagire con una base di dati o con un DBNS, tra cui:

- L'*amministratore della base di dati (Database Administrator DBA)* è la persona responsabile della progettazione, controllo e amministrazione. E' responsabile in particolare di garantire sufficienti prestazioni, di assicurare la affidabilità del sistema, di gestire le autorizzazioni di accesso ai dati;
- I *progettisti e programmatori di applicazioni* definiscono e realizzano i programmi che accedono alla base di dati;
- Gli *utenti* utilizzano la base di dati per le proprie attività. Essi possono essere suddivisi in due categorie: *utenti finali*, che utilizzano transazioni, *utenti casuali*, in grado di impiegare i linguaggi interattivi per l'accesso alla base di dati, formulando interrogazioni di tipo vario.

Vantaggi e svantaggi dei DBMS

Concludiamo riassumendo le caratteristiche essenziali delle basi di dati e dei DBMS e i relativi vantaggi e svantaggi.

Vantaggi:

- I DBMS permettono di considerare i dati come una risorsa comune di una organizzazione, a disposizione di tutte le sue componenti;
- La base di dati fornisce un modello unificato e preciso della parte del mondo reale di interesse dell'organizzazione;
- è possibile un controllo centralizzato dei dati, arricchito da forme di standardizzazione e beneficiando di “economie di scala”;
- La condivisione permette di ridurre ridondanza e inconsistenze;
- L'indipendenza dei dati, favorisce lo sviluppo di applicazioni più flessibili e facilmente modificabili.

L'uso dei DBMS comporta anche alcuni aspetti negativi, o almeno delicati, fra i quali i seguenti:

- i DBMS sono prodotti costosi, complessi e abbastanza diversi da molti altri strumenti informatici. La loro introduzione comporta notevoli investimenti sia diretti che indiretti;
- i DBMS forniscono una serie di servizi che sono necessariamente associati ad un costo.

2 Il modello relazionale

La maggior parte dei sistemi di basi di dati oggi sul mercato si fonda sul modello relazionale, che fu proposto in una pubblicazione scientifica, nel 1970, al fine di superare le limitazioni dei modelli all'epoca utilizzati a livello logico. L'affermazione del modello relazionale stata abbastanza lenta, a causa proprio dell'alto livello di astrazione.

Il modello relazionale: strutture

Modelli logici nei sistemi di basi di dati

Il modello relazionale si basa su due concetti, *relazione* e *tabella*, di natura diversa ma facilmente riconducibili l'uno all'altro. La nozione di *relazione* proviene dalla matematica, in particolare dalla teoria degli insiemi, mentre il concetto di *tabella* è semplice e intuitivo e risulta naturale e comprensibile anche per gli utenti finali.
Il modello relazione risponde al requisito dell'indipendenza dei dati, prevede una distinzione, nella descrizione dei dati, fra il livello logico e il livello fisico: gli utenti che accedono ai dati e i programmatori che sviluppano le applicazioni fanno riferimento al solo livello logico; per accedere ai dati non è necessario conoscere le strutture fisiche.

Relazioni e tabelle

Ricordiamo che, in matematica, dati due insiemi D_1 e D_2, si chiama prodotto cartesiano di D_1 e D_2, in simboli D_1 x D_2, l'insieme delle coppie ordinate (v_1, v_2) tali che v_1 è elemento di D_1 e v_2 di D_2.
Una relazione matematica sugli insiemi D_1 e D_2 (chiamati domini) è un sottoinsieme di D_1 x D_2.
Le relazioni possono essere rappresentate graficamente in maniera utilmente espressiva, sotto forma tabellare. Le relazioni (e le relative tabelle) possono essere utilizzate per rappresentare i dati di interesse per qualche applicazione.

Juventus	Lazio	3	1
Lazio	Milan	2	0
Juventus	Roma	1	2
Roma	Milan	0	1

Esempio relazione.

Essa è definita con riferimenti a due domini *intero* e *stringa*, ognuno dei quali compare due volte. La relazione è infatti un sottoinsieme del prodotto cartesiano:

Stringa x Stringa x Intero x Intero

Relazione con attributi

Per organizzare i dati nelle nostre basi di dati, possiamo dire che ciascuna *n*-upla contiene dati fra loro collegati; possiamo poi ricordare che una relazione è un *insieme*, quindi:

- non è definito alcun ordinamento fra le n-uple che le rappresentano c'è per necessità, un ordine, ma è occasionale, due tabelle con le stesse righe, ma in ordine inverso rappresentano la stessa relazione;
- le n-uple di una relazione sono distinte l'una dall'altra, in quanto tra gli elementi di un insieme non possono essere presenti due elementi uguali;
- Al tempo stesso ciascuna n-upla è, al proprio interno, ordinata, l'i-esimo valore di ciascuna proviene dall'i-esimo dominio. Scambiando il terzo dominio con il quarto, cambieremmo completamente il significato della nostra relazione.

L'ordinamento fra i domini di una relazione corrisponde in effetti a una caratteristica insoddisfacente del concetto di relazione matematica rispetto alla possibilità di organizzare e utilizzare i dati. Infatti, in vari contesti dell'informatica si tende a privilegiare notazioni non posizionali, rispetto a quelle posizionali.

Risulta evidente come le informazioni che siano interessati a organizzare nelle relazioni delle nostre basi di dati abbiamo una struttura che si può naturalmente ricondurre a quella dei record: una relazione è sostanzialmente un insieme di record omogenei, cioè definiti sugli stessi campi. Nel caso dei record, a ogni campo è associato un nome, detto *attributo*, che descrive il "ruolo" giocato dal dominio stesso. Dovendo identificare univocamente le componenti, gli attributi di una relazione debbono essere diversi l'uno dall'altro.
Modificando la definizione di relazione con l'introduzione degli attributi, e prima ancora di dare la definizione formale, possiamo vedere che l'ordinamento degli attributi risulta irrilevante:

SquadraDiCasa	*SquadraOspitata*	*RetiCasa*	*RetiOspitata*
Juventus	Lazio	3	1
Lazio	Milan	2	0
Juventus	Roma	1	2
Roma	Milan	0	1

Una relazione con attributi.

A questo punto non serve più avere un ordinamento dei domini poiché anche scambiandoli fra loro il significato rimane comunque certo.
Per formalizzare i concetti, stabiliamo la corrispondenza fra attributi e domini per mezzo della funzione DOM: $X \to D$, che associa a ciascun attributo $A \in X$ un dominio $DOM(A) \in D$. Poi diciamo che una tupla è una funzione *t* che associa a ciascun attributo $A \in X$ un valore del dominio $DOM(A)$.
Una relazione su X è un insieme di tuple su X.
Se *t* è una tupla su X e $A \in X$, allora $t[A]$ indica il valore di *t* su A. es:

t[SquadraOspitata] = Lazio

La stessa relazione è estesa anche ad insiemi di attributi. *t*[SquadraOspitata, RetiOspitata]

Relazioni e basi di dati

Di solito per un qualsiasi scopo non è sufficiente una singola relazione; una base di dati è in generale costituita da più relazioni, le cui tuple contengono valori comuni, ove necessario per stabilire corrispondenze.

Studenti

Matricola	*Cognome*	*Nome*	*Nascita*
276545	Rossi	Maria	25/11/71
485745	Neri	Anna	23/04/72
200768	Verdi	Fabio	12/02/72
587614	Rossi	Luca	10/10/71
937653	Bruni	Mario	01/12/71

Esami

Studente	*Voto*	*Lode*	*Corso*
276545	28		1
276545	27		4
937653	30	L	1
200768	24		4

Corsi

Codice	*Titolo*	*Docente*
1	Analisi	Giani
3	Chimica	Melli
4	Chimica	Belli

Una base di dati relazionale.

La base di dati in esempio mostra delle caratteristiche fondamentali del modello relazionale, che viene spesso indicata dicendo che esso è "basato su valori": i riferimenti fra dati in relazioni diverse sono rappresentati per mezzo di valori dei domini che compaiono nelle tuple.

Rispetto ad un modello basato su record e puntatori, il modello relazionale, basato sui valori, presenta diversi vantaggi:

- esso richiede di rappresentare solo ciò che è rilevante dal punto di vista dell'applicazione; i puntatori sono qualcosa di aggiuntivo, legato ad aspetti realizzativi; nei modelli con puntatori, il programmatore delle applicazioni fa riferimento a dati che non sono significativi per l'applicazione;
- la rappresentazione logica dei dati non fa alcun riferimento a quella fisica, che può anche cambiare nel tempo (indipendenza fisica del dato);
- essendo tutta l'informazione contenuta nei valori, è relativamente semplice trasferire i dati da un contesto ad un altro;

Possiamo a questo punto riassumere le definizioni relative al modello relazionale con un po' di precisione, distinguendo il livello degli schemi da quello delle istanze.

- Uno schema relazionale è costituito da un simbolo *R*, detto *nome della relazione*, e da un insieme di (nomi di) *attributi* $X = \{A_1, A_2, ..., A_n\}$, il tutto di solito indicato con *R(X)*. A ciascun attributo è associato un dominio;
- Uno *schema di base di dati* è un insieme di schemi di relazione con nomi diversi; $\Re = \{R_1(X_1), R_2(X_2), ..., R_n(X_n)\}$
- Una *instanza di relazione* (o semplicemente relazione) su uno schema *R(X)* è un insieme *r* di tuple su *X*;
- Un'*instanza di base di dati* (o semplicemente *base di dati*) su uno schema di base di dati è un insieme di relazioni $r = \{r_1, r_2, ..., r_n\}$, dove ogni r_1, per $1 \le i \le n$, è una relazione su schema $R_i(X_i)$.

Per esemplificare, possiamo dire che lo schema della base di dati in esempio è definito così:

R = {Studenti(Matricola, Cognome, Nome, Data di nascita), Esami (Studente, Voto, Corso), Corsi (Codice, Titolo, Docente)}

Dobbiamo comunque notare che, sulla base della definizione, siano ammissibili relazioni su un solo attributo.

LAVORATORI

Lavoratori
276545
485745
937653

Il modello relazionale permette di rappresentare informazione strutturata in modo articolata, come informazioni fisse e una sere di numero di righe variabili, ognuna relativa ad un insieme di voci omogenee. Per fare un esempio basta pensare ad una base di dati che rappresenti una serie di scontrini di un ristorante. Poiché le nostre relazioni hanno una struttura fissa, non è possibile rappresentare l'insieme delle ricevute con un'unica relazione. Possiamo però rappresentare le relative informazioni per mezzo di due relazioni, una relazione "Ricevute" che contiene i dati presenti una sola volta in ciascuna ricevuta, e la relazione "Dettaglio" che contiene le varie righe di ciascuna ricevuta, entrambe associate alla ricevuta stessa tramite il relativo numero.

Informazione incompleta e valori nulli

La struttura del modello relazionale, è indubbiamente molto semplice e potente. Al tempo stesso impone un certo grado di rigidità, in quanto le informazioni debbono essere rappresentate per mezzo di tuple di dati omogenee. Ad esempio in una relazione relativo a dati di persone, il valore dell'attributo "Telefono" potrebbe non essere disponibile per tutte le tuple. In questo caso, supponendo i numeri telefonici rappresentati per mezzo di interi, potremmo ad esempio utilizzare lo zero per indicare l'assenza di un valore significativo. In generale, però, questa scelta non risulta

soddisfacente, per due motivi: richiede l'esistenza di un valore del dominio mai utilizzato per valori significativi (per una data di nascita è difficile individuare un valore adeguato), l'uso di valori del dominio può generare confusione: la distinzione fra valori veri e valori fittizi è nascosta e quindi i programmi che accedono alla base di dati debbono tenerne conto, distinguendo opportunamente.
Per rappresentare in modo semplice la non disponibilità di valori, il concetto di relazione viene di solito esteso prevedendo che una tupla possa assumere, su ciascun attributo, o un valore del dominio, oppure un valore speciale, detto *valore nullo*, che denota appunto l'assenza di informazione, ma è un valore aggiuntivo rispetto a quelli del dominio.
Nelle rappresentazioni tabellari, si utilizza per il valore nullo il simbolo NULL.
Con riferimento all'esempio precedente degli studenti, bisogna considerare che un valore nullo sulla data di nascita nella prima relazione è tutto sommato ammissibile. Viceversa un valore nullo sul numero di matricola o sul codice di un corso genera problemi maggiori, in quanto questi valori utilizzati per stabilire correlazioni fra tuple di relazioni diverse. Valori nulli nella relazione Esami rende addirittura la tupla corrispondente inutilizzabile.
E' evidente come sia necessario controllare opportunamente la presenza dei valori nulli nelle nostre relazioni: solo alcune configurazioni debbono essere ammesse.

Vincoli di integrità

Le strutture del modello relazionale ci permettono di organizzare le informazioni di interesse per le nostre applicazioni. In molti casi però non è vero che qualsiasi insieme di tuple sullo schema rappresenti informazioni corrette per l'applicazione.
Consideriamo per esempio la solita base di dati:

- un voto pari a 36 in una tupla della relazione Esami non è ammissibile, in quanto i voti debbono essere compresi fra 18 e 30;
- attribuire una lode in una tupla in cui il valore non è 30 non è possibile, in quanto il voto può essere solo 30;
- nella relazione Studenti due studenti diversi con la stessa matricola è ancora una situazione impossibile poiché il numero serve proprio a identificare univocamente gli studenti;
- nella relazione Esami non possono comparire valori di matricola non presenti nella relazione Studenti, analogamente non può comparire un valore di corso non presente nella relazione Corsi.

In una base di dati, è opportuno evitare situazioni come quelle appena descritte.
A tale scopo è stato introdotto il concetto di *vincolo di integrità*, come proprietà che deve essere soddisfatta dalle istanze che rappresentano informazioni corrette per l'applicazione. Ogni vincolo può essere visto come un *predicato* che associa a ogni istanza il valore *vero* o *falso*. Se il predicato assume il valore *vero* diciamo che l'istanza soddisfa il vincolo.
E' possibile classificare i vincoli, distinguendo due categorie, la prima delle quali ha alcuni casi particolari:

- Un vincolo è *intrarelazionale* se il suo soddisfacimento è definito rispetto a singole relazioni della base di dati; può essere divisa in
  - *vincolo di tupla*, che è un vincolo che può essere valutato su ciascuna tupla indipendentemente dall'altra;
  - *vincolo su valori o su dominio* un vincolo, ancora più specifico, definito con riferimento a singoli valori, imponendo una restrizione sul dominio dell'attributo;
- un vincolo è *interrelazionale* se coinvolge più relazioni.

Vincoli di tupla

Esprimono condizioni sui valori di ciascuna tupla, indipendentemente dalle altre tuple.
Una possibile sintassi è quella che permette di definire espressioni booleane con atomi che confrontano valori di attributo o espressioni aritmetiche su valori di attributo.
$(Voto \geq 18) and (Voto \leq 30)$ $(not(Lode = 'lode')) or (Voto = 30)$

Chiavi

In questo paragrafo discutiamo i vincoli di chiave, che sono senz'altro i più importanti del modello relazionale; potremmo addirittura affermare che senza di essi il modello stesso non avrebbe senso.
Nel nostro esempio in cui la matricola identifica univocamente gli studenti, i valori sono tutti diversi. Analogamente, possiamo notare che nella relazione non vi sono coppie di tuple con gli stessi valori su ciascuno dei tre attributi Cognome, Nome e Nascita: anche i dati anagrafici identificano univocamente le persone.
Intuitivamente, una chiave è un insieme di attributi utilizzato per identificare univocamente le tuple di una relazione. Per formalizzare la definizione, procediamo in due passi:

- un insieme *K* di attributi è *superchiave* di una relazione *r* se *r* non contiene due tuple distinte t_1 e t_2 con $t_1[K] = t_2[K]$;
- *K* è *chiave* di *r* se è una superchiave minimale di *r*, cioè non esiste un'altra superchiave K' di *r* che sia contenuta in *K* come sottoinsieme proprio.

Nell'esempio:

- l'insieme {Matricola} è superchiave, nonché superchiave minimale; quindi è una chiave;
- l'insieme {Cognome, Nome, Nascita} è superchiave, inoltre nessuno dei suoi sottoinsiemi è superchiave; quindi è un'altra chiave;
- l'insieme {Matricola, Cognome} è superchiave, peraltro non è superchiave minimale, poiché un suo sottoinsieme è esso stesso superchiave minimale; quindi non è una chiave;

Possiamo ora fare alcune riflessioni sul concetto di chiave, che giustificano l'importanza a esso attribuita. In primo luogo, possiamo notare come ciascuna relazione e ciascuno schema di relazione abbiano sempre una chiave. Il fatto che su ciascuno schema di relazione possa essere definita almeno una chiave garantisce la accessibilità a tutti i valori di una base di dati e la loro univoca identificabilità.

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. Gli esempi suggeriscono la necessità di limitare la presenza di valori nulli nelle chiavi delle relazioni; in pratica, su una delle chiavi, detta *chiave primaria*, si vieta la presenza di valori nulli; sulle altre i nulli valori sono in genere ammessi. Gli attributi che costituiscono la chiave primaria vengono spesso evidenziati attraverso la sottolineatura.

Vincoli di integrità referenziale

INFRAZIONI

Codice	***Data***	***Agente***	***Articolo***	***Prov***	***Numero***
143256	25/10/92	567	44	RM	4E5432
987554	26/10/92	456	34	RM	4E5432
987557	26/10/92	456	34	RM	2F7643
630876	15/10/92	456	53	MI	2F7643
539856	12/10/92	567	44	MI	2F7643

AGENTI

Matricola	***CF***	***Cognome***	***Nome***
567	RSSM...	Rossi	Mario
456	NREL...	Neri	Luigi
638	NREP...	Neri	Piero

AUTO

Prov	***Numero***	***Proprietario***	***Indirizzo***
RM	2F7643	Verdi Pietro	Via Tigli
RM	1A2396	Verdi Pietro	Via Tigli
RM	4E5432	Bini Luca	Via Aceri
MI	2F7643	Luci Gino	Via Aceri

Una base di dati con vincoli di integrità referenziale.

La prima relazione contiene informazioni relative ad un insieme di infrazioni al codice della strada, la seconda agli agenti di polizia e la terza ad un insieme di veicoli. Le informazioni della prima sono rese complete e significative grazie ai riferimenti alle altre due relazioni: alla relazione Agenti, per il tramite dell'attributo Agente, che contiene numeri di matricola di agenti corrispondenti alla chiave primaria della relazione Agenti, e alla relazione Auto, per mezzo degli attributi Prov e Numero, che contengono valori degli omonimi attributi che formano la chiave primaria della relazione Auto.

Un *vincolo di integrità referenziale* (chiamato in inglese *foreign key* o *referential integrity constraint*) fra un insieme di attributi X di una relazione R_1 e un'altra relazione R_2 è soddisfatto se i valori su X di ciascuna tupla dell'istanza di R_1 compaiono come valori della chiave primaria dell'istanza R_2.

Nel caso in cui la chiave di R_2 è unica e composta di un solo attributo B allora il vincolo di integrità referenziale fra l'attributo A di R_1 e la relazione R_2 è soddisfatto se, per ogni tupla t_1 in R_1 per cui $t_1[A]$ non è nullo, esiste una tupla t_2 in R_2 tale che $t_1[A]=t_2[B]$.

Nel caso più generale, in cui la chiave è composta da più attributi, dobbiamo fare attenzione al fatto che ciascuno degli attributi in X deve corrispondere ad un preciso attributo della chiave primaria K di R_2, specificando un ordinamento sia nell'insieme X che in K. Anche se a volte il vincolo nell'ordine degli attributi può apparire pesante, è essenziale. A esempio, non sarà possibile stabilire la corrispondenza nel vincolo di integrità referenziale verso la relazione Auto per mezzo dei nomi degli attributi in quanto essi sono diversi da quelli della chiave primaria di Auto.

3. Algebra e calcolo relazionale

Poiché le basi di dati vengono utilizzate per rappresentare le informazioni di interesse per applicazioni che gestiscono dati, è evidente che i linguaggi per la specifica delle operazioni sui dati stessi costituiscono a loro volta una componente essenziale delle basi di dati e quindi di ciascun modello dei dati. Un aggiornamento può essere visto come una funzione che, data un'istanza di base di dati, produce un'altra base di dati, sullo stesso schema. Un'interrogazione, invece, è essenzialmente una funzione che, data una base di dati, produce una relazione, su un dato schema.

Algebra relazionale

L'algebra relazionale è un linguaggio procedurale basato su concetti di tipo algebrico. Esso è costituito da un insieme di operatori, definiti su relazione e che producono ancora relazioni come risultati.
Questi i vari operatori:

- quelli insiemistici tradizionali, *unione, intersezione, differenza,* che, con piccole avvertenze, possono essere definiti anche sulle relazioni;
- poi quelli più specifici, *ridenominazione, selezione, proiezione*;
- infine quello di *join,* in varie forme, *join naturale, prodotto cartesiano e theta-join.*

Unione, intersezione e differenza

Le relazioni sono insiemi e quindi ha senso definire su di esse gli operatori insiemistici tradizionali di unione, differenza e intersezione. Dobbiamo però prestare attenzione al fatto che una relazione non è genericamente un insieme di tuple, ma un insieme di tuple omogenee, definite sugli stessi attributi. Dunque, seppur potendo, non ha senso definire tali operatori su insiemi di tuple differenti poiché il risultato non avrebbe alcun senso.
Pertanto consideriamo ammissibili nell'algebra relazionali solo applicazioni degli operatori di unione, differenza e intersezione a coppie di operandi definite sugli stessi attributi.

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

Ridenominazione

La limitazione che abbiamo dovuto imporre agli operatori insiemistici, pur giustificata, risulta però particolarmente pesante. Per risolvere il problema, introduciamo uno specifico operatore, che ha come unico obiettivo proprio quello di adeguare i nomi degli attributi, a seconda delle necessità. L'operatore è detto di *ridenominazione*, perché appunto "cambia il nome degli attributi", lasciando inalterato il contenuto delle relazioni.
Esempio:

PATERNITÀ

Padre	***Figlio***
Adamo	Caino
Adamo	Abele
Abramo	Isacco
Abramo	Ismaele

MATERNITÀ

Madre	***Figlio***
Eva	Caino
Eva	Set
Sara	Isacco
Agar	Ismaele

In queste tabelle, seppur l'unione sarebbe sensata, gli attributi "padre" e "madre" differiscono, è quindi impossibile effettuare, ad esempio, un'unione. A tale scopo effettuiamo una rinominazione:

ϑGenitore← Padre (PATERNITÀ)

Genitore	*Figlio*
Adamo	Caino
Adamo	Abele
Abramo	Isacco
Abramo	Ismaele

A questo punto possiamo ottenere l'unione utilizzando due rinominazioni:

ϑGenitore← Padre (PATERNITÀ) ∪ ϑGenitore← Madre (MATERNITÀ)

Genitore	*Figlio*
Adamo	Caino
Adamo	Abele
Abramo	Isacco
Abramo	Ismaele
Eva	Caino
Eva	Set
Sara	Isacco
Agar	Ismaele

Sia r una relazione definita sull'insieme di attributi X e sia Y un (altro) insieme di attributi con la stessa cardinalità. Nel caso si debbano rinominare più attributi si procede similarmente, mentre gli attributi che mantengono il nome di partenza vengono, come visto, omessi.
ϑ Genitore, Progenie ← Padre, Figlio (PATERNITÀ)

Selezione

Possiamo ora esaminare gli operatori tipici dell'algebra relazionale, che permettono effettivamente di manipolare le relazioni. Si tratta di tre operatori: **selezione, proiezione** e **join**; selezione e proiezione svolgono funzioni che possiamo definire complementari. Sono entrambe definite su un operando e producono come risultato una porzione dell'operando. Più precisamente, la selezione produce un sottoinsieme delle tuple, su tutti gli attributi, mentre la proiezione da un risultato cui contribuiscono tutte le tuple, ma su un sottoinsieme degli attributi. La selezione genera "decomposizioni orizzontali" e la proiezione "decomposizioni verticali".
La selezione è indicata con il simbolo σ a pedice del quale viene indicata la "condizione di selezione" opportuna. Le condizioni di selezione possono prevedere confronti fra attributi e confronti fra attributi e costanti, e possono essere complesse, ottenute combinando condizioni semplici con i connettivi logici ∨ (or) ∧ (and) ¬ (not).
Data una relazione r(X), una *formula preposizionale F* su è una formula ottenuta combinando, con i connettivi logici indicati, condizioni atomiche del tipo $A\,\mathcal{J}\,B$ o $A\,\mathcal{J}\,c$, dove:

- $\mathcal{J}$ è un operatore di confronto;
- A e B sono attributi in X sui cui valori il confronto $\mathcal{J}$ abbia senso;
- c è una costante "compatibile" con il dominio di A (cioè tale che il confronto $\mathcal{J}$ sia definito).

Ecco un esempio
IMPIEGATI

Cognome	*Nome*	*Reparto*	*Capo*
Rossi	Mario	Vendite	De Rossi
Neri	Luca	Vendite	De Rossi
Verdi	Mario	Personale	Lupi

Rossi	Marco	Personale	Lupi

La seguente è una selezione:

$J_{\text{Reparto="Vendite"}}(\text{IMPIEGATI})$

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

Proiezione

La definizione dell'operatore di proiezione è ancora più semplice: dati una relazione r(X) e un sottoinsieme Y di X, la *proiezione* di r su Y (indicata con $\boldsymbol{P}y(r)$) è l'insieme di tuple Y ottenute dalle tuple di r considerando solo i valori su Y:

$\boldsymbol{P}y(r)=\{t[Y] \mid t \text{ appartienne a } r\}$

Esempio sulla base della precedente relazione:

$J_{\text{Cognome, Nome}}(\text{IMPIEGATI})$

Cognome	*Nome*
Rossi	Mario
Neri	Luca
Verdi	Mario
Rossi	Marco

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

Reparto	*Capo*
Vendite	De Rossi
Personale	Lupi

In generale, possiamo dire che il risultato di una proiezione contiene al più tante tuple quante l'operando, ma può contenerne di meno, come mostrato nell'esempio. In pratica il numero di righe rimane sicuramente lo stesso solo se la proiezione è fatta su attributi che corrispondono a superchiave della relazione.

Join

L'operatore 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 su valori. Esistono due varianti dell'operatore, il join naturale e il theta-join.

Join neutrale. Il *join naturale* è un operatore $\bowtie$ 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.

Esempio sulla base della precedente relazione:

R1

Impiegato	*Reparto*
Rossi	Vendite
Neri	Produzione
Bianchi	Produzione

R2

Reparto	*Capo*
Produzione	Mori
Vendite	Bruni

R1

Impiegato	*Reparto*	*Capo*
Rossi	Vendite	Bruni
Neri	Produzione	Mori
Bianchi	Produzione	Mori

E’ opportuno notare che è molto frequente eseguire join sulla base di valori della chiave di una delle relazioni coinvolte, esplicitando i riferimenti fra tuple che, come abbiamo più volte ripetuto, sono realizzati per mezzo di valori, soprattutto valori di chiavi.

Join completi e incompleti. Discutiamo ora diversi esempi di join, per svolgere alcune importanti considerazioni, con riferimento alla dimensione del risultato. Possiamo dire che abbiamo un join completo quando ciascuna tupla di ciascuna relazione contribuisce ad almeno una tupla del risultato (come ad esempio nel caso precedente). Può comunque avvenire che ci siano tuple di una relazione che non compaiono nel risultato; in inglese tali tuple vengono chiamate *dangling* (dondolanti).
Es:
Esempio sulla base della precedente relazione:

R1

Impiegato	***Reparto***
Rossi	Vendite
Neri	Produzione
Bianchi	Produzione

R2

Reparto	***Capo***
Produzione	Mori
Acquisti	Bruni

$R1 \bowtie R2$

Impiegato	***Reparto***	***Capo***
Neri	Produzione	Mori
Bianchi	Produzione	Mori

Come caso limite, è ovviamente possibile che nessuna delle tuple degli operandi sia combinabile, avremo in questo caso una relazione vuota come risultato.
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 delle due relazioni.

Join esterni. La caratteristica dell’operatore di join di “tralasciare” le tuple di una relazione senza controparte è utile in molti casi ma potenzialmente pericolosa in altri, in quanto può portare a omettere informazioni rilevanti. Allo scopo è stata proposta una variante chiamata *join esterno* (outer join) che le tuple diano un contributo al risultato, eventualmente estese con valori nulli ove non vi siano controparti opportune. Esistono tre varianti: il join esterno *sinistro*, che estende solo le tuple del primo operando, quello *destro*, che estende solo le tuple del secondo operando, e quello *completo* che le estende tutte.
Eccone la dimostrazione sugli esempi precedenti:

$R1 \bowtie_{LEFT} R2$

Impiegato	***Reparto***	***Capo***
Rossi	Vendite	NULL
Neri	Produzione	Mori
Bianchi	Produzione	Mori

$R1 \bowtie_{RIGHT} R2$

Impiegato	***Reparto***	***Capo***
Neri	Produzione	Mori

Bianchi	Produzione	Mori
NULL	Acquisti	Bruni

$R1 \bowtie_{FULL} R2$

Impiegato	*Reparto*	*Capo*
Rossi	Vendite	NULL
Neri	Produzione	Mori
Bianchi	Produzione	Mori
NULL	Acquisti	Bruni

Join n-ario, intersezione e prodotto cartesiano. Vediamo alcune proprietà dell'operatore di join naturale. In primo luogo è commutativo, poiché $r_1 \bowtie r_2$ è sempre uguale a $r_2 \bowtie r_1$, e associativo, in quanto $r_1 \bowtie (r_2 \bowtie r_3)$ è uguale a $(r_1 \bowtie r_2) \bowtie r_3$.
Particolare attenzione suscita il caso in cui gli attributi delle due relazioni sono fra loro disgiunti, in cui ciascuna tupla deriva tuple non hanno attributi in comune, non viene richiesta a esse nessuna condizione per partecipare insieme al join. Il risultato del join contiene le tuple ottenute combinando, in tutti i modi possibili, le tuple degli operandi. In questo caso praticamente diventa un prodotto cartesiano.

Theta-join ed equi-join. Osservando le caratteristiche di un prodotto cartesiano si nota che ha spesso ben poca utilità. Per questa ragione, viene spesso definito un operatore deviato, il *theta-join*, come prodotto cartesiano seguito da una selezione, nel modo seguente: $r_1 \bowtie_F r_2 = \sigma_F (r_1 \bowtie r_2)$
Esempio sulla base della precedente relazione:

IMPIEGATI

Impiegato	*Progetto*
Rossi	A
Neri	A
Neri	B

PROGETTI

Codice	*Nome*
A	Venere
B	Marte

IMPIEGATI $\bowtie_{Progetto=Codice}$ 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.
Dal punto di vista pratico il theta-join ed ancor più l'equi-join hanno una grande importanza, in quanto la maggior parte dei sistemi di basi di dati effettivamente esistenti non utilizzano i nomi di attributo per correlare relazioni, e pertanto non utilizzano il join naturale ma l'equi-join e il theta-join.

Interrogazioni in algebra relazionale

In generale un'interrogazione può essere definita come una funzione che, applicata a istanze di basi di dati, produce relazioni. In algebra relazionale, le interrogazioni su uno schema di base di dati vengono formulate con espressioni i cui atomi sono (nomi di) relazioni dello schema.
Esempi di interrogazioni su schema con due relazioni:
IMPIEGATI(Matricola, Nome, Età, Stipendio)
SUPERVISIONE(Capo, Impiegato)

La prima interrogazione: *trovare matricola, nome ed età degli impiegati che guadagnano più di 40 milioni.*

$\Pi_{\text{Matricola, Nome, Età}}(\sigma_{\text{Stipendio>40}}(\text{IMPIEGATI}))$

La seconda interrogazione: *trovare le matricole dei capi degli impiegati che guadagnano più di 40 milioni*.

$\Pi_{\text{Capo}}(\text{Supervisione} \bowtie_{\text{Impiegato=Matricola}} (\sigma_{\text{Stipendio>40}}(\text{IMPIEGATI})))$

La terza interrogazione: *trovare nome e stipendio dei capi degli impiegati che guadagnano più di 40 milioni*.

$\Pi_{\text{NomeC,StipC}}(J_{\text{MatrC,NomeC,StipC,EtàC} \leftarrow \text{Matr,Nome,Stip,Età}}(\text{IMPIEGATI}) \bowtie_{\text{MatrC=Capo}} (\text{Supervisione} \bowtie_{\text{Impiegato=Matricola}} (\sigma_{\text{Stipendio>40}}(\text{IMPIEGATI}))))$

La quarta: *trovare gli impiegati che guadagnano più del rispettivo capo, mostrando matricola, nome e stipendio di ciascuno di essi e del capo.*

$\Pi_{\text{Matr,Nome,Stip,MatrC,nomeC,StipC}}(\sigma_{\text{Stip>StiC}}(J_{\text{MatrC,NomeC,StipC,EtàC} \leftarrow \text{Matr,Nome,Stip,Età}}(\text{IMPIEGATI}) \bowtie_{\text{MatrC=Capo}} (\text{Supervisione} \bowtie_{\text{Imp=Matr}} (\text{IMPIEGATI}))))$

La quinta: *matricola e nome dei capi i cui impiegati guadagnano tutti più di 40 milioni*.

$\Pi_{\text{Matr,Nome}}(\text{IMPIEGATI} \bowtie_{\text{MatrC=Capo}} (\Pi_{\text{Capo}}(\text{Supervisione}) - \Pi_{\text{Capo}}(\text{Supervisione} \bowtie_{\text{Imp=Matr}} ((\sigma_{\text{Stip<=40}}(\text{IMPIEGATI})))))$

Equivalenza di espressioni algebriche

L'algebra relazionale permette di formulare espressioni fra loro equivalenti, facendo attenzione al fatto che l'equivalenza può essere assoluta, oppure dipendere dallo schema. L'equivalenza di espressioni dell'algebra risulta particolarmente importante dal punto di vista applicativo, nella fase di esecuzione delle interrogazioni. Infatti, le interrogazioni, specificate in linguaggio SQL, vengono tradotte in algebra relazionale e, appunto, con riferimento all'algebra, viene valutato il costo, sostanzialmente in termini di dimensioni dei risultati intermedi. In questo contesto, vengono spesso utilizzate trasformazioni di equivalenza, cioè operazioni che sostituiscono un'espressione con un'altra a essa equivalente. In particolare risultato interessanti le trasformazioni che riducono le dimensioni dei risultati intermedi e quelle che preparano un'espressine all'applicazione di una trasformazione che riduce le dimensioni dei risultati intermedi.

1. Atomizzazione delle selezioni: una selezione congiuntiva può essere sostituita da una cascata di selezioni atomiche;
2. Idempotenza delle proiezioni: una proiezione può essere trasformata in una cascata di proiezioni che eliminano i vari attributi in fasi diverse;
3. Anticipazione della selezione rispetto al join;
4. Anticipazione della proiezione rispetto al join;
5. Inglobamento di una selezione in un prodotto cartesiano a formare un join;
6. Distributività della selezione rispetto all'unione;
7. Distributività della selezione rispetto alla differenza;
8. Distributività della proiezione rispetto all'unione;

Algebra con valori nulli

Fin'ora abbiamo ipotizzato che le espressioni venissero applicate a relazioni prive di valori nulli. Nel caso ci siano invece dei valori nulli questo dato sconosciuto può rientrare in casi in cui quel valore serviva per una operazione. A proposito di queste interrogazioni, è stato proposto di utilizzare una logica a tre valori, in cui un predicato può essere vero, falso, o assumere un terzo nuovo valore di verità che chiamiamo *unknown* e rappresentiamo con il simbolo U. Le tabelle di verità dei connettivi logici, *not*, *and*, e *or*, per tenere conto del nuovo valore logico, si estendono nel modo seguente:

not	
F	V
U	V
V	F

and	V	U	F
V	V	U	F
U	U	U	F
F	F	F	F

or	V	U	F
V	V	V	V
U	V	U	U
F	V	U	F

Il valore *unknown* rappresenta un valore di verità intermedio tra vero e falso, e il significato dei tre connettivi in questo contesto diventa il seguente: il *not* è vero solo se il valore di partenza è falso, l'*and* è vero solo se tutti e due sono veri, e l'*or* è vero se almeno uno dei termini è vero.

Il metodo migliore per superare in pratica gli inconvenienti appena discussi consiste nel trattare i valori nulli da un punto di vista meramente sintattico e può essere usato sostanzialmente nello stesso modo sia con una logica a due valori sia con una a tre valori. Si introducono due nuove forme di condizioni atomiche di selezione, con lo scopo di verificare se un valore è specificato oppure nullo:

- *A* IS NULL assume valore vero su una tupla *t* se il valore di *t* su *A* è nullo e falso se esso è specificato;
- *A* IS NOT NULL assume valore vero se una tupla *t* se il valore di *t* su *A* è specificato e falso se esso è nullo.

Viste

Nel modello relazione, la tecnica prevista per mettere a disposizioni rappresentazioni diverse per gli stessi dati è quella delle *relazioni derivate,* relazioni in cui il contenuto è funzione del contenuto di altre relazioni. E' possibile che una relazione derivata sia funzione di altre relazioni derivate, a condizione che esista un ordinamento fra le relazioni derivate tale che ogni relazione sia definita solo in termini di relazioni di base e di relazioni derivate che la precedono nell'ordinamento.
In linea di principio, possono esistere due tipi di relazioni derivate:

- *Viste materializzate*: relazioni derivate effettivamente memorizzate nella base di dati;
- *Relazioni virtuali* (chiamate semplicemente *viste*): relazioni definite per mezzo di funzioni, non memorizzate nella base di dati, ma utilizzabili nelle interrogazioni come se lo fossero.

Le viste materializzate hanno il vantaggio di essere immediatamente disponibili ma è spesso oneroso mantenere il loro contenuto allineato con quelle delle relazioni da cui derivano. Le relazioni virtuali debbono essere ricalcolate per ogni interrogazione ma non presentano problemi di allineamento.
Le viste vengono definite nei sistemi relazionali per mezzo di espressioni del linguaggio di interrogazione.
L'uso delle viste può risultare vantaggioso per diversi ordini di motivi:

- Un utente interessato solo ad 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 ristrutturazione 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 della stesa, ma ricavabili dalle nuove relazioni.

Calcolo relazionale

Con il termine *calcolo relazionale* si fa riferimento ad una famiglia di linguaggi di interrogazione, basati sul calcolo dei predicati del primo ordine, che hanno la caratteristica di essere *dichiarativi,* cioè di specificare le proprietà del risultato delle interrogazioni, anziché la procedura seguita per generarlo. Esistono diverse versioni del calcolo relazionale e non è possibile illustrarle tutte. La più vicina al calcolo dei predicati è quella del *calcolo relazione su domini,* che presenta in modo naturale le caratteristiche originali di questi linguaggi. La seconda versione è quella del *calcolo su*

tuple con dichiarazione di range, che costituisce la base per molti dei costrutti disponibili per le interrogazioni nel linguaggio SQL.

Calcolo relazionale su domini

Le espressioni del calcolo relazione su domini hanno la forma: $\{A_1 : x_1,..., A_k : x_k \mid f\}$ dove:

- $A_1,..., A_k$ sono attributi distinti;
- $x_1,..., x_k$ sono variabili;
- f è una formula.

La lista di coppie $A_1 : x_1,..., A_k : x_k$ viene chiamata *target listi* in quanto definisce la struttura del risultato, che è costituita dalla relazione su $A_1,..., A_k$ che contiene le tuple i cui valori sostituiti a $x_1,..., x_k$ rendono vera la formula.

Prendiamo a questo punto le solite cinque interrogazioni:
IMPIEGATI(Matricola, Nome, Età, Stipendio)
SUPERVISIONE(Capo, Impiegato)

La prima interrogazione: *trovare matricola, nome ed età degli impiegati che guadagnano più di 40 milioni.*
{Matr : *m*, Nome : *n*, Età : *e* | $\exists s$(IMPIEGATI(Matr : *m*, Nome : *n*, Età : *e*, Stipendio : *s*) $\wedge$ *s* > 40) }
Che può anche essere scritta senza il quantificatore:
{Matr : *m*, Nome : *n*, Età : *e* | IMPIEGATI(Matr : *m*, Nome : *n*, Età : *e*, Stipendio : *s*) $\wedge$ *s* > 40 }

La seconda interrogazione: *trovare le matricole dei capi degli impiegati che guadagnano più di 40 milioni*.
{Capo : *c e* | IMPIEGATI(Matricola : *m*, Nome : *n*, Età : *e*, Stipendio : *s*) $\wedge$ SUPERVISIONE(Impiegato : *m*, Capo : *c*) $\wedge$ s>40 }

La terza interrogazione: *trovare nome e stipendio dei capi degli impiegati che guadagnano più di 40 milioni*.
{NomeC : *nc*, StipC : *sc* |
IMPIEGATI(Matricola : *m*, Nome : *n*, Età : *e*, Stipendio : *s*) $\wedge$ *s* > 40
SUPERVISIONE(Impiegato : *m*, Capo : *c*) $\wedge$
IMPIEGATI(Matricola : *m*, Nome : *nc*, Età : *ec*, Stipendio : *sc*) }

La quarta: *trovare gli impiegati che guadagnano più del rispettivo capo, mostrando matricola, nome e stipendio di ciascuno di essi e del capo*.
{Matr : *m*, Nome : *n*, Stip : *s,* MatrC *: mc,* NomeC : *nc*, StipC : *sc* |
IMPIEGATI(Matr : *m*, Nome : *n*, Età : *e*, Stipendio : *s*) $\wedge$
SUPERVISIONE(Impiegato : *m*, Capo : *c*) $\wedge$
IMPIEGATI(Matr : *c*, Nome : *nc*, Età : *ec*, Stipendio : *sc*) $\wedge$ *s* > *sc*}

La quinta: *matricola e nome dei capi i cui impiegati guadagnano tutti più di 40 milioni*.
{Matricola : *c*, Nome : *n* | IMPIEGATI(Matr : *m*, Nome : *n*, Età : *e*, Stipendio : *s*) $\wedge$
SUPERVISIONE(Impiegato : *m*, Capo : *c*) $\wedge$
$\neg\exists m'(\exists n'(\exists e'(\exists s'$ (IMPIEGATI(Matr : *m'*, Nome : *n'*, Età : *e'*, Stipendio : *s'*) $\wedge$
SUPERVISIONE(Impiegato : *m'*, Capo : *c*)) $\vee$ *s'*<= 40))))}

Pregi e difetti del calcolo su domini

Il calcolo relazionale presenta aspetti interessanti, soprattutto per la dichiaratività, ma anche alcuni difetti e limitazioni, che è opportuno discutere, per arrivare a versioni più interessanti e significative dal punti di vista pratico.

Innanzitutto un'espressione di un linguaggio di interrogazione è *indipendente dal dominio* se il risultato, su ciascuna istanza di base di dati, non varia al variare del dominio rispetto al quale l'espressione è valutata. Il calcolo relazionale non è indipendente dal dominio.
Se a questo punto diciamo che due linguaggi di interrogazione sono equivalenti quanto per ogni espressione dell'uno esiste un'espressione dell'altro a essa equivalente e viceversa, possiamo affermare che algebra e calcolo non sono equivalenti, poiché il calcolo, al contrario dell'algebra, ammette espressioni dipendenti dal dominio.
Oltre al problema della possibile dipendenza dal dominio, il calcolo relazione presenta un altro svantaggio, quello di richiedere numerose variabili, speso una per ciascun attributo di ciascuna relazione coinvolta. Peraltro, diventa poi necessario associare una struttura a ciascuna variabile e realizzare opportunamente le operazioni di confronto. Potremmo a questo punto definire un calcolo relazionale su tuple perfettamente corrispondente al calcolo su domini, ed equivalente a esso, quindi anche con la limitazione della dipendenza dal dominio. Nell'ambito del calcolo relazionale su tuple abbiamo quello con dichiarazioni di range che supera il problema della dipendenza dal dominio, attraverso la diretta associazione delle variabili alle relazioni della base di dati.

Calcolo su tuple con dichiarazioni di range

Le espressioni del *calcolo su tuple con dichiarazioni di range* hanno la forma:

$$\{T \mid L \mid f\}$$

dove:
T è la *target list*, con elementi del tipo $Y : x.Z$ con x variabile e Z sequenze di attributi;
L è la *range list*, che elenca le variabili libere della formula f con i relativi campi di variabilità;
f è una formula con:

- atomi del tipo x, AJc o $x.A_1Jx_2.A_2$;
- connettivi come nel calcolo su domini;
- quantificatori che associano i range alle relative variabili:
  $\exists x(R)(f)$ $\forall x(R)(f)$

Le dichiarazioni di range, introducendo le variabili, specificano che esse possono assumere come valore solo tuple nella relazione rispettivamente associata.

Prendiamo a questo punto le solite cinque interrogazioni:
IMPIEGATI(Matricola, Nome, Età, Stipendio)
SUPERVISIONE(Capo, Impiegato)

La prima interrogazione: *trovare matricola, nome ed età degli impiegati che guadagnano più di 40 milioni.*
{*i*.(Matricola, Nome, Età) | *i* (IMPIEGATI) | *i*.Stipendio > 40}

La seconda interrogazione: *trovare le matricole dei capi degli impiegati che guadagnano più di 40 milioni*.
{*s*.Capo | *i* (IMPIEGATI), *s*(SUPERVISIONE) | i.Matricola = *s*.Impiegato ∧ *i*.Stipendio > 40}

La terza interrogazione: *trovare nome e stipendio dei capi degli impiegati che guadagnano più di 40 milioni*.
{NomeC, StipC : *i'*.(Nome,Stip) |
i' (IMPIEGATI), *s*(SUPERVISIONE),*i*(IMPIEGATI) |
i'.Matricola = *s*.Capo ∧ *s*.Impiegato = *i*.Matricola ∧ *i*.Stipendio > 40}

La quarta: *trovare gli impiegati che guadagnano più del rispettivo capo, mostrando matricola, nome e stipendio di ciascuno di essi e del capo.*
{ *i*. (nome, Matr, Stip), NomeC, MatrC, StipC . *i'*.(Nome, Matr, Stip) |
i(IMPIEGATI), *s*(SUPERVISIONE), *i'*(IMPIEGATI) |

i.Matricola = s.Impiegato $\wedge$ s.Capo = i'.Matricola $\wedge$ i.Stipendio > i'.Stipendio }

La quinta: *matricola e nome dei capi i cui impiegati guadagnano tutti più di 40 milioni*.
{ i. (Matricola, nome) |
i(IMPIEGATI), s(SUPERVISIONE) |
i.Matricola = s.Capo $\wedge$ $\forall$ i'(IMPIEGATI)($\forall$ s'(SUPERVISIONE)
($\neg$ (s.Capo = s'.capo $\wedge$ s'.Impiegato = i'.Matricola) $\vee$ i'.Stipendio > 40))}

Purtroppo, il calcolo su tuple con dichiarazioni di range non permette di esprimere tutte le interrogazioni che possono essere formulate in algebra relazionale. In particolare, le interrogazioni i cui risultati possono provenire indifferentemente da due o più relazioni non possono essere espresse in questa versione del calcolo.
Per questo motivo, SQL, il linguaggio pratico effettivamente utilizzato per l'interrogazione di basi di dati e che è basato sul calcolo su tuple con dichiarazioni di range, prevede un costrutto esplicito di unione, per esprimere interrogazioni che altrimenti risulterebbero non esprimibili. Comunque, mentre l'operatore di unione non è esprimibile in questa versione del calcolo, gli operatori di intersezione e differenza risultano esprimibili.

Datalog

L'idea fondamentale su cui si basa il linguaggio *Datalog* è quella di adattare alle basi di dati il linguaggio di programmazione logica *Prolog*. Sintatticamente ne è una versione semplificata.
Abbiamo in Datalog due tipi di predicati:

- i predicati *estensionali*, che corrispondono alle relazioni nella base di dati;
- i predicati *intenzionali*, che sono specificati per mezzo di regole logiche. Concettualmente definiscono viste sulla base di dati.

Le *regole Datalog* hanno la forma: *testa* $\leftarrow$ *corpo* in cui:

- la *testa* è un predicato atomico simile a quelli utilizzati nel calcolo relazionale su domini;
- il *corpo* è una lista di condizioni atomiche dello stesso tipo e/o di condizioni di confronto fra variabili o fra variabili e costanti.

Sono imposte le seguenti condizioni:

- I predicati estensionali possono comparire solo nel corpo delle regole;
- Se una variabile compare nella testa di una regola, allora deve comparire anche nel corpo della stessa regola;
- Se una variabile compare in un atomo di confronto, allora deve comparire anche in un atomo nel corpo della stessa regola.

Una caratteristica fondamentale del Datalog, che lo distingue dagli altri linguaggi visti fin'ora è la ricorsività: è possibile che un predicato intenzionale sia definito in termini di se stesso.

4. SQL

SQL è un acronimo di *Structured Query Language*. Era originariamente il linguaggio di interrogazione del DBMS relazionale *System R*. SQL è stato poi adottato da molti altri sistemi, è stato standardizzato ed è diventato il linguaggio di riferimento per le basi di dati relazionali.
SQL no è solo un linguaggio di interrogazione. Contiene infatti al suo interno sia le funzionalità di un *Data Definition Language*, DDL, sia quelle di un *Data Manipulation Language*, DML.

Standardizzazione di Sql

La diffusione di SQL è dovuta in gran parte alla intensa opera di standardizzazione dedicata a questo linguaggio dagli organismi di standardizzazione ANSI e ISO.
La prima definizione di uno standard per il linguaggio è stata promulgata nel 1986 dall'ANSI. Lo standard è stato poi esteso in modo limitato nel 1989; l'aggiunta più significativa di questa versione è stata la definizione dell'integrità referenziale. A questa versione dello standard si fa riferimento con il nome SQL-89.
Una seconda versione, in gran parte compatibile con la prima ma molto arricchita di nuove funzionalità, è stata pubblicata nel 1992. A questa versione si fa riferimento col nome SQL-92 o SQL-2.
E' stata poi presentata una versione recente dello standard, SQL-3, che verrà anche chiamata SQL-99. SQL-3 è pienamente compatibile con SQL-2 ma è ancora lontano dall'essere comunemente adottato.
Per quantificare in modo preciso l'aderenza allo standard, sono stati definiti tre livelli di complessità dei costrutti del linguaggio, denominati rispettivamente *Entry SQL, Intermediate SWL* e *Full SQL*. Il livello Entri SQL è abbastanza simile a SQL-89. Il livello intermediate SQl contiene le caratteristiche ritenute più importanti per rispondere alle esigenze del mercato, e viene attualmente offerto da diverse versioni recenti dei prodotti relazionali di maggior diffusione. Il libello Full SQL contiene delle funzioni avanzate, che i sistemi stanno aggiungendo progressivamente.

Definizioni dei dati in SQL

In questo paragrafo illustriamo la definizione degli schemi delle basi di dati. Questa sarà la notazione adottata:

- le parentesi angolari (<,>) permettono di isolare un termine della sintassi;
- le parentesi quadre ([,]) indicano che il termine all'interno è opzionale, ovvero può non comparire o comparire una sola volta;
- le parentesi graffe ({,}) indicano invece che il termine racchiuso può non comparire o essere ripetuto un numero arbitrario di volte;
- le barre verticali (|) indicano che deve essere scelto uno tra i termini separati dalle barre; un elenco di termini in alternativa può essere racchiuso tra parentesi angolari.

Le parentesi tonde dovranno essere sempre intese come termini del linguaggio SQL e non come simboli per la definizione della grammatica.
I termini del linguaggio sono rappresentati con il **`questo font`**, mentre i termini variabili sono scritti in *corsivo.*

I domini elementari

SQL mette a disposizione sei famiglie di domini elementari, a partire dai quali si possono definire i domini da associare agli attributi dello schema.

Carattere

Il dominio **`character`** permette di rappresentare singoli caratteri oppure stringhe; la lunghezza delle stringhe di caratteri può essere fissa o variabile.

La sintassi è la seguente:

character [**varying**] [(*Lunghezza*)]
[**character set** *NomeFamigliaCaratteri*]

Es:
character(20)
character varying (1000) **character set Greek**

Bit

Questo dominio introdotto in SQL-2, viene utilizzato da attributi che possono assumere solo il valore 0 o il valore 1. Il dominio *bit* viene tipicamente usato per rappresentare attributi, detti *flag*, che specificano se l'oggetto rappresentato da una tupla possiede o meno una certa proprietà.

bit [**varying**] [(*Lunghezza*)]

Es:
bit(1)
bit varying (16) anche usato **barbit**(16)

Tipi numerici esatti

Questa famiglia contiene i domini che permettono di rappresentare valori esatti, interi o con una parte decimale di lunghezza prefissata. SQL mette a disposizione quattro diversi tipi numerici esatti:

- **numeric** [(*Precisione* [, *Scala*])]
- **decimal** [(*Precisione* [, *Scala*])]
- **integer**
- **smallint**

I domini **numeric** e **decimal** rappresentano numeri in base decimale. Il parametro *Precisione* specifica il numero di cifre significative (prima della virgola), il parametro *scala* specifica la scala di rappresentazione, ovvero quante cifre devono apparire dopo la virgola.
Nel caso non interessi avere una parte frazionaria si utilizzano **integer** e **smallint**.

Tipi numerici approssimati

SQL fornisce i seguenti tipi:

- **float** [(*Precisione*)]
- **double precision**
- **real**

Tutti questi domini permettono di descrivere numeri approssimati mediante una rappresentazione in virgola mobile, in cui a ciascun numero corrisponde una coppia di valori: la mantissa e l'esponente. La mantissa è un valore frazionario, mentre l'esponente è un numero intero. Al dominio float può essere associata una precisione, che rappresenta il numero di cifre dedicate alla rappresentazione della mantissa, mentre la precisione nell'esponente dipende dall'implementazione.

Data e ora

Questa famiglia di domini permette di rappresentare *istanti* di tempo e offre tre diverse forme:

- **date**
- **time** [(*Precisione*)] [**with time zone**]
- **timestamp** [(*Precisione*)] [**with time zone**]

Ciascuno di questi domini è strutturato e decomponibile in un insieme di campi. Il dominio **date** ammette i campi **year**, **month** e **day**, il dominio **time** ammette i campi **hour**, **minute** e **second**, infine **timestamp** ammette tutti i campi, da **year** a **second**.

Se l'opzione **with time zone** è specificata, allora risulta possibile accedere a due campi **timezone hour** e **timezone minute** che rappresentano la differenza di fuso orario tra l'ora locale e l'ora universale.

Intervalli temporali

Questa famiglia di domini permette di rappresentare intervalli di tempo. La sintassi è:

interval *PrimaUnitàDiTempo* [**to** *UltimaUnitàDiTempo*]

PrimaUnitàDiTempo e *UltimaUnitàDiTempo* definiscono le unità di misura che devono essere usate, dalla più precisa alla meno precisa. E' così possibile definire domini come **interval year to month**. La prima unità di misura che compare nella definizione, qualunque essa sia, può essere caratterizzata dalla precisione, che rappresenta il numero di cifre, in base 10, usate nella rappresentazione.

Definizione di schema

SQL consente la definizione di uno schema di base di dati come collezione di oggetti; ogni schema è costituito da un insieme di *domini*, *tabelle*, *indici*, *asserzioni*, *viste* e *privilegi*, definito dalla seguente sintassi:

create schema [*nomeSchema*] [**authorization** *Autorizzazione*]
{ *DefElementoSchema* }

Autorizzazione rappresenta il nome dell'utente proprietario dello schema, se non specificato viene assunto quello dell'utente.

Definizione delle tabelle

Una tabella SQL è costituita da una collezione ordinata di attributi e da un insieme di vincoli.

create table *NomeTabella*
(
NomeAttributo Dominio [*ValoreDiDefault*] [*Vincoli*]
{ , *NomeAttributo Dominio* [*ValoreDiDefault*] [*Vincoli*] }
Altri vincoli
)

Esempio:

```
create table Dipartimento
(
  Nome        char(20) primary key,
  Indirizzo   char(50),
  Città       char(20)
)
```

Definizione dei domini

Nella definizione delle tabelle si può far riferimento a domini predefiniti del linguaggio, o a domini definiti dall'utente a partire dai domini predefiniti.

create domanin *NomeDominio* **as** *TipoDiDato*
[*ValoreDiDefault*]
[*Vincoli*]

Al contrario dei meccanismi di definizione dei tipi dei linguaggi di programmazione, SQL non mette a disposizione dei costruttori di dominio come il record o l'array. Questa caratteristica,

derivata dal modello relazionale dei dati, il quale richiede che tutti gli attributi siano caratterizzati da un dominio elementare. La dichiarazione di nuovi domini permette di associare un insieme di vincoli ad un nome di dominio, il che è importante quando ad esempio si deve ripetere la stessa definizione di attributo nell'ambito di diverse tabelle.

Specifica di valori di default

Nella sintassi per la definizione dei domini e delle tabelle, si può osservare la presenza di un termine *ValoreDiDefault* in corrispondenza di ogni dominio ed attributo.

Default < *GenericoValore* | **user** | **null** >

GenericoValore rappresenta un valore compatibile con il dominio. L'opzione **user** impone come valore di default l'identificativo dell'utente che esegue il comando di aggiornamento della tabella. L'opzione **null** corrisponde al valore di default di base.

Vincoli intrarelazionali

Sia nella definizione di domini che nella definizione delle tabelle è possibile definire dei vincoli, ovvero delle proprietà che devono essere verificate da ogni istanza della base di dati. Il costrutto più potente per specificare vincoli generici, sia interrelazionali che intrarelazionali, è il costrutto di check, che richiede però di formulare delle interrogazioni sulla base di dati.
I più semplici vincoli di tipo intrarelazionale sono i vincoli *not null*, *inique* e *primari key*.

Il vincolo **not null** indica che il valore nullo non è ammesso come valore dell'attributo;
Un vincolo **unique** si applica ad un attributo o un insieme di attributi di una tabella e impone che i valori dell'attributo siano una (super)chiave, cioè righe differenti della tabella non possano avere gli stessi valori, eccezion fatta per il valore nullo, sempre che sia ammesso.
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; in questo caso si fa seguire la specifica dell'attributo dalla parola chiave **unique** (così come per **not null**).
La seconda alternativa è invece necessaria quando bisogna definire il vincolo per un insieme di attributi. In questo caso, dopo aver definito gli attributi della tabella, si usa la seguente espressione:

unique (*Attributo* { , *Attributo* })

Ecco un esempio:

```
Nome        character(20) not null,
Cognome     character(20) not null,
unique (Cognome,Nome)
```

ecco un altro esempio:

```
Nome        character(20) not null unique,
Cognome     character(20) not null unique
```

Si noti che i due esempi sono differenti e danno diversi vincoli.

Primary key

Come noto è di solito necessario per specificare per ogni relazione la *chiave primaria*, il più importante tra gli identificatori della relazione. SQL permette così di specificare il vincolo *primary key* una sola volta per ogni tabella. Come il vincolo *inique*, 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 ne fanno parte non possono assumere il valore nullo

```
Nome        character(20),
Cognome     character(20),
```

```
primary key (Cognome, Nome)
```

Vincoli interrelazionali

I vincoli interrelazionali più diffusi e significativi sono i vincoli di integrità referenziale. In SQL, per la definizione si usa l'apposito vincolo di *foreign key*, ovvero di *chiave esterna*.
Il vincolo impone che per ogni riga della tabella 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 vincolo che la sintassi impone è che l'attributo cui si fa riferimento sia soggetto a un vincolo *unique.* Più attributi possono essere coinvolti nel vincolo, quando la chiave della tabella esterna è costituita da un insieme di attributi.
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 invece 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`**.
Per tutti gli altri vincoli visti fino ad ora, si assume che quando il sistema rileva una violazione, il comando di aggiornamento venga 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 offerte diverse alternative per rispondere alle violazioni generate da modifiche sulla tabella esterna. Il motivo di questa asimmetria è dovuto al particolare significato della tabella esterna, che sul piano applicativo rappresenta la tabella principale alle cui variazioni la tabella interna deve adeguarsi. La politica di reazione può essere diversa a seconda del comando di aggiornamento che introduce le violazioni.
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, senza che il sistema provi a 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 nullo al posto del valore cancellato nella tabella esterna;
- **`no action`**; la cancellazione non viene consentita.

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 e drop.

Comando alter.
Il comando 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 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.

Comando drop.
Mentre il comando alter effettua delle modifiche sui domini o sullo schema delle tabelle, il comando drop permette di rimuovere dei componenti.

```
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. Con l'opzione cascade invece, tutti gli oggetti specificati devono essere rimossi. 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, ovvero non vi sono elementi nella cui definizione compaiono elementi che sono stati rimossi.

Cataloghi relazionali

Anche se solo in parte previsto dallo standard, tutti i DBMS relazionali gestiscono il proprio dizionario dei dati (ovvero la descrizione delle tabelle presenti nella base dei dati) mediante una struttura relazionale, cioè tramite tabelle. La base di dati contiene anche quindi un insieme di tabelle contenente i cosiddetti metadati, che costituisce il catalogo della base di dati. Tale caratteristica delle implementazioni dei sistemi relazionali viene detta riflessività. I comandi di definizione e modifica dello schema della base di dati potrebbero così essere sostituiti da comandi di manipolazione operanti direttamente sulle tabelle del dizionario dei dati, rendendo superflua l'introduzione di appositi comandi per la definizione dello schema. Questa ipotesi va però scartata, sia per il poco successo della standardizzazione dei dizionari, sia per il bisogno di rendere immediatamente riconoscibili i comandi di manipolazione degli schemi.

Interrogazioni in SQL

La parte di SQL dedicata alla formulazione di interrogazioni fa parte del DML.

Dichiaratività di SQL

SQL esprime le interrogazioni in modo dichiarativo, ovvero si specifica l'obiettivo dell'interrogazione e non il modo in cui ottenerlo. In ciò SQL segue i principi del calcolo relazionale e si contrappone a linguaggi di interrogazione procedurali, come l'algebra relazionale, in

cui l'interrogazione specifica i passi da compere per estrarre le informazioni dalla base di dati. Per essere eseguita l'interrogazione SQL viene passata all'ottimizzatore di interrogazioni, un componente del DBMS il quale analizza l'interrogazione e formula a partire da questa un'interrogazione equivalente nel linguaggio procedurale interno del sistema. Questo linguaggio procedurale è nascosto all'utente, che può trascurare gli aspetti di traduzione e ottimizzazione.

Interrogazioni semplici

Le operazioni di interrogazione in SQL vengono specificate per mezzo dell'istruzione select.

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

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

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

Il risultato di una interrogazione SQL è così una tabella con una riga per ogni riga selezionata dalla clausola **`where`** e le cui colonne si ottengono dalla valutazione delle espressione *AttrEspr* che appaiono nella target list.
La target list specifica gli elementi dello schema della tabella risultato. Come target list può anche comparire il carattere speciale ∗, che rappresenta la selezione di tutti gli attributi delle tabelle elencate nella clausola **`from`**. Nella target list possono comparire generiche espressioni sul valore degli attributi di ciascuna riga selezionata.

Quando si desidera formulare un'interrogazione che coinvolge righe appartenenti a più di una tabella, so pone come argomento della clausola **`from`** l'insieme di 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. Per identificare le tabelle da cui vengono estratti gli attributi si usa il punto per identificare le tabelle da cui vengono estratti gli attributi.

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

La clausola where ammette come argomento una espressione booleana costruita combinando predicati semplici con gli operatori and, or e not. 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 di caratteri arbitrari.

Per selezionare i termini con valori nulli SQL fornisce il predicato **`is null`** (e **`is not null`**) , la cui sintassi è

Attributo **`is`** [**`not`**] **`null`**

Una significativa differenza tra SQL e algebra relazionale è data dalla gestione dei duplicati. 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 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.

Join interni ed esterni

Una sintassi alternativa per la specifica 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.
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.
Il parametro *TipoJoin* specifica qual è il tipo di join da usare, e ad esso si possono sostituire i termini **`inner`**, **`right outer`**, **`left outer`** o **`full outer`**. L'**`inner join`** rappresenta il tradizionale theta-join dell'algebra relazionale.
Con il join interno le righe che vengono coinvolte nel join sono in generale un sottoinsieme delle righe di ciascuna tabella. Può infatti capitare che alcune righe non vengano considerate in quanto non esiste una corrispondente riga nell'altra tabella per cui la condizione sia soddisfatta.
Il join esterno (outer join) esegue un join mantenendo però tutte le righe che fanno parte di una o entrambe le tabelle coinvolte.
Esistono appunto tre varianti dei join esterni: **`left`**, **`right`** e **`full`**. Il **`left join`** fornisce come risultato il join interno esteso con le righe della tabella che compare a sinistra per le quali non esiste una corrispondente riga nella tabella di destra; il **`right join`** si comporta in modo simmetrico; infine, il **`full join`** restituisce il join interno esteso con le righe escluse di entrambe le tabelle.
In alcune implementazioni di SQL si rappresenta il join esterno aggiungendo all'identificativo degli attributi un particolare carattere o sequenza di caratteri (ad esempio * o (+)). In questo modo diventa possibile formulare il join esterno senza ricorrere alla sintassi che abbiamo visto. Queste soluzioni sono però al di fuori dello standard SQL e non sono perciò portabili da un sistema all'altro.
Un'ulteriore estensione di SQL-2 permette di far precedere a ogni join la parola chiave natural. In questo modo si consente la specifica del join naturale dell'algebra relazionale, che prevede di utilizzare nel join di due tabelle una condizione implicita di uguaglianza su tutti gli attributi caratterizzati dallo stesso nome.
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, oltre che per altre ragioni.
Per prima cosa, utilizzando gli alias è possibile far riferimento a più esemplari della stessa tabella, in modo analogo all'uso dell'operatore di ridenominazione dell'algebra relazionale. Tutte le volte che si introduce un alias per una tabella dichiara in effetti una variabile di tipo tabella, che possiede come valore il contenuto della tabella di cui è alias.
La definizione di alias risulta anche molto importante per la realizzazione di sofisticate interrogazioni nidificate.
Ordinamento.

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

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 di attributo, si considerano i valori degli attributi successivi, in sequenza. L'ordine su ciascun attributo può essere ascendente o discendente, a seconda che si sia usato il qualificatore **asc** o **desc**. Se il qualificatore è omesso, si assume un ordinamento ascendente.

Operatori aggregati

Gli operatori aggregati costituiscono una delle più importanti estensioni di SQL rispetto all'algebra relazionale.
In algebra relazionale tutte le condizioni vengono valutate su una tupla alla volta: la condizione è sempre un predicato che viene valutato su ciascuna tupla indipendentemente da tutte le altre. Spesso però si devono valutare delle proprietà che dipendono da insiemi di tuple. Ad esempio il numero di tuple. Gli operatori aggregati vengono gestiti come un'estensione delle normali interrogazioni.
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.*
Gli altri quattro operatori aggregati invece ammettono come argomento un attributo o un'espressione, eventualmente preceduta dalla parola 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** possono applicare anche su stringhe di caratteri o su istanti di tempo.

< **sum** | **max** | **min** | **avg** > ([**distinct** | **all** | *AttrEspr*)

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

- **sum**: restituisce la somma di valori posseduti dall'espressione costruita sui nomi degli attributi;
- **max** e **min**: restituiscono rispettivamente il valore massimo e minimo;
- **avg**: restituisce la media dei valori dell'espressione.

Possiamo anche valutare diversi operatori aggregati nell'ambito della stessa interrogazione. La sintassi SQL non ammette che nella stessa target list compaiano funzioni aggregate ed espressioni al livello di riga, eccetto per le interrogazioni che fanno suo della clausola **group by**.

Interrogazioni con raggruppamento

Molto spesso sorge l'esigenza di applicare l'operatore aggregato distintamente a sottoinsiemi di righe. SQL mette a disposizione la clausola group by, che permette di specificare come dividere le tabelle in sottoinsiemi.

```
select Dipart, sum(Stipendio)
from Impiegato
group by Dipart
```

Per prima cosa l'interrogazione viene eseguita come se la clausola **group by** no 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 l'aggregazione.
La sintassi SQL impone che, in una interrogazione che fa uso della clausola **`group by`**, possa comparire come argomento della **`select`** solamente un sottoinsieme degli attributi usati nella clausola **`group by`**.
Predicati sui gruppi.
Abbiamo visto come tramite la clausola **`group by`** le righe possano venire raggruppate in sottoinsiemi. Una applicazione può aver bisogno di considerare solo i sottoinsiemi che soddisfano certe condizioni. Per le 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 una interrogazione che fa uso della clausola **`group by`**.
La sintassi permette anche la definizione di interrogazioni con clausola **`having`** senza una corrispondente clausola **`group by`**. In questo caso, l'intero insieme di righe è trattato come un unico raggruppamento, ma questo ha in generale un limitato campo di applicabilità, perché se la condizione non è soddisfatta il risultato sarà vuoto. Come la clausola **`where`**, anche la clausola **`having`** ammette come argomento una espressione booleana su predicati semplici.

Interrogazioni di tipo insiemistica

SQL mette a disposizione anche degli operatori insiemistica, 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. Ogni interrogazione che faccia uso degli operatori intersect ed except può essere espressa utilizzando altri costrutti del linguaggio.

SelectSQL { < **`union`** | **`intersect`** | **`except`** > [**`all`**] *SelectSQL* }

Gli operatori insiemistici, al contrario del resto del linguaggio, assumono come default di eseguire una eliminazione dei duplicati. Ci sono due ragioni che giustificano questa differenza: in primo luogo, l'eliminazione dei duplicati rispetta molto meglio il tipico significato di questi operatori; in secondo luogo, l'esecuzione di queste operazioni richiede di effettuare un'analisi delle righe che rende molto limitato il costo aggiuntivo della eliminazione dei duplicati.
Un'altra osservazione è che SQL non richiede che gli schemi su cui vengono effettuate le operazioni insiemistiche siano identici, 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.

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 una 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.*
La soluzione offerta da SQL consiste nell'estendere, con le parole chiave all o any, i normali operatori di confronto relazionale (=, <>, <, >, <= e >=). La sintassi richiede la compatibilità di dominio tra l'attributo restituito dalla interrogazione nidificata e l'attributo con cui avviene il confronto.

```
select *
from Impiegato
where Dipart = any (select Nome
```

from Dipartimento
where città = 'Firenze')

Un'interrogazione nidificata può anche essere espressa mediante un join tra le tabelle. La scelta tra l'una e l'altra rappresentazione può essere dettata dal grado di leggibilità della soluzione. In casi così semplici non vi sono differenze, ma per interrogazioni più complicate la scomposizione in interrogazioni distinte può migliorare la leggibilità.

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.

Una interpretazione molto semplice ed intuitiva delle interrogazioni nidificate consiste nell'assumere che l'interrogazione nidificata venga eseguita prima di analizzare le righe della interrogazione interna. Il risultato della interrogazione può essere salvato in una tabella temporanea e il controllo sulle righe della interrogazione interna può essere fatto accedendo direttamente al risultato temporaneo.

Talvolta però l'interrogazione nidificata fa riferimento al contesto della interrogazione che la racchiude; tipicamente ciò accade tramite una variabile definita nell'ambito della query più esterna (passaggio di binding). La presenza del meccanismo di passaggio di binding arricchisce il potere espressivo di SQL.

L'interrogazione nidificata è un componente della clausola **`where`** e dovrà anch'essa essere valutata separatamente per ogni riga prodotta nella valutazione della 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 nell'ambito di una query nidificata. Se una interrogazione possiede interrogazioni nidificate allo stesso livello, le variabili introdotte nella clausola **`from`** di una query non potranno essere usate nell'ambito dell'altra query.

Introduciamo ora l'operatore logico **`exists`**. Questo operatore ammette come parametro una interrogazione nidificata e restituisce il valore vero solo se l'interrogazione fornisce un risultato non vuoto.

Manipolazione dei dati in SQL

La parte di Data Manipulation Language comprende i comandi per interrogare e modificare il contenuto della base di dati. I comandi che permettono di modificare la base di dati sono insert, delete ed update.

Inserimento

Il comando di inserimento di righe presenta due sintassi alternative:

`insert into` *NomeTabella* [*ListaAttributi*] < **`values`** (*ListaDiValori*) | *selectSQL* >

La prima forma permette di inserire singole righe all'interno delle tabelle. 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.

```
Insert into ProdottiMilanesi
        ( select codice, descrizione
          from Prodotto
        where LuogoProd = 'Milano')
```

Ciascuna forma del comando possiede uno specifico campo di applicazione. La prima forma è quella tipicamente usata all'interno dei programmi per riempire una tabella con i dati forniti direttamente dagli utenti. La seconda forma permette invece di inserire dati in una tabella a partire da altre informazioni presenti in altre basi 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; se l'inserimento del valore nullo viola un vincolo di *not null* definito sull'attributo, l'inserimento viene rifiutato.

Cancellazione

Il comando **delete** elimina righe dalle tabelle:

delete from *NomeTabella* [**where** *Condizione*]

Quando la condizione argomento della clausola **where** non viene specificata, il comando cancella tutte le righe dalla tabella, altrimenti vengono rimosse solo le righe che soddisfano la condizione. Qualora esista un vincolo di integrità referenziale con politica di **cascade** in cui la tabella viene referenziata, allora la cancellazione di righe dalla tabella può comportare la cancellazione di righe appartenenti ad altre tabelle.
La condizione rispetta la sintassi della select, per cui possono comparire al suo interno anche interrogazioni nidificate che fanno riferimento ad altre tabelle.

Modifica

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*]

Il comando di **update** permette di aggiornare uno o più attributi delle righe di *NomeTabella* che soddisfano l'eventuale *Condizione*. Se la condizione non compare, come al solito si suppone di default un valore vero 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;
2. il risultato di una generica interrogazione SQL;
3. il valore nullo; oppure
4. il valore di default per il dominio.

Altre definizioni dei dati in SQL

Vincoli di integrità generici

Abbiamo visto che SQL permette di specificare un certo insieme di vincoli sugli attributi e sulle tabelle, soddisfacendo le più importanti esigenze, ma senza esaurire i bisogni delle applicazioni. Per specificare ulteriori vincoli, SQL-2 ha introdotto la clausola **check**, con la seguente sintassi:

check (*Condizione*)

Le condizioni che si possono usare sono quelle che possono apparire come argomento della clausola **where** di una interrogazione SQL. La condizione deve essere sempre verificata affinché la base di dati sia corretta.
Una efficace dimostrazione della potenza del costrutto consiste nel far vedere come i vincoli predefiniti possano tutti essere descritti con la clausola **check**.
In primo luogo, i vincoli predefiniti permettono una rappresentazione molto più compatta e leggibile. Con la clausola **check** si perde anche la possibilità di associare ai vincoli una politica di reazione alle violazioni.

Asserzioni

Grazie alla clausola check è possibile definire anche un ulteriore componente dello schema di una base di dati, le *asserzioni*. Le asserzioni, introdotte in SQL-2, rappresentano dei vincoli che non sono però associati a nessun attributo o tabella in particolare, ma appartengono direttamente allo schema.
Le asserzioni permettono poi di esprimere vincoli che non sarebbero altrimenti definibili, come vincoli su più tabelle o vincoli che richiedono che una tabella abbia una cardinalità minima.

create assertion *NomeAsserzione* **check** (*Condizione*)

Esempio:

```
create assertion AlmenoUnImpiegato
        check (1 <= (select count(*) from Impiegato))
```

Ogni vincolo d'integrità, definito tramite **check** o tramite asserzione, è associato a una politica di controllo che specifica se il vincolo è immediato o differito. I vincoli immediati sono verificati immediatamente dopo ogni modifica della base di dati, mentre i vincoli differiti sono verificati solo al termine dell'esecuzione di una serie di operazioni.
Tutti i vincoli predefiniti (not null, unique e primary key, foreign key) sono verificati in modo immediato e la loro violazione causa un rollback parziale (l'operazione di modifica appena eseguita che viola il vincolo viene disfata).
Quando invece si rileva una violazione di un vincolo differito al termine della transazione, non c'è modo di individuare l'operazione che ha causato la violazione, e perciò viene disfata l'intera sequenza di operazione (rollback).
E' possibile cambiare il tipo di controllo associato al vincolo, assegnandogli la modalità immediata o differita Ciò avviene tramite i comandi **set constraints** [*NomeVincolo*] **immediate** e **set constraints** [*NomeVincolo*] **deferred**.

Viste

Le viste vengono definite in SQL associando un nome ed una lista di attributi al risultato dell'esecuzione di una interrogazione. La sintassi SQL non ammette però dipendenze ricorsive, né immediate, né transitive.

create view *NomeVista* [(*ListaAttributi*)] **as** *SelectSQL*
[**with** [**local** | **cascaded**] **check option**]

L'interrogazione SQL deve restituire un insieme di attributi pari a quelli contenuti nello schema.

```
create view ImpiegatiAmmin(Matricola, Nome, Cognome, Stipendio) as
  select Matricola, Nome, Cognome, Stipendio
  from Impiegato
  where Dipart = ' Amministrazione' and
        stipendio > 10
```

Su certe viste è permesso effettuare operazioni di modifica, che verranno tradotte negli opportuni comandi di modifica al livello delle tabelle di base da cui la vista dipende.
I sistemi commerciali tipicamente considerano una vista aggiornabile solo se è definita su una sola tabella; qualche sistema chiede pure che l'insieme di attributi della vista contenga almeno una chiave primaria della tabella. La clausola **check** optino può essere utilizzata solo nel contesto di queste viste. Essa specifica che possono essere ammessi aggiornamenti solo sulle righe della vista, e dopo gli aggiornamenti le righe devono continuare ad appartenere alla vista.
Nel caso in cui una vista sia definita in termini di altre viste, l'opzione **local** o **cascaded** specifica se il controllo sul fatto che le righe vengono rimosse dalla vista debba essere effettuato solo all'ultimo livello o se deve essere propagato a tutti i livelli di definizione.

Controllo dell'accesso

La presenza di meccanismi di protezione dei dati riveste grande rilevanza in molte applicazioni. Uno dei compiti più importanti di un amministratore di basi di dati consiste nello scegliere ed

implementare opportune politiche di controllo di accesso. SQL prevede innanzitutto che ogni utente sia identificato in modo univoco dal sistema. L'identificazione dell'utente può sfruttare le funzionalità del sistema operativo o essere indipendente.

Risorse e privilegi

Le risorse che il sistema protegge sono normalmente tabelle, ma si può proteggere un qualsiasi componente del sistema, come gli attributi di una tabella, viste e domini. Di regola l'utente che crea la risorsa ne è il proprietario ed è autorizzato a compiere su di essa qualsiasi operazione. SQL offre meccanismi di gestione flessibili, mediante i quali è possibile specificare quali sono le risorse cui devono accedere gli utenti e quali sono invece le risorse che devono essere mantenute private. Il sistema basa il controllo di accesso su un concetto di privilegio. Gli utenti possiedono ei privilegi d accesso alle risorse del sistema.
Ogni privilegio è caratterizzato dai seguenti parametri:

1. la risorsa cui si riferisce;
2. l'utente che concede il privilegio;
3. l'utente che riceve il privilegio;
4. l'azione che viene premessa sulla risorsa;
5. se il privilegio può essere trasmesso o meno ad altri utenti.

Quando una risorsa viene creata, il sistema concede automaticamente tutti i privilegi su tale risorsa al creatore. I privilegi disponibili sono:

- **`insert`** : permette di inserire un nuovo oggetto nella risorsa;
- **`update`** : permette di aggiornare il valore di un oggetto;
- **`delete`** : permette di rimuovere oggetti dalla risorsa;
- **`select`** : permette di leggere la risorsa, ovvero utilizzarla nell'ambito di una interrogazione;
- **`references`** : permette che venga fatto un riferimento a una risorsa nell'ambito della definizione dello schema di una tabella. Con il privilegio references l'utente cui è concesso il privilegio può definire un vincolo di foreign key, richiedendo che un attributo della propria tabella abbia valori contenuti tra le chiavi della tabella referenziata;
- **`usage`** : permette che venga usata la risorsa, ad esempio nell'ambito della definizione dello schema di una tabella, ma solo per risorse come i domini;

Il privilegio di effettuare un **`drop`** o un **`alter`** di un oggetto non può essere concesso, ma rimane di competenza del creatore dell'oggetto stesso. I privilegi vengono concessi o revocati tramite le istruzioni **`grant`** e **`revoke`**.

Comandi per concedere e revocare privilegi

La sintassi del comando **`grant`** è la seguente:

`grant` *Privilegi* **`on`** *Risorsa* **`to`** *Utenti* [**`with grant option`**]

Il commando permette di concedere i *Privilegi* sulla *Risorsa* agli *Utenti.*

Il comando **`revoke`** fa invece l'inverso: sottrai a un utente i privilegi che gli erano stati concessi:

`revoke` *Privilegi* **`on`** *Risorsa* **`from`** *Utenti* [**`restrict | cascade`**]

Tra i privilegi che possono essere rimossi, oltre a quelli che possono comparire come argomento del commando di **`grant`**, vi è pure il privilegio **`grant option`**, derivante dall'uso dell'opzione **`with grant option`**.
L'unico utente che può sottrarre privilegi ad un altro utente è l'utente che aveva concesso i privilegi in primo luogo. Il comando **`revoke`** può eliminare tutti i privilegi che erano stati concessi, o limitarsi a revocarne un sottoinsieme.
L'opzione **`restrict`** è il valore di default e specifica che il comando non deve essere eseguito qualora la revoca dei privilegi all'utente comporti qualche altra revoca di privilegi.

5. Metodologie e modelli per il progetto

In questo capitolo affrontiamo il problema della progettazione di basi di dati da un punto di vista generale e proponiamo alcuni strumenti di lavoro.
La metodologia di riferimento è articolata in tre fasi: la *progettazione concettuale*, la *progettazione logica* e la *progettazione fisica*.

Introduzione alla progettazione

Il 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 i costi delle varie alternative possibili, le priorità di realizzazione;
- **Raccolta e analisi dei requisiti**: individuare e studiare le proprietà e funzionalità che il sistema informativo dovrà avere; Questa fase richiede una interazione con gli utenti del sistema basata su uno studio preliminare su: unità organizzative omogenee che utilizzeranno il sistema e individuazione delle attività che devono essere supportate dal sistema; un piano di sviluppo del sistema con priorità e tempi di realizzazione; studio di fattibilità che stimi i costi in termini di budget, di impegno del personale e le inefficienze temporanee dovute al cambio di sistema e di modalità di lavoro. Questa fase prevede di individuare le proprietà e le funzionalità del sistema producendo anche una descrizione informale dei dati coinvolti e delle operazioni su di essi. Per ottemperare al meglio a questa fase è necessario: scegliere termini specifici evitando sinonimi e omonimi, standardizzare le frasi, evitare frasi contorte e rendere espliciti i riferimenti. E' necessario inoltre individuare il linea di massima i requisiti software ed hardware del sistema.
- **Progettazione**: si divide in progettazione dei dati e progettazione delle applicazioni. Nella prima si individuano la struttura e l'organizzazione che i dati dovranno avere, nell'altra si definiscono le caratteristiche dei programmi applicativi; Possono procedere in parallelo o in cascata;
- **Implementazione**: realizzare il sistema informativo secondo la struttura e le caratteristiche definite nella fase di progettazione; viene costruita la base di dati e prodotto il codice dei programmi;
- **Validazione e collaudo**: verificare il corretto funzionamento e la qualità del sistema, verificando correttezza dei dati, delle applicazioni, dei tempi di risposta nelle varie condizioni operative, verificando la sicurezza dei dati e la resistenza ai guasti;
- **Funzionamento**: il sistema diventa operativo ed esegue i compiti per il quale è stato predisposto; se non si verificano malfunzionamenti o revisioni delle funzionalità del sistema, questa attività richiede solo operazioni di gestione e manutenzione.

Il processo non è quasi mai strettamente sequenziale, spesso durante l'esecuzione di una delle attività citate, bisogna rivedere decisioni prese nelle attività precedenti.
Le basi di dati costituiscono in effetti solo una delle componenti di un sistema informativo che tipicamente include anche i programmi applicativi, le interfacce con l'utente e altri programmi di servizio.

Metodologie di progettazione e basi di dati

La 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 in ingresso e uscita delle varie fasi.

Le proprietà che una metodologia deve garantire sono:

- La *generalità* rispetto alle applicazione e ai sistemi in gioco;

- La *qualità del prodotto* in termini di correttezza, completezza ed efficienza rispetto alle risorse impiegate;
- La *facilità d'uso* sia delle strategie sia dei modelli di riferimento.

Nell'ambito delle basi di dati si è consolidata una metodologia di progetto che ha dato prova di soddisfare le proprietà descritte. Tale metodologia è articolata in tre fasi principali da effettuare in cascata e si fonda su un principio dell'ingegneria semplice ma molto efficace: separare in maniera netta le decisioni relative a "cosa" rappresentare in una base di dati da quelle relative a "come" farlo.

- **Progettazione concettuale**. Il suo scopo è 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. In questa fase viene prodotto uno *schema concettuale* e fa riferimento ad un *modello concettuale* dei dati. In questa fase, il progettista deve cercare di rappresentare il contenuto informativo della base di dati, senza preoccuparsi delle modalità con le quali queste informazioni verranno codificate;
- **Progettazione logica**. Consiste nella traduzione dello schema concettuale nel modello di rappresentazione dei dati adottato dal sistema di gestione. Viene prodotto uno *schema logico* della base di dati e fa riferimento a un *modello logico* dei dati, ancora secondo una rappresentazione indipendente da dettagli fisici. Si fa comunemente uso anche di tecniche formali di verifica della qualità dello schema logico ottenuto; nel caso del modello relazionale dei dati, la tecnica utilizzata di solito è quella della *normalizzazione*.
- **Progettazione fisica**. Lo schema logico viene completato con la specifica dei parametri fisici di memorizzazione dei dati; viene prodotto uno *schema fisico* e fa riferimento a un *modello fisico* dei dati che dipende dallo specifico sistema di gestione scelto.

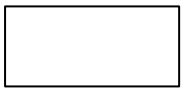
E' bene fare una distinzione tra *specifiche sui dati*, che riguardano il contenuto della base di dati, e *specifiche sulle operazioni*, che riguardano l'uso che utenti e applicazioni fanno della base di dati.
Nella progettazione concettuale si fa uso soprattutto delle specifiche sui dati, mentre le specifiche sulle operazioni servono solo a verificare che lo schema concettuale sia completo.
Nella progettazione logica lo schema concettuale in ingresso riassume le specifiche sui dati, mentre le specifiche sulle operazioni si utilizzano per ottenere uno schema logico che renda tali operazioni eseguibili in maniera efficiente.
Nella progettazione fisica si fa uso dello schema logico e delle specifiche sulle operazioni per ottimizzare le prestazioni del sistema.

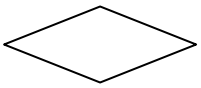
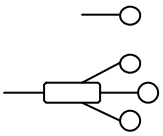
Il risultato della progettazione di una base di dati non è solo lo schema fisico, ma è costituito anche dallo schema concettuale e dallo schema logico. Lo schema concettuale fornisce infatti una rappresentazione della base di dati di alto livello, mentre lo schema logico fornisce una descrizione concreta del contenuto della base di dati.
A partire dai requisiti rappresentati da documenti e moduli di vario genere, acquisiti anche attraverso l'interazione con gli utenti, viene costruito uno schema Entità-Relazione che descrive a livello concettuale la base di dati. Questa rappresentazione viene tradotta poi in uno schema relazionale, costituito da una collezione di tabelle. Infine i dati vengono descritti da un punto di vista fisico e vengono specificate strutture ausiliarie, come gli indici per l'accesso efficiente ai dati.

Il modello Entità-Relazione

Il modello Entità-Relazione (E-R) è un modello *concettuale* di dati e fornisce una serie di strutture, detti *costrutti* atte a descrivere la realtà di interesse in una maniera facile da comprendere. Questi costrutti vengono utilizzati per definire schemi che descrivono l'organizzazione e la struttura delle *occorrenze* dei dati, ovvero, dei valori assunti dai dati nel variare del tempo.

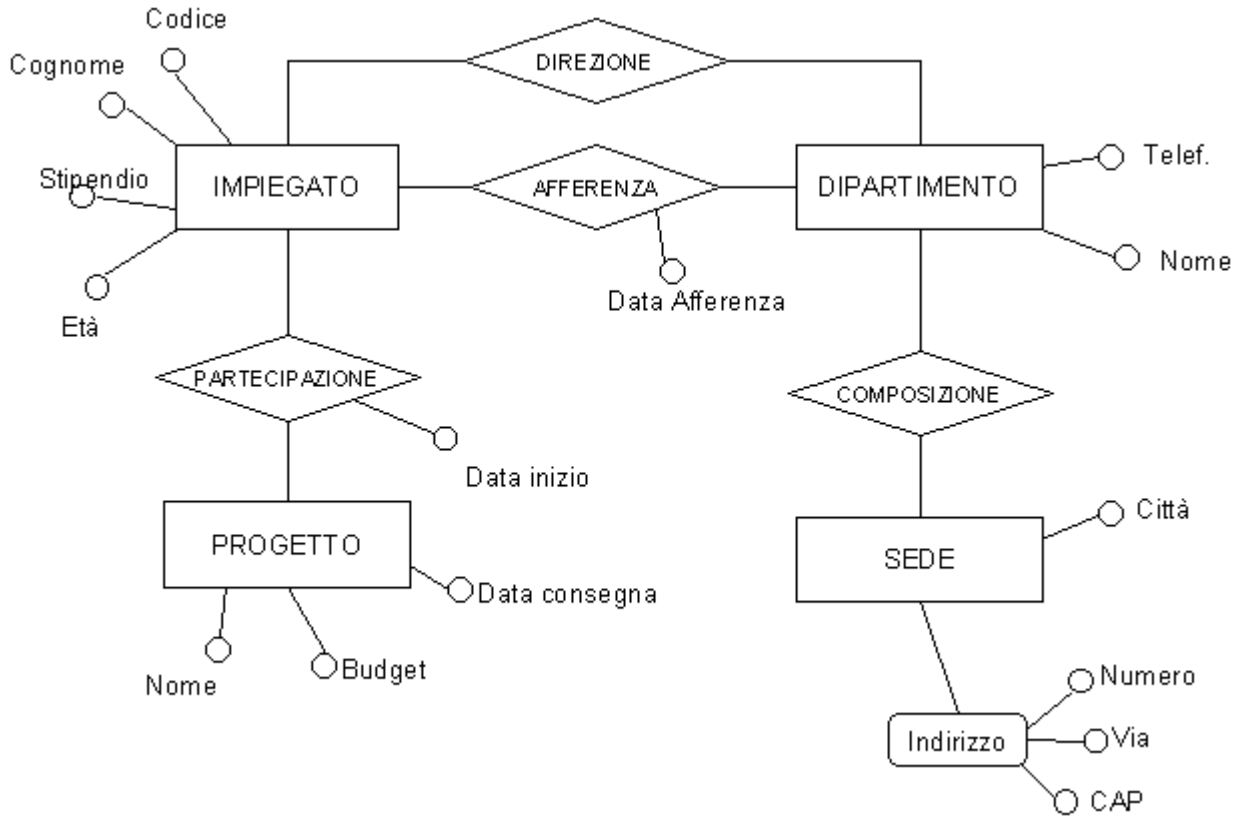
I costrutti principali del modello

Entità 	Rappresentano classi di oggetti che hanno proprietà comuni ed esistenza "autonoma" ai fini dell'applicazione di interesse. Un'occorrenza di una entità è un oggetto che della classe che l'entità rappresenta. In questo il modello E_R, rispetto al modello relazionale, consente di rappresentare un oggetto senza conoscerne le proprietà. Es: CITTÀ, IMPIEGATO, ACQUISTO

Relazioni 	Rappresentano legami logici, significativi per l'applicazione di interesse, tra due o più entità. Un'occorrenza di relazione è una ennupla costituita da occorrenze di entità, una per ciascuna delle entità coinvolte. Es: RESIDENZA è una relazione fra CITTÀ e IMPIEGATO 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. Un aspetto molto importante è il seguente: l'insieme delle occorrenze di una relazione del modello E-R è, a tutti gli effetti, una relazione matematica tra le occorrenze delle entità coinvolte, ovvero, è un sottoinsieme del loro prodotto cartesiano. Questo indica che non ci possono essere ennuple ripetute. E' anche possibile avere relazioni *ricorsive*, ovvero relazioni tra una entità e se stessa. In questo caso è necessario stabilire i due ruoli che l'entità coinvolta gioca nella relazione. Questo può essere fatto associando degli identificatori alle linee uscenti della relazione ricorsiva. E' possibile infine avere relazioni che coinvolgono più di due entità.
Attributi semplici e composti 	Descrivono le proprietà elementari di entità o relazioni che sono di interesse ai fini dell'applicazione. Un attributo 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. I domini non vengono riportati sullo schema ma sono descritti nella documentazione associata. Può risultare comodo, qualche volta, 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*.

Costruzione di schemi con i costrutti di base

I tre costrutti del modello E_R visti fino ad ora consentono già di costruire schemi per descrivere realtà di una certa complessità.



Uno schema Entità-Relazione

Altri costrutti del modello

Esaminiamo ora i rimanenti costrutti del modello E-R.

Cardinalità delle relazioni

Vengono specificate per ciascuna partecipazione di entità a una relazione e descrivono il numero minimo e massimo di occorrenze di relazione cui una occorrenza dell'entità può partecipare. Dicono quindi quante volte, in una relazione tra entità, un'occorrenza di una di queste entità può essere legata a occorrenze delle altre entità coinvolte.

In linea di principio è possibile assegnare un qualunque intero non negativo a una cardinalità di una relazione, con l'unico vincolo, che la cardinalità minima deve essere minore o uguale alla cardinalità massima. In realtà, nella maggior parte dei casi, è sufficiente utilizzare solo tre valori: zero, uno e il simbolo N (che indica genericamente un intero maggiore di uno).
In particolare:

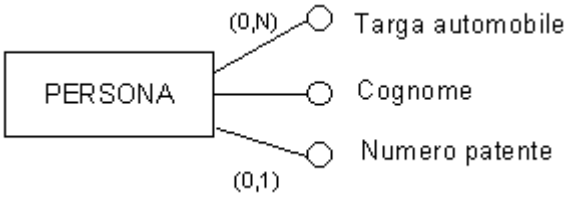
- Per la cardinalità minima, zero o uno: nel primo caso si dice che la partecipazione è opzionale, nel secondo è obbligatoria;
- Per la cardinalità massima, uno o molti (N): nel primo caso la partecipazione dell'entità relativa può essere vista come una funzione che associa a una occorrenza dell'entità una sola occorrenza dell'altra entità; nel secondo c'è invece 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 cardinalità coinvolte definiscono una *relazione uno a uno*. Le relazioni aventi un'entità con cardinalità massima pari a 1 e l'altra pari a N sono denominate *relazioni uno a molti.* Infine le relazioni aventi cardinalità massima pari a N per entrambe le entità si definiscono *relazioni molti a molti.*

Per le cardinalità minime va detto invece che il caso di partecipazione obbligatoria per tutte le entità coinvolte è piuttosto raro.

Cardinalità degli attributi

Così come per le relazioni, possono essere specificate anche cardinalità per gli attributi, sia delle entità che delle relazioni, che descrivono il numero minimo e massimo di valori dell'attributo associati ad ogni occorrenza. Nella maggior parte dei casi la cardinalità è pari a (1,1) e viene omessa. Il valore per certi attributi può però a volte essere nullo, oppure possono esistere diversi valori di un certo attributo per una occorrenza di entità.

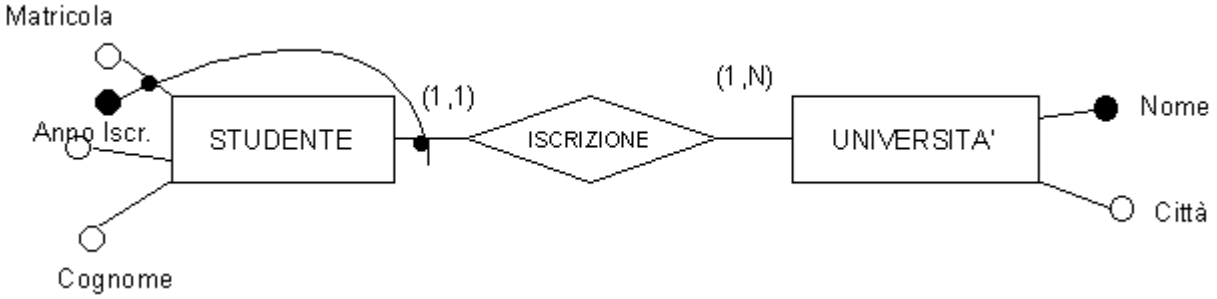


In maniera simile alle partecipazioni le occorrenze di entità alle relazioni, diremo che un attributo con cardinalità minima pari a zero è *opzionale* per la relativa entità o relazione, mentre è *obbligatorio* se la cardinalità minima è pari ad uno. Diremo infine che un attributo è *multivalore* se la sua cardinalità massima è pari a N.

Identificatori delle entità

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

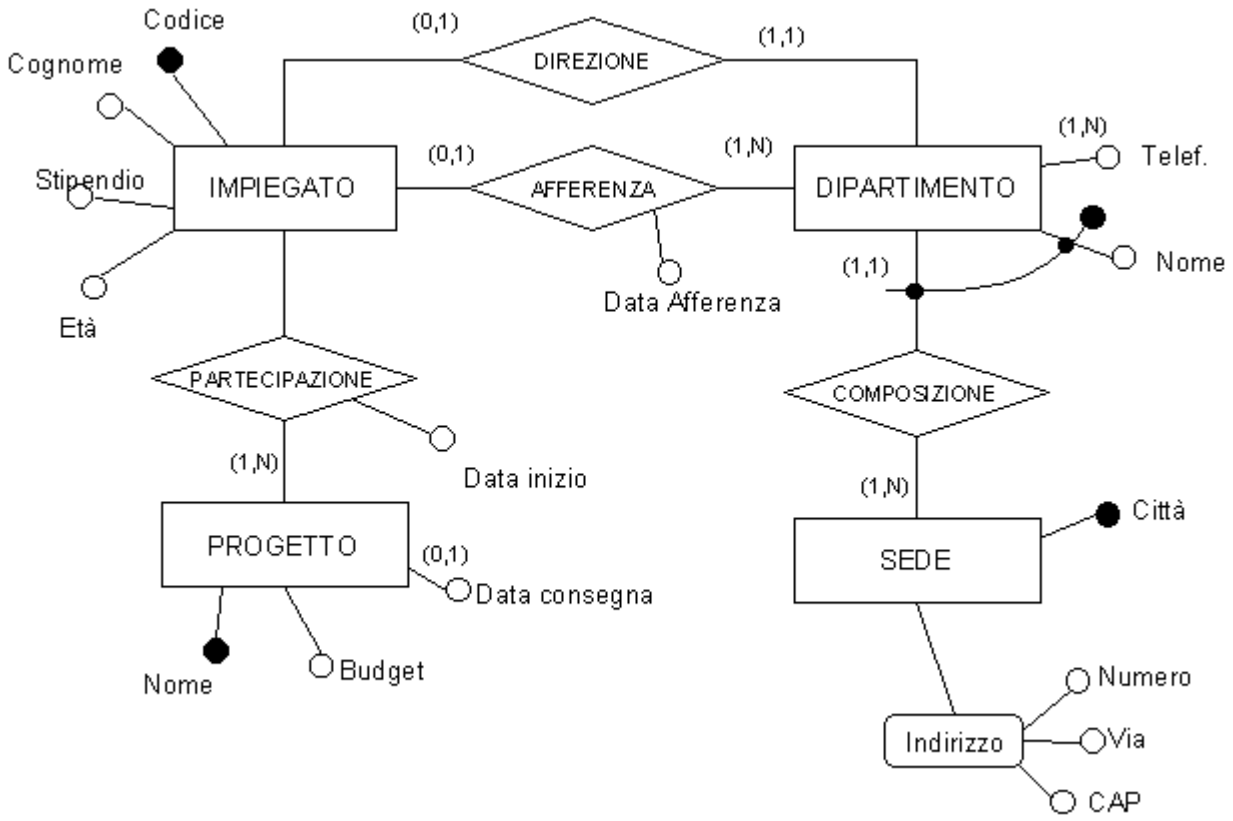
Alcune volte però gli attributi di un'entità non sono sufficienti a identificare univocamente le sue occorrenze. In questi casi un'entità E può essere identificata da altre entità solo se tali entità sono coinvolte in una relazione cui E partecipa con cardinalità (1, 1).Nei casi in cui l'identificazione di una 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 deve avere cardinalità (1,1);
- Una identificazione esterna può coinvolgere una 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, ma ne può avere in generale più d'uno;

Possiamo a questo punto completare il nostro esempio:



Generalizzazioni

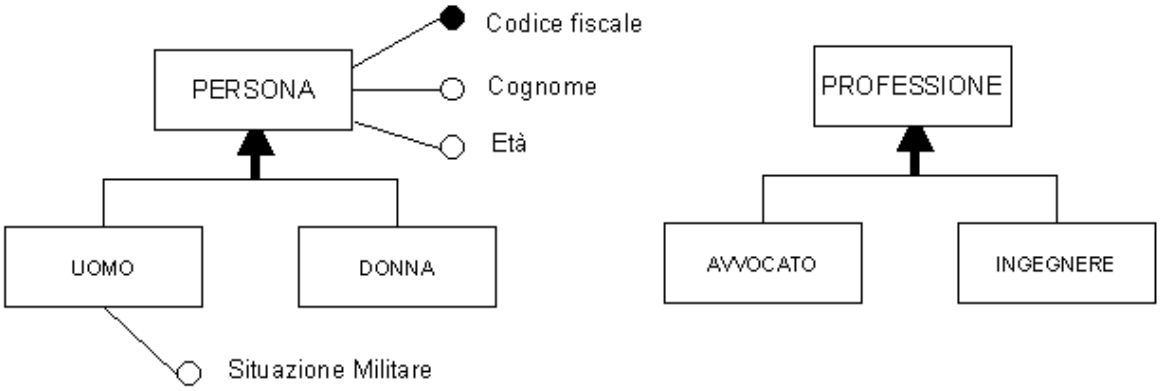
Rappresentano legami logici tra un entità E, detta entità *padre*, e una o più entità $E_1,\ldots,E_n$ dette entità *figlie*. Si dice in questo caso che E è *generalizzazione* di $E_1,\ldots,E_n$ e che le entità $E_1,\ldots,E_n$ sono *specializzazione* dell'entità E.

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

Ogni occorrenza di una entità figlia è anche una occorrenza dell'entità padre.

Ogni proprietà dell'entità padre (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 padre. Per le entità figlie, le proprietà ereditate non vanno rappresentate esplicitamente.



Le generalizzazioni possono essere classificate sulla base di due proprietà tra loro ortogonali:

- una generalizzazione è *totale* se ogni occorrenza della classe padre è una occorrenza di almeno una delle entità figlie, altrimenti è *parziale*;
- una generalizzazione è *esclusiva* se ogni occorrenza della classe padre è al più una occorrenza di una delle entità figlie, altrimenti è *sovrapposta.*

Esempi:
la generalizzazione del primo esempio è totale poiché gli uomini e le donne costituiscono tutte le persone ed esclusiva poiché una persona può essere o uomo o donna, non entrambi. La seconda invece è parziale poiché mancano molte categorie e sovrapposta poiché uno può fare più lavori.

In generale, una stessa entità può esser coinvolta in più generalizzazioni diverse. Possono esserci inoltre generalizzazioni su più livelli: si parla in questo caso di *gerarchia* di generalizzazioni. Infine una generalizzazione può avere solo una entità figlia, parliamo in questo caso di *sottoinsieme.*

6. La progettazione concettuale

La progettazione concettuale di una base di dati consiste nella costruzione di uno schema Entità-Relazione in grado di descrivere al meglio le specifiche sui dati di una applicazione.

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

- Gli utenti della applicazione, attraverso opportune interviste.
- Tutta la documentazione esistente che ha attinenza con il problema allo studio.
- Eventuali realizzazioni preesistenti, ovvero applicazioni che si devono rimpiazzare o che devono interagire in qualche maniera con il sistema da realizzare.

Nella fasi di acquisizione delle specifiche, gioca un importante ruolo l'interazione con gli utenti del sistema informativo. In genere gli utenti a livello più alto possiedono una visione più ampia ma meno dettagliata.
Nel corso delle interviste è opportuno effettuare con l'utente verifiche di comprensione e consistenza sulle informazioni che si stanno raccogliendo. Partendo quindi dai principali aspetti del problema allo studio, si procede cercando di acquisire via via maggiori dettagli.
La specifica dei requisiti raccolti avviene spesso facendo uso di descrizioni in linguaggio naturale. E' molto importante effettuare una profonda analisi del testo che descrive le specifiche per filtrare le eventuali inesattezze e i termini ambigui presenti. E' chiaro che il linguaggio naturale porta a un certo numero di ambiguità e imprecisioni. Proviamo allora a fissare alcune regole generali per ottenere una specifica dei requisiti più precisa e senza ambiguità.

- Scegliere il corretto livello di astrazione
- Standardizzare la struttura delle frasi
- Evitare frasi contorte
- Individuare sinonimi/omonimi e unificare i termini
- Rendere esplicito il riferimento tra termini
- Costruire un glossario dei termini

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 e informarci anche sulla frequenza con la quale le varie operazioni vengono eseguite.

Criteri generali di rappresentazione

Prima di affrontare le metodologie di progetto, cerchiamo di stabilire alcuni criteri generali per tradurre una specifica informale in un costrutto del modello Entità-Relazione. Nel caso della progettazione concettuale conviene, in buona sostanza, 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 una entità.
Se un concetto ha una struttura semplice e non possiede proprietà rilevanti associate, è opportuno rappresentarlo con un attributo di un altro concetto cui si riferisce.
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.
Se uno o più concetti risultano essere casi particolari di un altro, è opportuno rappresentarli facendo uso di una generalizzazione.
I criteri visti hanno validità generale, sono cioè indipendenti dalla strategia di progettazione scelta.

Strategie di progetto

Lo schema concettuale a partire dalle sue specifiche può essere considerato a tutti gli effetti un processo di ingegnerizzazione e, come tale, risultano a esso applicabili le strategie di progetto utilizzate anche in altre discipline.

Strategia top-down

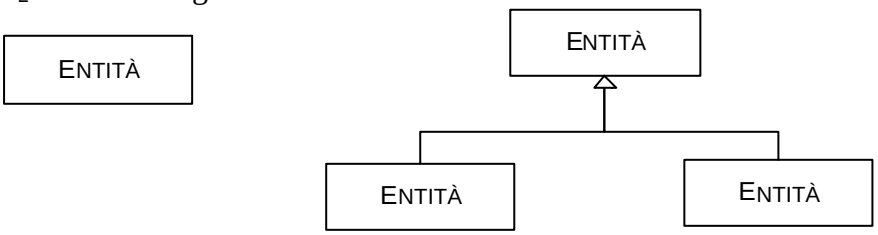
In questa strategia, lo schema concettuale viene prodotto mediante una serie di raffinamenti successivi a partire da uno schema iniziale. Lo schema viene poi via via raffinato mediante opportune trasformazioni che aumentano il dettaglio dei vari concetti presenti.
Nel passaggio da un livello di raffinamento a un altro, lo schema viene modificato facendo uso di alcune trasformazioni elementari che vengono denominate *primitive di trasformazione top-down*. Come si può vedere, queste primitive operano su un singolo concetto dello schema e lo trasformano in una struttura più complessa in grado di descrivere il concetto di partenza con maggior dettaglio.

- Trasformazione T_1: si applica quando si comprende che una entità descrive in realtà due concetti diversi legati logicamente tra di loro.
- Trasformazione T_2: si applica quando si comprende che una entità è composta da sotto-entità distinte.
- Trasformazione T_3: si applica quando si comprende che una relazione descrive in realtà due relazioni diverse tra le medesime entità.
- Trasformazione T_4: si applica quando si comprende che una relazione descrive in realtà un concetto con esistenza autonoma ai fini della applicazione.
- Trasformazione T_5: si applica per aggiungere proprietà (attributi) a entità.
- Trasformazione T_6: si applica per aggiungere proprietà a relazioni, in maniera simile alla trasformazione T5.

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. 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.

T_1: Da entità a relazione tra entità

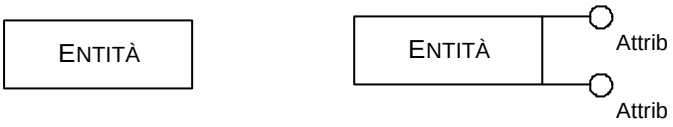
T_2: Da entità a generalizzazione



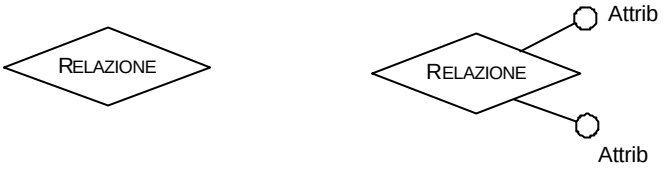
T_3: Da relazione a insieme di relazioni

T_4: Da relazione a entità con relazioni

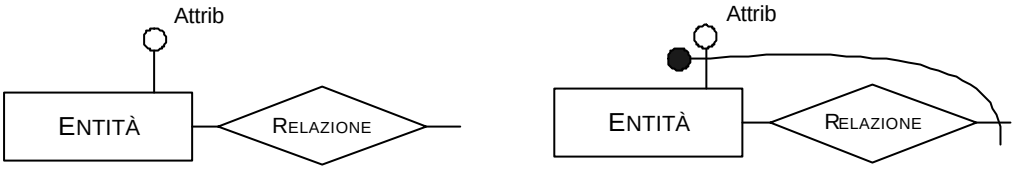
T_5: Introduzione di attributi su entità



T_6: Introd. di attributi su relazioni



T_7: Introduzione di chiavi esterne



Strategia bottom-up

In questa strategia, 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. Questo procedimento viene rappresentato con le seguenti fasi: la fase di decomposizione delle specifiche, la successiva fase di rappresentazione delle componenti di base e la fase finale di integrazione degli schemi elementari.

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

Trasformazione T1: si applica quando si individua nelle specifiche una classe di oggetti con proprietà comuni.

Trasformazione T2: si applica quando si individua nelle specifiche un legame logico tra due entità.

Trasformazione T3: si applica quando si individua nelle specifiche un legame tra diverse entità riconducibile a una generalizzazione.

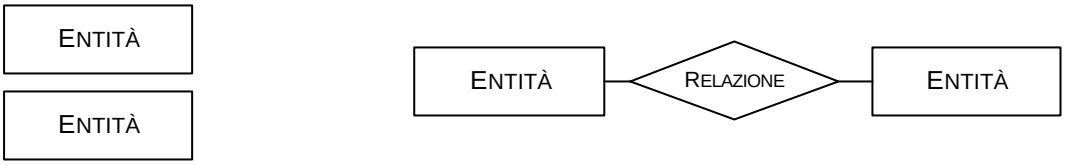
Trasformazione T4: si applica quando, a partire da una serie di attributi, si individua una entità che può essere vista come aggregazione di tali attributi.

Trasformazione T5. si applica in maniera simile alla trasformazione T4, quando si individua una relazione che può essere vista come aggregazione di attributi.

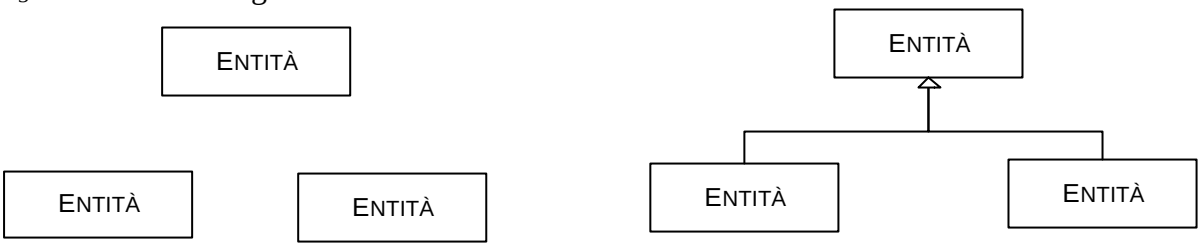
Il vantaggio della strategia è che si adatta a una decomposizione del problema in componenti più semplici, facilmente individuabili, il cui progetto può essere affrontato anche da progettisti diversi. E' quindi una strategia che si presta bene a lavori svolti in collaborazione o suddivisi all'interno 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.

T_1: Generazione di entità

T_2: Generazione di relazione



T_3: Generazione di generalizzazioni



T_4: Aggregazione di attributi su entità

T_5: Aggregazione di attributi su relazione

Strategia inside-out

Questa strategia 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 cioè prima i concetti concettualmente più vicini ai concetti iniziali, per poi muoversi verso quelli più ontani attraverso una navigazione tra le specifiche.
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 tutte le specifiche per individuare concetti non ancora rappresentati e descrivere i nuovi concetti nel dettaglio. Non è quindi possibile procedere per livelli di astrazione come avviene nella strategia top-down.

Strategia mista

La strategia mista 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 della applicazione. Questo schema scheletro fornisce una visione unitaria, sia pure astratta, dell'intero progetto e favorisce le fasi di integrazione degli schemi sviluppati separatamente.

A questo punto possiamo procedere considerando, anche separatamente, questi concetti principali e proseguire per raffinamenti successivi (top-down) oppure estendere il sottoschema con concetti non ancora rappresentati (bottom-up).
La strategia mista è probabilmente la più flessibile tra le strategia viste perché si adatta bene a esigenze contrapposte. In effetti questa strategia ingloba anche la strategia inside-out che, come abbiamo detto, è solo un caso particolare della strategia bottom-up. C'è anche da dire che, in quasi tutti i casi pratici di una certa complessità, la strategia mista è l'unica che si può effettivamente adottare perché, come abbiamo detto all'inizio di questo capitolo, è spesso necessario cominciare la progettazione quando non sono ancora disponibili tutti i dati e, dei dati noti, abbiamo spesso delle conoscenze a livelli di dettaglio non omogenei.

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 costrutti come una generalizzazione tra relazioni. I secondi riguardano un uso di costrutti che non rispetta la loro definizione. La correttezza di uno schema si può verificare per ispezione, confrontando i concetti presenti nello schema in via di costruzione con le specifiche e con le definizioni dei costrutti del modello usato.

Completezza

Uno schema è completo quando rappresenta tutti i dati di interesse e quando tutte le operazioni possono essere eseguite a partire dai concetti descritti nello schema. Si può verificare controllando che tutte le specifiche sui dati siano rappresentate da qualche concetto dello schema.

Leggibilità

Uno schema 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.
Alcuni suggerimenti per rendere uno schema più leggibile sono i seguenti:

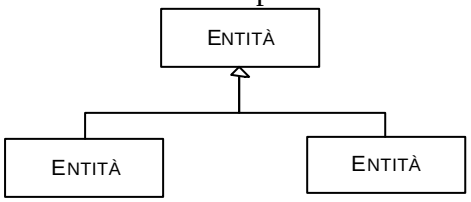
- disporre i costrutti su una griglia scegliendo come elementi centrali quelli con più legami;
- tracciare solo linee perpendicolari e cercare di minimizzare le intersezioni;
- disporre le entità che sono padri di generalizzazioni sopra le relative entità figlie.

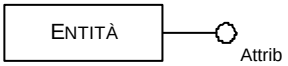
Minimalità

Uno schema è minimale quando tutte le specifiche sui dati sono rappresentate una sola volta nello schema. Non è minimale quando esistono ad esempio delle ridondanze. La minimalità si può verificare per ispezione, controllando se esistono concetti che possono essere eliminati dallo schema.

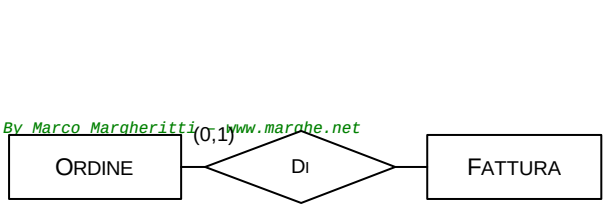
Semplificazioni

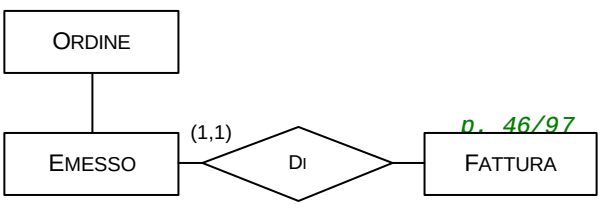
Eliminare le entità pendenti





Eliminare le entità non rappresentative





Aumentare l'autoesplicatività

Una metodologia generale

Non accade quasi mai che un progetto prosegua sempre in maniera top-down o bottom-up. Indipendentemente dalla strategia scelta, nelle situazioni reali capita infatti di modificare lo schema in via di costruzione sia con trasformazioni top-down che con trasformazioni bottom-up. Presentiamo perciò una metodologia per la progettazione concettuale con il modello E-R con riferimento alla strategia mista.

1. Analisi dei requisiti
    a. Costruire un glossario dei termini
    b. Analizzare i requisiti ed eliminare le ambiguità presenti
    c. Raggruppare i requisiti in insiemi omogenei
2. Passo base
    a. Individuare i concetti più rilevanti e rappresentarli in uno schema scheletro
3. Passo di decomposizione (se necessario)
    a. Effettuare una decomposizione dei requisiti con riferimento ai concetti presenti nello schema scheletro
4. Passo iterativo (da ripetere per ogni schema
    a. Raffinare i concetti presenti sulla base delle loro specifiche
    b. Aggiungere nuovi concetti allo schema per descrivere specifiche non ancora descritte
5. Passo di integrazione (a fronte del passo 3)
    a. Integrare i vari sotto-schemi in uno schema generale facendo riferimento allo schema scheletro
6. Analisi di qualità
    a. Verificare la correttezza dello schema ed eventualmente ristrutturare lo schema
    b. Verificare la completezza dello schema ed eventualmente ristrutturare lo schema
    c. Verificare la minimalità, documentare le ridondanze ed eventualmente ristrutturare lo schema
    d. Verificare la leggibilità dello schema ed eventualmente ristrutturare lo schema.

Nella metodologia presentata, viene solo brevemente citata un'importante attività che dovrebbe invece accompagnare tutte le fasi. La documentazione degli schemi.
Va notato che l'analisi della qualità costituisce un importante momento di verifica dello stato corrente del progetto nel quale è spesso necessario dover effettuare delle ristrutturazioni per rimediare a "errori" fatti nelle fasi precedenti.

7. La 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. 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. La semplificazione dello schema si rende necessaria perché non tutti i costrutti del modello E-R hanno una traduzione naturale nei modelli logici.
Pertanto è necessario prevedere sia un'attività di *riorganizzazione*, sia un'attività di *traduzione*. Poiché la riorganizzazione può essere in buona misura discussa indipendentemente dal modello logico, è utile di solito articolare la progettazione logica in due fasi:

- Ristrutturazione dello schema E-R, indipendente 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 a uno specifico modello logico 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. Questo schema e il modello logico scelto costituiscono i dati di ingresso della seconda fase. 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

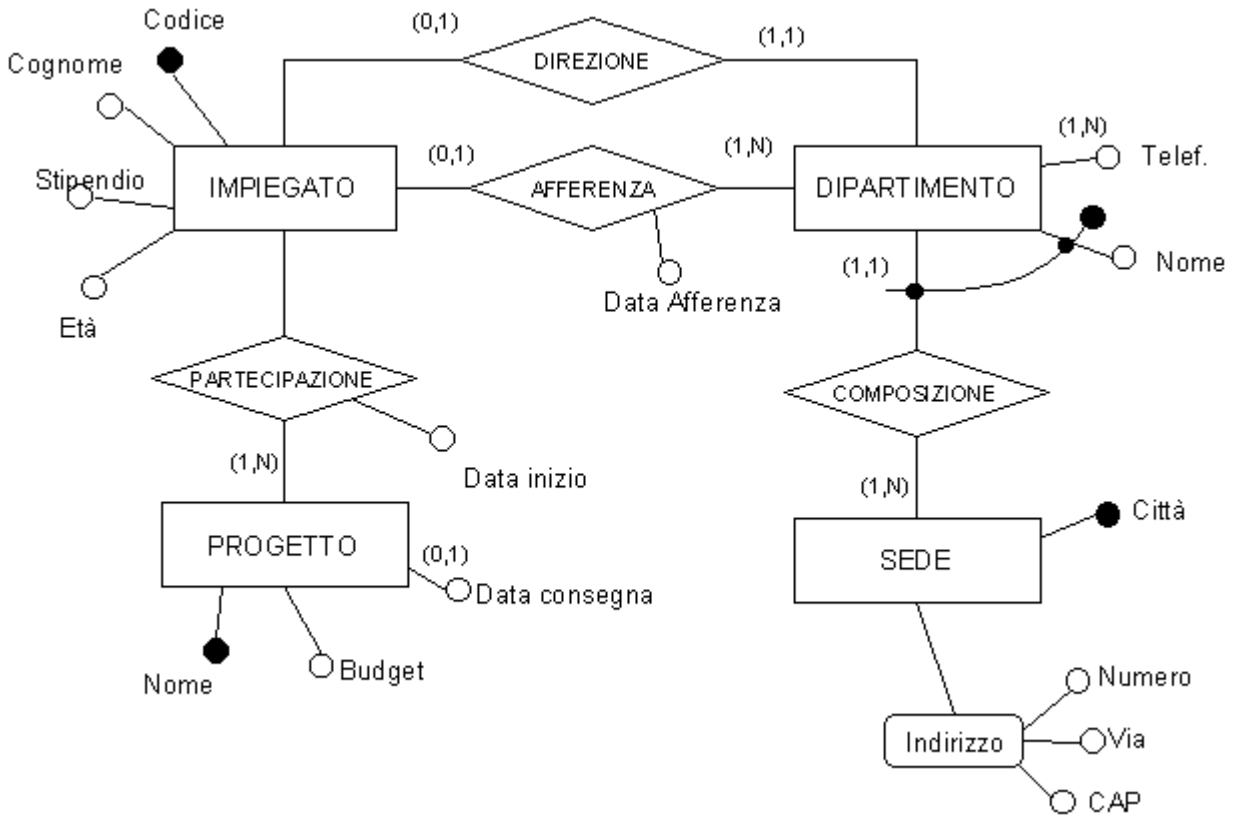
Abbiamo detto che uno schema E-R può essere modificato per ottimizzare alcuni *indici di prestazione* del progetto. E' possibile effettuare studi di massima dei due parametri che generalmente regolano le prestazioni dei sistemi software:

- costo di una operazione: viene valutato in termini di numero di occorrenze di entità e associazioni che mediamente vanno visitate per rispondere a una operazione sulla base di dati;
- occupazione di memoria: viene valutato in termini dello spazio di memoria 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. Ovvero:
  - numero di occorrenze di ogni entità e associazione dello schema;
  - dimensioni di ciascun attributo.
- Caratteristiche delle operazioni. Ovvero:
  - tipo dell'operazione (interattiva o batch);
  - frequenza (numero medio di esecuzioni in un certo intervallo di tempo);
  - dati coinvolti (entità e/o associazioni).

Sebbene una 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". In base a questa regola, l'ottanta percento del carico è generato dal venti per cento delle operazioni. (usiamo come esempio lo schema precedente e qui riportato.)



Il volume dei dati e le caratteristiche generali delle operazioni possono essere descritti facendo uso di tabelle.

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

Tavola delle operazioni

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

Nella *tavola dei volumi* vengono riportati tutti i concetti dello schema con il volume previsto a regime. Nella *tavola delle operazioni* riportiamo, per ogni operazione, la frequenza prevista e un simbolo che indica se l'operazione è interattiva o batch. Nella tavola dei volumi, il numero delle occorrenze delle associazioni dipende da due parametri: il numero di occorrenze delle entità coinvolte nelle associazioni da due parametri: il numero di occorrenze delle entità coinvolte nelle associazioni e il numero di partecipazioni di una occorrenza di entità alle occorrenze di associazioni.

Per ogni operazione, possiamo inoltre 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 di interesse.
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 relazioni necessario per eseguire l'operazione. Il tutto viene riassunto in una *tavola degli accessi.*

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

Nell'ultima colonna di questa tabella viene riportato il tipo di accesso: L per accesso in lettura e S per accesso in scrittura.

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 in più concetti o, viceversa, accorpare concetti separati in un unico concetto.
- Scelta degli identificatori primari. 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 da altri dati. I casi più seguenti sono:

- attributi derivabili, occorrenza per occorrenza, da altri attributi della stessa entità;
- attributi derivabili da attributi di altre entità, di solito attraverso funzioni aggregative;
- attributi derivabili da operazioni di conteggio di occorrenze;
- attributi derivabili dalla composizione di altre associazioni in presenza di cicli.

La presenza di un dato derivato in uno schema E-R 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 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 gerarchie

Dato che il modello relazionale non permette di rappresentare direttamente una generalizzazione, risulta necessario trasformare questo costrutto in altri costrutti del modello E-R per i quali esiste invece una traduzione semplice: le entità e le associazioni.
Per rappresentare una generalizzazione mediante entità e associazioni abbiamo essenzialmente tra alternative possibili.

1. Accorpamento delle figlie della generalizzazione nel padre. Le entità figlie e le loro proprietà vengono eliminate ed aggiunte all'entità padre. A tale entità viene aggiunto un ulteriore attributo che serve a distinguere il "tipo" di un occorrenza.
2. Accorpamento del padre della generalizzazione nelle figlie. L'entità padre viene eliminata e, per la proprietà dell'ereditarietà, i suoi attributi, il suo identificatore e le relazioni cui tale entità partecipava vengono aggiunti alle entità figlie.
3. Sostituzione della generalizzazione con associazioni. La generalizzazione si trasforma in due associazioni uno a uno che legano rispettivamente l'entità padre con le entità figlie. Non ci sono trasferimenti di attributi o associazioni e le entità figlie sono identificate esternamente dall'entità padre. Nello schema ottenuto vanno aggiunti però dei vincoli: ogni occorrenza padre non può partecipare contemporaneamente ad entrambe le relazioni con le entità figlie.

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 alla occupazione di memoria e al costo delle operazioni coinvolte. E' possibile comunque stabilire alcune regole:

- L'alternativa 1) è conveniente quando le operazioni non fanno molta distinzione tra le occorrenze e tra gli attributi delle entità. In questo caso, 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 alternative.
- L'alternativa 2) è possibile solo se la generalizzazione è totale, altrimenti le occorrenze del padre che non sono occorrenze di nessun figlio, non sarebbero rappresentate. E' conveniente quando ci sono operazioni che si riferiscono solo ad un figlio e fanno quindi distinzioni tra di essi. In questo caso abbiamo un risparmio di memoria rispetto a 1) poiché gli attributi non assumono mai valori nulli. Inoltre c'è una riduzione degli accessi rispetto a 3) poiché non si deve visitare il padre per accedere ad alcuni attributi dei figli.
- L'alternativa 3) è conveniente quando la generalizzazione non è totale e ci sono operazioni che si riferiscono solo alle occorrenze di un figlio e del padre. In questo caso abbiamo un risparmio di memoria rispetto alla scelta 1) ma c'è un incremento degli accessi per mantenere la consistenza delle occorrenze rispetto ai vincoli introdotti.

La ristrutturazione delle generalizzazioni è un tipico caso per il quale il semplice conteggio delle istanze e degli accessi non è sempre sufficiente per scegliere la migliore alternativa possibile. Da quanto detto infatti, la terza alternativa sembrerebbe sempre poco conveniente. Questa restrutturazione in realtà ha il grosso vantaggio di generare entità con pochi attributi, questo si traduce in strutture logiche di piccole dimensioni per le quali un accesso fisico permette di recuperare molti dati in una volta sola.
Le alternative viste non sono in effetti le uniche ammesse, ma è possibile effettuare ristrutturazioni che sono combinazioni delle tre trasformazioni presentate.
Per quanto riguarda infine le generalizzazioni su più livelli, si può procedere analogamente, analizzando una generalizzazione alla volta a partire dal fondo dell'intera gerarchia. In base a quanto detto, sono possibili diverse configurazioni, ottenibili per combinazione delle ristrutturazioni di base, sia a livello della singola generalizzazione, sia lungo i vari livelli della gerarchia.

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.

Partizionamenti di entità

Questa ristrutturazione è conveniente se le operazioni che coinvolgono frequentemente l'entità originaria richiedono o solo informazioni di un blocco di attributi o di un altro.

Un partizionamento di questo tipo è un esempio di *decomposizione verticale* di una entità, nel senso che si suddivide il concetto operando sui suoi attributi. E' comunque possibile effettuare anche delle *decomposizioni orizzontali* nelle quali la suddivisione avviene sulle occorrenze dell'entità. E' interessante osservare come una decomposizione orizzontale corrisponda all'introduzione di una generalizzazione a livello logico.
I partizionamenti orizzontali hanno un effetto collaterale: quello di dover duplicare tutte le associazioni cui l'entità originaria partecipa. D'altra parte i partizionamenti verticali generano entità con pochi attributi che possono essere tradotte in strutture logiche sulle quali, con un solo accesso, è possibile recuperare molti dati.

Eliminazione di attributi multivalore

Un particolare tipo di partizionamento che è opportuno trattare a parte è quello di 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.

Accorpamento di entità

L'accorpamento è l'operazione inversa del partizionamento. Questa operazione può essere motivata dal fatto che operazioni più frequenti su un entità richiedano sempre i dati relativi ad un'altra entità e quindi si vuole risparmiare gli accessi necessari per risalire a questi dati attraverso la relazione che li lega. Gli accorpamenti si effettuano in genere su associazioni di tipo uno a un, raramente su associazioni uno a molti e praticamente mai su relazioni molti a molti, poiché queste relazioni creano molte ridondanze. La presenza di ridondanze può essere comunque analizzata e discussa in maniera efficace con la tecnica della *normalizzazione*.

Altri tipi di partizionamento/accorpamento

Abbiamo parlato finora di partizionamento e accorpamento di entità ma lo stesso discorso si può estendere alle associazioni. Può capitare che in alcuni casi decomporre una associazione tra due entità in due o più associazioni tra le medesime entità, per separare occorrenze dell'associazione originale accedute sempre separatamente e, viceversa, accorpare due o più associazioni tra le medesime entità in un'unica associazione.

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. I sistemi di gestione di basi di dati richiedono generalmente di specificare una *chiave primaria* sulla quale vengono costruite automaticamente delle strutture ausiliarie, dette *indici*, per il reperimento efficiente di dati.
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.
- Per gli stessi motivi del punto precedente 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 una entità è da preferire rispetto agli altri.

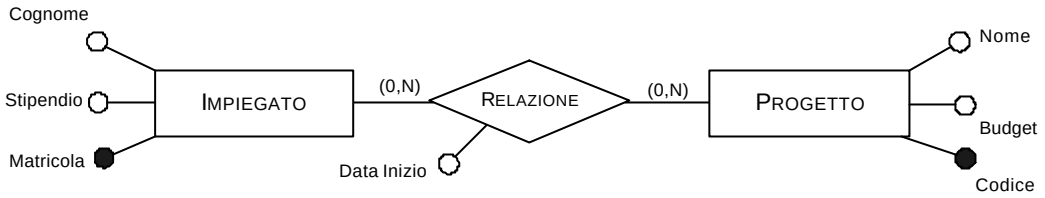
A questo punto, se nessuno degli identificatori candidati soddisfa tali requisiti, è possibile pensare di introdurre un ulteriore attributo all'entità: questo attributo conterrà valori speciali (detti *codici*) generati appositamente per identificare le occorrenze delle entità.

Traduzione verso il modello relazionale

La seconda fase della progettazione logica corrisponde a una traduzione tra modelli di dati diversi: si costruisce uno schema logico *equivalente,* in grado cioè di rappresentare le medesime informazioni.

Entità e associazioni molti a molti

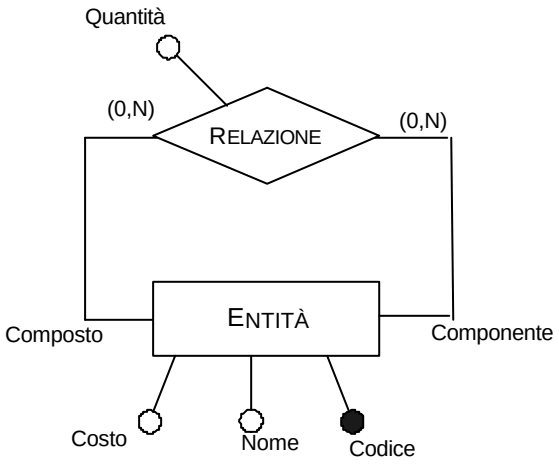
Consideriamo il seguente schema:



La sua traduzione naturale prevede:
per ogni entità, una relazione con lo stesso nome avente per attributi i medesimi attributi dell'entità e per chiave il suo identificatore;
per associazione, una relazione con lo stesso nome avente per attributi gli attributi dell'associazione e gli identificatori delle entità coinvolte; tali identificatori formano la chiave della relazione.
Lo schema relazionale che si ottiene è quindi il seguente:

IMPIEGATO (Matricola, Cognome, Stipendio)
PROGETTO (Codice, Nome, Budget)
PARTECIPAZIONE (Matricola, Codice) , DataInizio)

Per lo schema ottenuto esistono due vincoli di integrità referenziale tra gli attributi Matricole e Codice di PARTECIPAZIONE e gli omonimi attributi di Impiegato e Progetto.
Per rendere più comprensibile il significato dello schema è conveniente effettuare alcune ridenominazioni. Per esempio, nel nostro caso si può chiarire il contenuto della relazione PARTECIPAZIONE definendola come segue:
PARTECIPAZIONE (Matricola, Codice) , DataInizio)
La ridenominazione è a volte essenziale, per esempio nel caso di associazioni ricorsive.

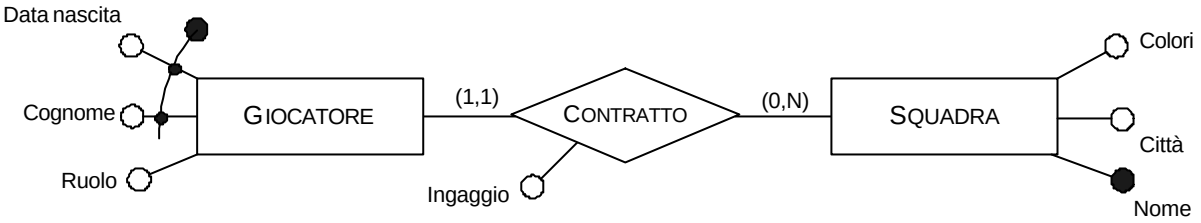


Questo schema si traduce nelle due relazioni:
PRODOTTO (Codice, Nome, Costo)
COMPOSIZIONE (Composto, Componente) , Quantità)
In questo schema entrambi gli attributi Composto e Componente contengono codici di prodotti. Esiste quindi un vincolo di integrità referenziale tra questi attributi e l'attributo Codice della relazione Prodotto.

Le associazioni con più di due entità partecipanti si traducono in maniera analoga a quanto detto per le associazioni binarie.

Associazioni uno a molti

Consideriamo lo schema con associazione uno a molti:



Secondo la regola vista per le associazioni molti a molti, la traduzione di questo schema dovrebbe essere la seguente:

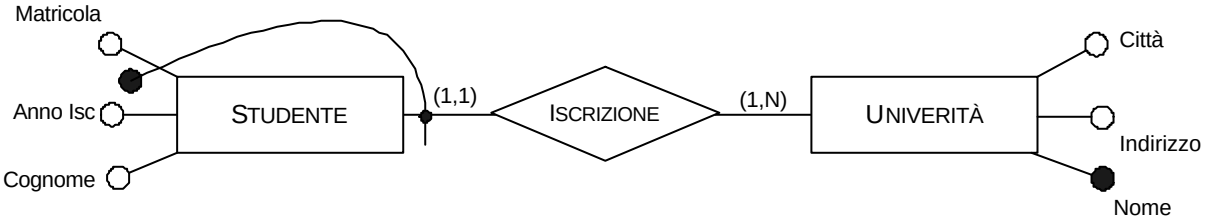
GIOCATORE (Cognome, DataNascita, Ruolo)
SQUADRA (Nome, Città, ColoriSociali)
CONTRATTO (Giocatore, DataNascitaGiocatore , NomeSquadra, Ingaggio)

Va notato che nella relazione CONTRATTO, la chiave è costituita solo dall'identificatore di GIOCATORE perché la cardinalità dell'associazione ci dicono che ogni giocatore ha un contratto con una sola squadra. A questo punto le relazioni GIOCATORE e CONTRATTO hanno la stessa chiave ed è allora possibile fonderle in un'unica relazione:
GIOCATORE (Cognome, DataNascita , Ruolo, NomeSquadra, Ingaggio)
SQUADRA (Nome, Città, ColoriSociali)

In questo schema esiste ovviamente il vincolo di integrità referenziale tra l'attributo NomeSquadra della relazione GIOCATORE e l'attributo Nome della relazione SQUADRA.

Entità con identificatore esterno

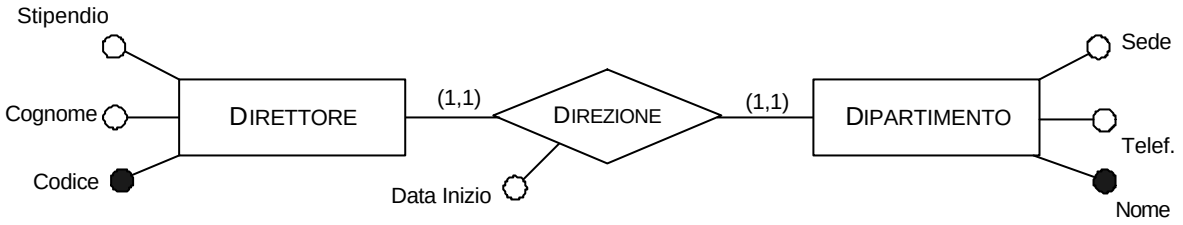
Le entità con identificatori esterni danno luogo a relazioni con chiavi che includono gli identificatori delle entità "identificanti".



Lo schema relazionale corrispondente è il seguente:
STUDENTE (Matricola, NomeUniversità , Cognome, AnnoIscrizione)
UNIVERSITÀ (Nome, Città, Indirizzo)
Nel quale esiste il vincolo di integrità referenziale tra l'attributo NomeUniversità della relazione STUDENTE e l'attributo Nome della relazione UNIVERSITÀ.
Come si può vedere, rappresentando l'identificatore esterno si rappresenta direttamente anche l'associazione tra le due entità. Ricordiamo infatti che le entità identificate esternamente partecipano all'associazione sempre con una cardinalità minima e massima pari a uno.

Associazioni uno a uno

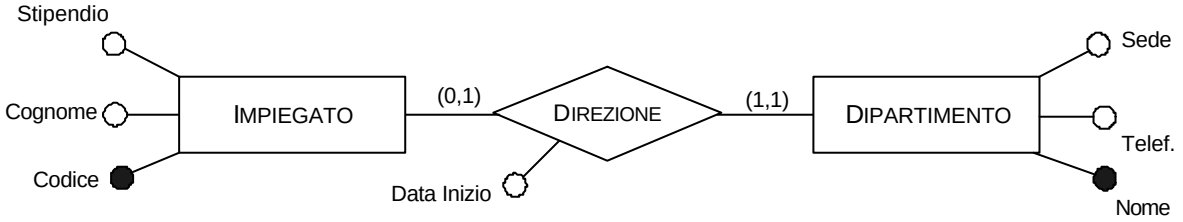
Per le associazioni uno a uno ci sono, in genere, diverse possibilità di traduzione. Cominciamo a vedere le associazioni uno a uno con partecipazioni obbligatorie per entrambe le entità.



Per questo tipo di associazioni abbiamo due possibilità simmetriche e ugualmente valide:
DIRETTORE (Codice, Cognome, Stipendio, DipartimentoDiretto, InizioDirezione)
DIPARTIMENTO (Nome, Telefono, Sede)

con il vincolo di integrità referenziale tra l'attributo DipartimentoDiretto della relazione DIRETTORE e l'attributo Nome della relazione DIPARTIMENTO, oppure:
DIRETTORE (Codice, Cognome, Stipendio)
DIPARTIMENTO (Nome, Telefono, Sede, Direttore, InizioDirezione)

per il quale esiste il vincolo di integrità referenziale tra l'attributo Direttore della relazione DIPARTIMENTO e l'attributo Codice della relazione DIRETTORE.
E' possibile quindi rappresentare l'associazione in una qualunque delle relazioni che rappresentano le due entità. Trattandosi di una relazione biunivoca tra le occorrenze delle entità, sembrerebbe possibile una ulteriore alternativa nella quale si rappresentano tutti i concetti in un'unica relazione contenente tutti gli attributi che lo schema che stiamo traducendo è il risultato di una fase di ristrutturazione nella quale sono state effettuate precise scelte anche riguardo l'accorpamento e il partizionamento di entità.



Nel caso che la prima entità abbia partecipazione opzionale abbiamo una soluzione preferibile rispetto alle altre:
DIRETTORE (Codice, Cognome, Stipendio)
DIPARTIMENTO (Nome, Telefono, Sede, Direttore, InizioDirezione)

Nel caso infine che entrambe le entità abbiano partecipazione opzionale, come ad esempio se nel caso precedente un dipartimento possa non avere direzioni, esiste una ulteriore possibilità che prevede tre relazioni separate:
DIRETTORE (Codice, Cognome, Stipendio)
DIPARTIMENTO (Nome, Telefono, Sede)
DIREZIONE (Direttore, Dipartimento, DataInizioDirezione)

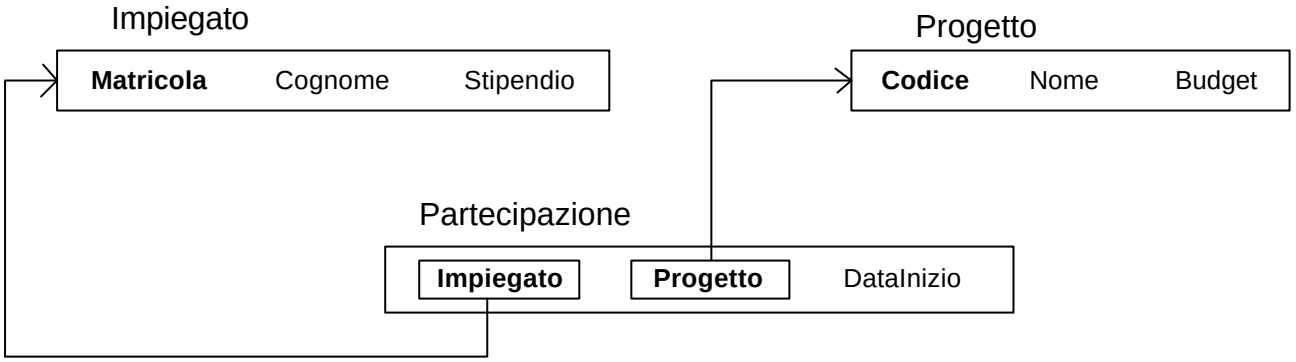
Questa relazione ha il vantaggio, rispetto a quella in cui si accorpa la relazione Direzione in una delle altre due, di non presentare mai valori nulli sugli attributi che rappresentano l'associazione. Per contro, abbiamo bisogno di una relazione in più, con un conseguente aumento della complessità della base di dati.

Documentazione di schemi logici

Il risultato della progettazione logica non è costituito solo da un semplice schema di base di dati ma anche da una documentazione a esso associata. Buona parte della documentazione dello schema concettuale in ingresso alla fase di progettazione logica può essere ereditata dallo schema logico ottenuto come risultato di questa fase. A questo documentazione ne va aggiunta però dall'altra, in grado di descrivere i vincoli di integrità referenziale introdotti dalla traduzione.

A tale riguardo, è possibile adottare un semplice formalismo grafico che permette di rappresentare sia le relazioni con i relativi attributi, sia i vincoli di integrità referenziale esistenti tra le varie relazioni.

In questi diagrammi le chiavi delle relazioni sono rappresentate in grassetto, le frecce indicano vincoli di integrità referenziale e la presenza di eventuale asterischi sui nomi di attributo indica la possibilità di avere valori nulli.



Si può osservare che, con questo formalismo, si riesce a mantenere traccia delle associazioni dello schema E-R originale. Questo può risultare utile per individuare i *cammini di join*, ovvero le operazioni di join necessarie per ricostruire l'informazione rappresentata dalle associazioni originarie.

E' interessante osservare come, sia possibile rappresentare esplicitamente anche le associazioni dello schema E-R di partenza alle quali, nello schema relazionale equivalente, non corrisponde nessuna relazione.

8. La normalizzazione

In questo capitolo studieremo alcune proprietà, dette *forme normali*, che certificano la qualità dello schema di una base di dati relazionale. Questo concetto può essere utilizzato per effettuare controlli di qualità di basi di dati relazionali e costituisce per questo un utile strumento di analisi nell'ambito dell'attività di progettazione di una base di dati. Per gli schemi che non soddisfano una forma normale, è inoltre possibile applicare un procedimento, detto di *normalizzazione*, che consente di trasformare questi schemi non normalizzati in nuovi schemi per i quali il soddisfacimento di una forma normale è garantito.
La teoria della normalizzazione costituisce un utile strumento di verifica, in grado di suggerire emendamenti, ma che non può sostituire, le metodologie di analisi e progettazione.
La teoria della normalizzazione è stata studiata nell'ambito del modello relazione.

Ridondanze e anomalie

Impiegato	*Stipendio*	*Progetto*	*Bilancio*	*Funzione*
Rossi	20000000	Marte	2000	Tecnico
Verdi	35000000	Giove	15000	Progettista
Verdi	35000000	Venere	15000	Progettista
Neri	55000000	Venere	15000	Direttore
Neri	55000000	Giove	15000	Consulente
Neri	55000000	Marte	2000	Consulente
Mori	48000000	Marte	2000	Direttore
Mori	48000000	Venere	15000	Progettista
Bianchi	48000000	Venere	15000	Progettista
Bianchi	48000000	Giove	15000	Direttore

Esempio di base di dati relazionale.

Si può notare 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.

- Il valore dello stipendio di ciascun impiegato è ripetuto in tutte le tuple relative ad esso: si ha quindi una *ridondanza*.
- 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: *anomalia di aggiornamenti.*
- Se un impiegato interrompe la partecipazione a tutti i progetti senza lasciare l'azienda non è possibile conservare traccia del suo nome e del suo stipendio, che potrebbero rimanere di interesse: *anomalie di cancellazione*.
- Se si hanno informazioni su un nuovo impiegato, no è possibile inserirle finché non viene assegnato a un progetto: *anomalia di inserimento.*

Una motivazione intuitiva della presenza di questi inconvenienti può essere la seguente: abbiamo usato un'unica relazione per rappresentare informazioni eterogenee.
Generalizzando possiamo arrivare alle seguenti conclusioni, che evidenziano i difetti presentati da relazioni che riuniscono concetti fra loro disomogenei.

- E' possibile che alcuni dati debbano essere ripetuti in diverse tuple, senza aggiungere in tal modo informazioni significative.
- Se alcune informazioni sono ripetute in modo ridondante, il relativo aggiornamento deve essere ripetuto per ciascuna occorrenza dei relativi dati.
- La cancellazione di una tupla, motivata dal fatto che non è più valido l'intero insieme di concetti da essa espressi, può comportare l'eliminazione di tutti i concetti in questione, cioè anche di quelli che conservano la loro validità.

- L'inserimento di informazioni relative a uno solo dei concetti di pertinenza per una relazione non è possibile se non esiste un intero insieme di concetti in grado di costituire una tupla completa.

Dipendenze funzionali

Per studiare i concetti introdotti informalmente , è necessario fare uso di uno specifico strumento di lavoro: la *dipendenza funzionale.* Si tratta di un particolare vincolo di integrità per il modello relazionale che descrive legami di tipo funzionale tra gli attributi di una relazione.
Possiamo cioè dire che il valore dell'attributo Impiegato determina il valore dell'attributo Stipendio o 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.
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 *X*, 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 di attributi *Y*, risulta che t_1 e t_2 hanno gli stessi valori anche sugli attributi *Z*.
Impiegato → Stipendio
Progetto → Bilancio

Una seconda 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 un qualunque altro attributo o insieme di attributi dello schema di *r*.
Possiamo infine dire che il vincolo di dipendenza funzionale *generalizza* il vincolo di chiave.

Forma normale di Boyce e Codd

Definizione di forma normale di Boyce e Codd

Riprendendo l'esempio notiamo che le due proprietà causa di anomalie corrispondono esattamente ad attributi coinvolti in dipendenze funzionali.

- La proprietà "Lo stipendio di ciascun impiegato è funzione del solo impiegato" implica il soddisfacimento della dipendenza funzionale Impiegato → Stipendio
- La proprietà "Il bilancio di ciascun progetto dipende dal solo progetto" corrisponde alla dipendenza Progetto → Bilancio

Inoltre l'attributo funzione indica, per ciascuna tupla, il ruolo svolto dall'impiegato nel progetto ed è unico per ciascuna coppia Impiegato-progetto.

- La proprietà "In ciascun progetto, ciascuno degli impiegati coinvolti può svolgere una e una sola funzione" è conseguenza del fatto che gli attributi Impiegato e Progetto formano la chiave della relazione Impiegato Progetto → Funzione

Le prime due dipendenze sono causa di anomalie, la terza no.
Possiamo quindi concludere che le ridondanze e le anomalie sono causate dalle dipendenze funzionali $X \rightarrow Y$ che permettono la presenza di più tuple fra loro uguali sugli attributi in *X*, cioè dalle dipendenze funzionali $X \rightarrow Y$ tali che *X* non contiene la chiave.
Precisiamo queste idee per mezzo del concetto di forma normale di Boyce e Codd, che prende il nome dai suoi ideatori. Una relazione *r* è in *forma normale di Boyce e Codd* se per ogni dipendenza funzionale (non banale) $X \rightarrow Y$ 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 forma normale di Boyce Codd, perché i concetti indipendenti sono separati, uno per relazione.

Decomposizione in forma normale di Boyce e Codd

Data una relazione che non soddisfa la forma normale di Boyce e Codd è possibile, in molti casi, sostituirla con due o più relazioni normalizzate attraverso un processo detto di *normalizzazione*. Se

una relazione rappresenta più concetti indipendenti, allora va decomposta in relazioni più piccole, una per ogni concetto.

Impiegato	*Stipendio*
Rossi	20000000
Verdi	35000000
Neri	55000000
Mori	48000000
Bianchi	48000000

Progetto	*Bilancio*
Marte	2000
Giove	15000
Venere	15000

Impiegato	*Progetto*	*Funzione*
Rossi	Marte	Tecnico
Verdi	Giove	Progettista
Verdi	Venere	Progettista
Neri	Venere	Direttore
Neri	Giove	Consulente
Neri	Marte	Consulente
Mori	Marte	Direttore
Mori	Venere	Progettista
Bianchi	Venere	Progettista
Bianchi	Giove	Direttore

Nell'esempio, la separazione delle dipendenze è stata facilitata dalla struttura delle dipendenze stesse, "naturalmente" separate e indipendenti l'una dall'altra.

In generale però le dipendenze possono avere una struttura complessa: può non essere necessario (o possibile) basare la decomposizione su tutte le dipendenze e può essere difficile individuare quelle su cui si deve basare la decomposizione.
Vediamo un altro esempio:

Impiegato	*Categoria*	*Stipendio*
Neri	3	30000000
Verdi	3	30000000
Rossi	4	50000000
Mori	4	50000000
Bianchi	5	72000000

E' facile verificare che tale relazione soddisfa le dipendenze Impiegato → Categoria e Categoria → Stipendio. Allo stesso tempo avremmo potuto individuare, insieme alla dipendenza funzionale Categoria → Stipendio, anche la dipendenza Impiegato → Categoria Stipendio (anziché Impiegato → Categoria).
In effetti, una trattazione completa della normalizzazione richiederebbe come primo passo uno studio approfondito delle proprietà delle dipendenze funzionali. Ribadiamo che, in questo contesto, la teoria della normalizzazione risulta comunque utile, come strumento di verifica dei prodotti sia della progettazione concettuale sia della progettazione logica.

Proprietà delle decomposizioni

Decomposizione senza perdita

Per discutere la prima proprietà, esaminiamo questa relazione

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

Tale relazione soddisfa le dipendenze funzionali
Impiegato → Sede

Progetto → Sede
che sostanzialmente, specificano il fatto che ciascun impiegato opera presso un'unica sede e che ciascun progetto è sviluppato presso un'unica sede. Operando come nei casi precedenti, separando cioè sulla base delle dipendenze, saremmo portati a decomporre la relazione in due parti:

- Una relazione sugli attributi Impiegato e Sede, in corrispondenza alla dipendenza Impiegato → Sede
- L'altra sugli attributi Progetto e Sede, in corrispondenza alla dipendenza funzionale Progetto → Sede

Esaminando le due relazioni notiamo come sia possibile ricostruire le informazioni sulla partecipazione degli impiegati ai progetti solamente utilizzando l'attributo Sede, che è l'unico attributo comune alle due relazioni. Purtroppo però, in questo caso, non riusciamo a ricostruire tutte e sole le informazioni nella relazione originaria.
Possiamo generalizzare l'osservazione notando come la ricostruzione della relazione originaria a partire dalle sue proiezioni debba intuitivamente essere effettuata per mezzo di una operazione di join naturale delle due proiezioni.
Diciamo che *r* si *decompone senza perdita* su X_1 e X_2 se il join delle due proiezioni è uguale a *r* stessa. E' irrinunciabile che una decomposizione effettuata a fini di normalizzazione sia senza perdita.
E' possibile individuare una condizione che garantisce la decomposizione senza perdita di una relazione:
Sia r una relazione su X e siano X_1 e X_2 sottoinsiemi di X tali che $X_1 \cup X_2 = X$. Inoltre sia $X0 = X_1 \cap X_2$. Se *r* soddisfa la dipendenza funzionale $X_0 \rightarrow X_1$,oppure la dipendenza funzionale $X_0 \rightarrow X_2$, allora *r* si decompone senza perdita su X_1 e X_2.

In altre parole, possiamo dire che *r* si decompone senza perdita su due relazioni se l'insieme degli attributi comuni alle due relazioni è chiave per almeno una delle relazioni composte.
E' opportuno notare che la condizione enunciata sia sufficiente ma non strettamente necessaria a garantire la decomposizione senza perdita: esistono infatti relazioni che non soddisfano nessuna delle due dipendenze, ma al tempo stesso si decompongono senza perdita.

Conservazione delle dipendenze

Consideriamo l'esempio precedente. Volendo rimuovere le anomalie, potremmo pensare di sfruttare solo la dipendenza Impiegato → Sede per ottenere una decomposizione senza perdita, ottenendo due relazioni, una sugli attributi Impiegato e Sede e l'altra sugli attributi impiegato e Progetto.. Il join delle due relazioni porta effettivamente la relazione originaria, per cui possiamo dire che la relazione si decompone senza perdita su Impiegato, Sede e Impiegato, Progetto. La decomposizione però presenta 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.
Generalizzando possiamo quindi concludere che, in ogni decomposizione, ciascuna delle dipendenze funzionali dello 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.

Qualità delle decomposizioni

Per riassumere le considerazioni svolte possiamo affermare che le decomposizioni dovrebbero sempre soddisfare le proprietà di *decomposizione senza perdita* e *conservazione delle dipendenze*.

- La decomposizione senza perdita garantisce che le informazioni nella relazione originaria siano ricostruibili con precisione.
- La conservazione delle dipendenze garantisce che le relazioni decomposte hanno la stessa capacità della relazione originaria di rappresentare i vincoli di integrità e quindi di rilevare aggiornamenti illeciti: a ogni aggiornamento lecito sulle relazioni decomposte.

Di conseguenza, nel seguito considereremo accettabili, cioè di qualità sufficiente, solo le decomposizioni che soddisfano queste due proprietà. Dato uno schema che violi una forma normale, l'attività di normalizzazione è quindi volta a ottenere una decomposizione che sia senza perdita, che conservi le dipendenze e che contenga relazioni in forma normale.

Terza forma normale

Definizione di terza forma normale

Nella maggior parte dei casi si può raggiungere l'obiettivo di una buona decomposizione in forma normale di Boyce e Codd. Talvolta però questo non è possibile.

Dirigente	***Progetto***	***Sede***
Rossi	Marte	Roma
Verdi	Giove	Milano
Verdi	Venere	Milano
Neri	Saturno	Milano
Neri	Venere	Milano

Su tale relazione possiamo supporre che siano definite le seguenti dipendenze:

- Dirigente → Sede : ogni dirigente opera presso una sede;
- Progetto Sede → Dirigente : ogni progetto ha più dirigenti che ne sono responsabili, ma in sedi diverse, e ogni dirigente può essere responsabile di più progetti; però, per ogni sede, un progetto ha un solo responsabile.

La relazione non è in forma normale di boyce e Codd perché il primo membro della dipendenza Dirigente → Sede non è superchiave. Al tempo stesso possiamo notare come non sia possibile alcuna buona decomposizione di questa relazione.
Possiamo dunque affermare che, talvolta, la "forma normale di boyce e Codd non è raggiungibile". In questi casi, si rincorre a una condizione meno restrittiva, che sostanzialmente consente situazioni come quella descritta, ma non ammette ulteriori fonti di ridondanza e anomalia.
Tale condizione definisce una nuova forma normale: diremo che una relazione è in *terza forma normale* se, per ogni dipendenza funzionale X → Y definita su di essa, almeno una delle seguenti condizioni è verificata:

- X contiene una chiave K di r;
- ogni attributo in Y è contenuto in almeno una chiave di r.

In sostanza abbiamo che la terza forma normale è meno forte della forma normale di Boyce e Codd e quindi non offre le medesime garanzie di qualità per una relazione; ha però rispetto a essa il vantaggio di essere sempre ottenibile. E' possibile infatti dimostrare che una qualunque relazione che non soddisfa la terza forma normale è certamente decomponibile senza perdita e con conservazione delle dipendenze in relazioni in terza forma normale.
In nome di questa forma normale suggerisce l'esistenza di altre forme normali che citiamo.
La ***prima forma normale*** stabilisce semplicemente una condizione che sta alla base del modello relazionale stesso: gli attributi delle relazioni sono definiti su valori atomici e non su valori complessi quali insiemi o relazioni.
La ***seconda forma normale*** è una variante debole della terza che consente dipendenze funzionali.
Esistono anche altre forme normali che fanno riferimento peraltro a vincoli di integrità diversi dalle dipendenze funzionali. Tutte queste forme normali vengono poco usate nelle applicazioni odierne in quanto è stato rilevato che la terza forma normale e la forma normale di Boyce e Codd già forniscono il giusto compromesso tra semplicità e qualità dei risultati.

Decomposizione in terza forma normale

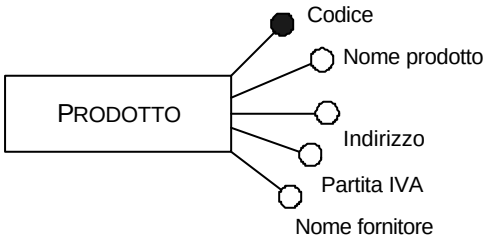
La decomposizione in terza forma normale può procedere, intuitivamente, come suggerito nel caso della forma normale di Boyce e Codd: una relazione che non soddisfa la terza forma normale si decompone in relazioni ottenute per proiezione sugli attributi corrispondenti alle dipendenze funzionali, con l'unica accortezza di mantenere sempre una relazione che contiene una chiave della relazione originaria.
Il risultato di questa decomposizione in terza forma normale produce nella maggior parte dei casi schemi in forma normale di Boyce e Codd. In particolare, si può dimostrare che se una relazione ha solo una chiave allora le due forme normali coincidono, cioè una relazione in terza forma normale è anche in forma normale Boyce e Codd.

Progettazione di basi di dati e normalizzazione

La teoria della normalizzazione, anche studiata in modo semplificato, può essere utilizzata come base per operazioni di verifica di qualità di schemi, sia nella fase di progettazione concettuale sia in quella di progettazione logica.
Le attività di progettazione sono sempre soggette a errori e incompletezze, quindi una revisione delle relazioni ottenute in sede di progettazione logica può portare alla individuazione di imprecisioni nella formulazione dello schema concettuale: la verifica è spesso relativamente semplice, poiché l'individuazione delle dipendenze funzionali e delle chiavi deve essere svolta nell'ambito di una singola relazione, che corrisponde a una entità o una associazione già analizzata nella progettazione concettuale.

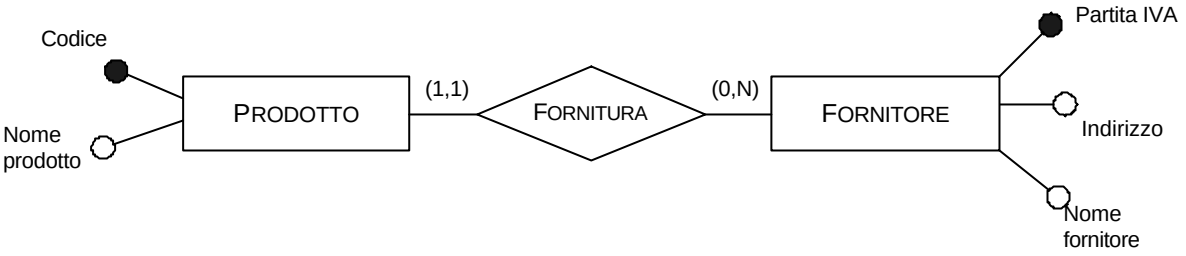
Verifiche di normalizzazione su entità

Le idee alla base della normalizzazione possono essere utilizzate anche durante la fase di progettazione concettuale, con riferimento quindi ai costrutti del modello Entità-Relazione. In effetti, è possibile considerare ciascuna entità e ciascuna associazione come una relazione. In particolare, la relazione che corrisponde a una entità ha attributi che corrispondono esattamente agli attributi dell'entità. La verifica di normalizzazione può quindi procedere come visto finora. In pratica, 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).
Esempio:



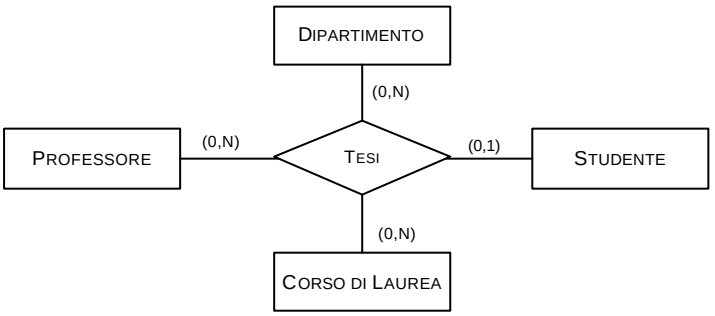
Nell'individuare le dipendenze relative a tale entità possiamo notare che possono esistere fornitori diversi con lo stesso nome o lo stesso indirizzo, mentre tutte le proprietà di ogni fornitore sono identificate dalla partita IVA: sussiste cioè la dipendenza PartitaIVA → NomeFornitore Indirizzo. Inoltre tutti gli attributi dipendono funzionalmente dall'attributo Codice, che costituisce quindi l'identificatore dell'entità.
Poiché l'unico identificatore dell'entità è costituito dal solo attributo Codice, possiamo concludere che l'entità viola la terza forma normale, in quanto la dipendenza PartitaIVA → NomeFornitore Indirizzo ha un primo membro che no contiene l'identificazione e un secondo membro composto da attributi che non fanno parte della chiave. In questi case la verifica di la normalizzazione ci permette di segnalare il fatto che lo schema concettuale non è accurato e ci suggerisce di decomporre l'entità stessa.

La decomposizione può avvenire, come abbiamo visto in precedenza, con diretto riferimento alle dipendenze, oppure, più semplicemente, ragionando qualitativamente sui concetti rappresentati dall'entità insieme a quelli che derivano dalle dipendenze funzionali. Nel caso in esame , rilevando la dipendenza funzionale, abbiamo capito che il concetto di fornitore è in effetti indipendente da quello di prodotto e ha proprietà associate. Possiamo quindi dire che è opportuno modellare il concetto di fornitore per mezzo di una entità, con identificatore costituito dall'attributo PartaIVA e ulteriori attributi NomeFornitore e Indirizzo.



Verifiche di normalizzazione su associazioni

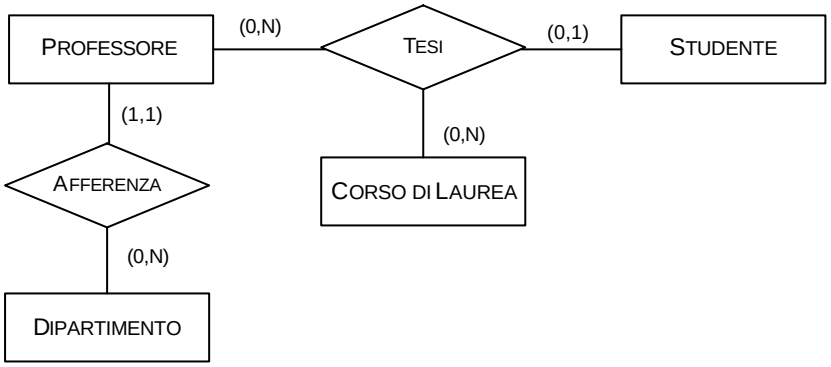
Per quanto riguarda le associazioni, il ragionamento è per certi aspetti più semplice, perché l'insieme delle occorrenze di ciascuna associazione è una relazione, quindi è possibile applicare direttamente i concetti connessi con le forme normali, ma per altri più complesso, perché i domini su cui tale relazione è definito sono gli insiemi delle occorrenze delle entità coinvolte.
Consideriamo questo esempio:



Esaminiamo 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 ;
- ogni professore afferisce a un solo dipartimento e gli studenti sotto la supervisione svolgono la tesi presso tale dipartimento.

Supponendo che non sia rilevante ai fini della tesi di laurea l'afferenza del professore al corso di laurea cui lo studente è iscritto, possiamo 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.
La terza dipendenza causa quindi una violazione della terza forma normale. In effetti l'afferenza di un professore a un dipartimento è un concetto indipendente dall'esistenza di studenti che svolgono la tesi con il professore stesso. Trasformiamo quindi il tutto in:


PROFESSORE
(0,N)
TESI
(0,1)
STUDENTE
(1,1)
(0,N)
AFFERENZA
CORSO DI LAUREA
(0,N)
DIPARTIMENTO

9. Tecnologia di un database server

In questo capitolo ci concentriamo sugli aspetti tecnologici che caratterizzano il funzionamento di un database server.
In un server per la gestione dei dati troviamo i seguenti componenti:

- *l'ottimizzatore*, con il compito di decidere le strategie di accesso ai dati per rispondere alle interrogazioni;
- il gestore dei metodi di accesso ai dati, detto *relational storage system* (RSS) di un DMBS relazionale;
- il *buffer manager*, per la gestione del trasferimento effettivo delle pagine nella base di dati dai dispositivi di memoria di massa alla memoria centrale;
- il *controllore della affidabilità,* per preservare correttamente il contenuto della base di dati in presenza di malfunzionamenti e guasti;
- il *controllore della concorrenza*, per regolare gli accessi concorrenti alla base di dati e garantire che le applicazioni non interferiscano negativamente fra loro causando perdite di consistenza.

Preliminarmente introdurremo la nozione di *transazione,* che è fondamentale per comprendere i requisiti che la tecnologia dei DBMS deve realizzare.

Definizione di transazione

Una *transazione* identifica una unità elementare di lavoro svolta da una applicazione. Un sistema che mette a disposizione un meccanismo per la definizione e l'esecuzione di transizioni viene detto *sistema transazionale*.
Una transazione può essere definita sintatticamente: ogni transazione, quale che sia il linguaggio di programmazione in cui essa è scritta, è incapsulata all'interno di due comandi: **`begin transaction`** (**`bot`**) e **`end transaction`** (**`eot`**). All'interno del codice delle transazioni, possono comparire due istruzioni particolari: **`comit work`** e **`abort work`**. L'effetto di questi due comandi è decisivo per l'esito della transazione, la quale va a buon fine solo a seguito di una commit, mentre non ha alcun effetto tangibile sulla base di dati quando viene eseguito un abort.
Viene detta ben formata una transazione iniziata da **`begin transaction`**, conclusa da **`end transaction`**, nel cui corso viene eseguito uno solo dei due comandi **`commit work`** o **`abort work`** e in cui non avvengono operazioni di accesso e/o modifica alla base di dati successive all'esecuzione del comando di commit o abort.

Proprietà acide delle transazioni

Tutto il codice che viene eseguito all'interno di una coppia bot-eot gode di proprietà particolari, le cosiddette *proprietà acide* delle transizioni: *atomicità*, *consistenza*, *isolamento* e *persistenza*, ove ACID denota le iniziali di : "Atomicity, Consistency, Isolation, Durability".

Atomicità

L'atomicità rappresenta il fatto che una transazione è una unità indivisibile di esecuzione; o vengono resi visibili tutti gli effetti di una transazione oppure la transazione non deve avere alcun effetto. L'atomicità ha significative conseguenze sul piano operativo. Il sistema deve essere in grado di ricostruire la situazione esistente all'inizio di una transazione, disfacendo il lavoro svolto dalle istruzioni eseguite fino a quel momento (**`undo`**). Viceversa, dopo una commit, il sistema deve assicurare che la transazione lasci la base di dati nel suo stato finale; come vedremo, ciò può comportare di dover rifare il lavoro svolto (**`redo`**).
Quando viene eseguito il comando **`rollback work`**, la situazione è simile ad un "suicidio" autonomamente deciso nell'ambito della transazione.

Consistenza

La consistenza richiede che l'esecuzione della transazione non violi i vincoli di integrità definiti sulla base di dati. La verifica di vincoli di integrità di tipo immediato può essere fatta nel corso dei dati che causa la violazione del vincolo. Invece la verifica di vincoli di integrità di tipo di tipo

differito deve essere effettuata alla conclusione della transazione, dopo che l'utente ha richiesto il commit.

Isolamento

L'isolamento richiede che l'esecuzione di una transazione sia indipendente dalla contemporanea esecuzione di altre transazioni. In particolare si richiede che il risultato dell'esecuzione concorrente i un insieme di transazioni sia analogo al risultato che le stesse transazioni otterrebbero qualora ciascuna di esse fosse eseguita da sola.

Persistenza

La persistenza invece richiede che l'effetto di una transazione che ha eseguito il commit correttamente non venga più perso.

Transazione e moduli di sistema

Atomicità e persistenza sono garantite dal controllo di affidabilità. L'isolamento è garantito dal controllo della concorrenza. La consistenza è infine gestita dai compilatori del DDL, che introducono opportuni controlli di consistenza nei dati e opportune procedure per la loro verifica, eseguite poi dalle transazioni.

Controllo di concorrenza

Una unità di misura che viene tipicamente utilizzata per caratterizzare il carico applicativo di un DBMS è il numero di transazioni al secondo (tps) che devono essere gestite dai DBMS per soddisfare le applicazioni, e raggiungono anche decine o centinaia di tps.

Per questo motivo è indispensabile che le transazioni vengono eseguite concorrentemente; è impensabile infatti una loro esecuzione seriale. Solo la concorrenza delle transazioni consente un uso efficiente del DBMS, massimizzando il numero di transazioni servite per secondo e minimizzando i tempi di risposta.

Architettura

Il controllo di concorrenza fa riferimento alla frontiera di più basso livello nell'architettura del DBMS, relativa alle *operazioni di ingresso/uscita*, che attuano il trasferimento di blocchi di memoria di massa alla memoria centrale. Consideriamo operazioni di lettura (read) e scrittura (write). Ogni operazione di lettura comporta il trasferimento di un blocco da memoria di massa alla memoria centrale e ogni operazione di scrittura comporta il trasferimento opposto. I blocchi di ingresso/uscita vengono chiamati pagine una volta caricati in memoria. Le operazioni di lettura e scrittura vengono gestite da un modulo di sistema, detto genericamente *scheduler*.

Le azioni r(x) e w(x) denotano rispettivamente la lettura e la scrittura della pagina in cui il dato x è memorizzato nel DBMS.

Anomalie delle transazioni concorrenti

L'esecuzione concorrente di varie transazioni può causare alcuni problemi di correttezza, o anomalie; la presenza di queste anomalie motiva la necessità di controllare la concorrenza.

Perdita di aggiornamento

Supponiamo di avere due transazioni identiche che operano sullo stesso oggetto della base di dati:

t_1: $r(x), x = x + 1, w(x)$

t_2: $r(x), x = x + 1, w(x)$

In caso di una possibile esecuzione concorrente delle due transazioni, è possibile che:

Transazione t1	*Transazione t2*
bot	
$r_1(x)$	
$x = x + 1$	
	bot
	$r_2(x)$
	$x = x + 1$

	$w_2(x)$
	commit
$w_1(x)$	
commit	

In tal caso il valore finale di *x* è 3, perché entrambe le transazioni leggono 2 come valore iniziale. Questa anomalia prende il nome di *perdita di aggiornamento* (*lost update*).

Letture sporche

Consideriamo le stesse transazioni, e analizziamo la seguente esecuzione:

Transazione t1	*Transazione t2*
bot	
$r_1(x)$	
$x = x + 1$	
$w_1(x)$	**bot**
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	commit
abort	

Il valore finale di *x* al termine dell'esecuzione è 4 invece di 3. L'aspetto critico di questa esecuzione è la lettura della transazione t2, che vede uno stato intermedio generato dalla t1. Questa anomalia prende il nome di *lettura sporca* (*dirty read*).

Letture inconsistenti

Supponiamo invece che la transazione t1 svolga solamente operazioni di lettura, ma che ripeta la lettura del dato *x* in istanti successivi:

Transazione t1	*Transazione t2*
bot	
$r_1(x)$	
	bot
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	commit
$r_1(x)$	
commit	

In tal caso *x* assume il valore 2 dopo la prima lettura e 3 dopo la seconda.

Aggiornamento fantasma

Si consideri una base di dati con 3 oggetti *x*, *y*, *z* che soddisfano un vincoli di integrità, tali cioè che x+y+z = 1000:

Transazione t1	*Transazione t2*
bot	
$r_1(x)$	
	bot
	$r_2(y)$
$r_1(y)$	
	$y = y - 100$
	$r_2(z)$
	$z = z + 100$
	$w_2(y)$
	$w_2(z)$
	commit
$r_1(z)$	
$s = x + y + z$	
commit	

La transazione t2 non altera la somma dei valori e quindi rispetta il vincolo di integrità, però la variabile s della transazione t1, che dovrebbe contenere 1000 contiene in realtà 1100. Questa anomalia prende il nome di *aggiornamento fantasma* (*ghost update*).

Teoria del controllo di concorrenza

Definiamo una transazione come una sequenza di azioni di lettura o scrittura, che si riconoscono come eseguite da una stessa transazione in quanto caratterizzate dallo stesso indice. Assumiamo che la transazione sia iniziata dal comando transazionale begin transaction e terminata da end transaction, che vengono omessi; il sistema di controllo di concorrenza si pone l'obiettivo di accettare o rifiutare esecuzioni concorrenti durante l'evoluzione delle transazioni, quindi non può conoscere l'esito finale. Ad esempio una transazione t1 è rappresentata dalla sequenza:
$t_1 : r_1(x)\ r_1(y)\ w_1(x)\ w_1(y)\ldots$
Poniamo anche come ragionevoli assunzioni che ogni transazione non legga o scriva lo stesso oggetto più volte.
Dato che le transazioni avvengono in modo concorrente, le operazioni di ingresso/uscita vengono richieste da varie transazioni in istanti successivi. Uno *schedale* rappresenta la sequenza di operazioni di ingresso/uscita presentate da transazioni concorrenti. Uno schedale S1 è quindi una sequenza del tipo:
$S_1 : r_1(x)\ r_2(z)\ w_1(x)\ w_2(z)\ldots$
Il controllo di concorrenza ha la funzione di accettare alcuni schedule e rifiutarne altri, in modo ad esempio da evitare che si verifichino le anomalie illustrate. Tale compito è effettuato da uno *scheduler*, il quale ha il compito di tenere traccia di tutte le operazioni compiute sulla base di dati dalle transazioni, e accettare o rifiutare le operazioni che vengono progressivamente richieste dalle transazioni.
Inizialmente ci occuperemo di caratterizzare la correttezza degli schedale assumendo che le transazioni che compaiono abbiano un esito noto; in questo modo è possibile ignorare le transazioni che producono un abort, togliendo dallo schedale tutte le loro azioni, e concentrarsi solo sulle transazioni che producono un commit. Lo schedule si dice una commit-proiezione della esecuzione delle operazioni di i/o. Comunque uno schedule si deve comportare indipendentemente dall'esito finale con determinati criteri.
Vogliamo ora determinare delle condizioni sugli schedule che garantiscono la correttezza dell'esecuzione delle corrispondenti transazioni. Definiamo seriale uno schedale in cui le azioni di tutte le transazioni compaiono in sequenza, senza essere inframezzate da istruzioni di altre transazioni.
L'esecuzione della commit-proiezione di uno schedule è corretta quando produce lo stesso risultato prodotto da un qualunque schedule seriale delle stesse transazioni.
Per chiarire cosa intendiamo con "produrre lo stesso risultato" introduciamo la *view-equivalenza*, la *conflict-equivalenza*, il *locking a due fasi* e il controllo di concorrenza basato su *timestamp*.

View-equivalenza

Si definisce dapprima una relazione che lega coppie di operazioni di lettura-scrittura: una 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 $w_k(x)$ tra le due operazioni. Una operazione di scrittura è detta *scrittura finale* se è l'ultima scrittura di quell'oggetto.
Due schedule vengono detti view-equivalenti se possiedono la stessa relazione "legge-da" e le stesse scritture finali. Uno schedule è detto view-serializzabile se è view-equivalente a uno schedule seriale. Gli schedule view-serializzabili sono denominati VSR.
Il concetto di view-equivalenza è utilizzato per confrontare tra diversi schedule. Al contrario, qualora non sia dato uno schedule di confronto, è necessario confrontare lo schedule con tutti gli schedule seriali ottenuti permutando in tutti i modi possibili le transazioni presenti. 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.

Conflict-equivalenza

Una nozione di equivalenza più facilmente utilizzabile richiede la definizione di conflitto. Si dice che l'azione a_i è in *conflitto* con a_j se entrambe operano sullo stesso oggetto e almeno una di esse è una write. Possono esistere conflitti lettura-scrittura o conflitti scrittura-scrittura.
Si dice che lo schedule Si conflict-equivalente allo schedule Sj se i due schedule presentano le stesse operazioni e ogni coppia di operazioni in conflitto è nello stesso ordine in entrambi gli schedule. Uno schedule risulta quindi conflict-seriabilizzabile se esiste uno schedule seriale ad esso conflict-equivalente.
L'insieme di tutti e soli gli schedule conflict-serializzabili viene denominato CSR.
E' possibile determinare se uno schedule è conflict-serializzabile tramite il *grafo dei conflitti.* Il grafo è costruito facendo corrispondere un nodo ad ogni transazione. Si traccia quindi un arco orientato tra due transazioni se esiste almeno un conflitto tra una azione di uno e dell'altro. Si può dimostrare che lo schedule è serializzabile se e solo se il grafo è aciclico. Nonostante la complessità lineare, la conflict-serializzabilità risulta ancora eccessivamente onerosa in pratica.

Locking a due fasi

I meccanismi di controllo di concorrenza utilizzati in pratica superano le limitazioni discusse in precedenza. Tra di essi, il principale è il locking, usato da quasi tutti i DBMS commerciali. Il locking si basa su un principio molto semplice e cioè che tutte le operazioni di lettura e scrittura devono essere protette tramite la esecuzione di tre diverse primitive: r_lock, w_lock, e unlock. Lo scheduler (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 una struttura dato.
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 sullo stesso dato.

Quando una transazione segue queste regole si dice ben formata rispetto al locking;. 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 ad un lock esclusivo, "incrementando il livello di lock".
Il gestore della concorrenza riceve le richieste di lock provenienti dalle transazioni, e concede i lock in base ai lock precedentemente concessi alle altre transazioni. 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 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.
In pratica, solo quando un oggetto è bloccato in lettura è possibile dare risposta positiva ad un'altra richiesta di lock in lettura. Nel caso di unlock di una risorsa bloccata in condiviso, la risorsa torna libero quando non ci sono altre transazioni in lettura che operano su di essa; altrimenti essa rimane bloccata in lettura.
I meccanismi di locking possono essere usati senza ulteriori restrizioni per garantire che le transazioni che lavorano sulla base di dati accedano ai dati in mutua esclusione. Per avere però la garanzia che le tradizioni che seguano uno schedule serializzabile è necessario porre la seguente restrizione, relativa all'ordinamento delle richieste di lock, che prende il nome di principio del *lock a due fasi* (*Two Phase Locking*, 2PL).
Locking a due fasi: Una transazione, dopo aver rilasciato un lock, non può acquisirne altri.
Come conseguenza di questo 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 e una

seconda in cui i lock acquisiti vengono rilasciati. Si tenga conto che il passaggio da un r_lock a un w_lock costituisce un incremento del livello di lock, che può quindi comparire nella fase di accesso.
Un sistema in cui le transazioni sono ben formate rispetto al locking, con un lock manager che rispetta la politica dei conflitti 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.
Locking a due fasi stretto (strict 2PL): i lock di una transazione possono essere rilasciati solo dopo aver correttamente effettuato le operazioni di commit/abort.

Controllo di concorrenza basato sui timestamp

Questo metodo utilizza i timestamp, cioè identificatori associati ad ogni evento temporale che definiscono un ordinamento totale sugli eventi. Nei sistemi centralizzati, il timestamp viene generato leggendo i valore dell'orologio di sistema al momento in cui è avvenuto l'evento. Il controllo di concorrenza mediante timestamp (*metodo* TS) avviene nel seguente modo:

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

Questo metodo di controllo di concorrenza, forse il più semplice di tutti dal punto di vista della realizzazione, impone che le transazioni risultino serializzate in base all'ordine in cui esse acquisiscono il loro timestamp. Ad 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 *ts* più grande che ha letto *x*. Lo scheduler non fa altro che permettere o no l'operazione, secondo la politica:

- read(x,ts): se ts < WTM(x) la transazione viene uccisa, altrimenti la richiesta viene accettata; in tal caso RTM(x) viene aggiornato e posto pari al massimo tra RTM(x) e ts.
- Write(x,ts): se ts < WTM(x) o ts z RTM(x) la transazione viene uccisa, altrimenti la richiesta viene accettata; in tal caso WTM(x) viene aggiornato e posto pari a ts.

In pratica ogni transazione non può leggere o scriver eun 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.
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 memorizzati 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.

Meccanismi per la gestione dei lock

Un lock manager 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 r_lock, w_lock e unlock, in genere caratterizzata dai seguenti parametri:

r_lock(T x, errcode, timeout)
w_lock(T, x, errcode, timeout)
unlock(T, x)

T rappresenta l'identificativo della transazione, x l'elemento per il quale si richiede o si rilascia il lock; errcode rappresenta un valore restituito dal lock manager, e vale 0 qualora la richiesta sia stata soddisfatta, mentre assume un valore diverso da zero qualora la richiesta no sia stata soddisfatta; timeout rappresenta l'intervallo massimo di tempo che la procedura chiamante è disposta ad aspettare per ottenere il lock sulla risorsa.

Quando un processo richiede una risorsa e la richiesta può essere soddisfatta, il lock manager tiene traccia del cambiamento dello stato della risorsa nelle sue tabelle interne e restituisce immediatamente il controllo al processo.
Quando invece la richiesta non può essere immediatamente soddisfatta, il sistema inserisce in una coda associata alla risorsa il processo richiedente; 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 ad esso la risorsa.
Quando infine 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.
Alle tabelle di lock si accede molto di frequente; per questo, il lock manager mantiene queste informazioni in maniera centrale, in modo da minimizzare i tempi di accesso. Le tabelle hanno la seguente struttura: a ciascun oggetto si associano due bit di stato e un contatore, che rappresenta il numero di processi in attesa di quell'oggetto.

Blocco critico

Il blocco critico costituisce un problema rilevante, tipico dei sistemi concorrenti in cui si introducono condizioni di attesa. Il problema può essere dato da una transazione che resta in attesa che si liberi un oggetto y bloccato da una seconda transazione che a sua volta è in attesa che si liberi l'oggetto x bloccato dalla prima transazione.
La probabilità che due transazioni che operano un solo accesso vadano in conflitto è 1/n; la probabilità che si verifichi un blocco critico di lunghezza 2 è pari alla probabilità di un secondo conflitto, e quindi vale $1/n^2$.
Limitatamente al caso di blocchi critici costituiti da coppie di transazioni, la probabilità di conflitto cresce linearmente col numero globale k di transazioni presenti nel sistema e quadraticamente col numero medio m di risorse cui ciascuna transazione fa accesso.
Tre sono le tecniche che vengono comunemente usate per risolvere il problema del blocco critico:

1. timeout;
2. prevenzione (deadlock prevention)
3. rilevamento (deadlock detection).

Uso del timeout
La tecnica del timeout è molto semplice. Le transazioni rimangono in attesa di una risorsa per un tempo prefissato. Se passa questo tempo e la risorsa non è stata ancora concessa, allora alla richiesta di lock viene data risposta negativa. Per quanto riguarda la scelta del valore di timeout, bisogna saper valutare pro e contro tra due diversi aspetti: da una parte un valore elevato del timeout tende a risolvere tardi i blocchi critici, d'altra parte un timeout troppo basso corre il rischio di rilevare come blocchi anche situazioni che non lo sono.
Prevenzione dei blocchi critici
Vi sono diverse tecniche utilizzate per prevenire l'insorgenza di un blocco critico. Una tecnica prevede di richiedere il lock di tutte le risorse necessarie alla transazione in una sola volta. Esso comporta però il problema che le transazioni spesso non conoscono a priori le risorse cui vogliono accedere.
Un'altra tecnica di prevenzione dei deadlock si basa sul fatto che le transazioni acquisiscano un timestamp, e consiste nel consentire l'attesa di una transazione t_i, su una risorsa acquisita da t_j solamente se vale una determinata relazione di precedenza fra i timestamp. In questo modo circa il 50% delle richieste che generano un conflitto possono attendere in coda, mentre nel restante 50% dei casi una transazione deve essere uccisa.
Le politiche di scelta della transazione da uccidere possono essere interrompenti e no interrompenti. Una politica è interrompente se può risolvere il conflitto uccidendo la transazione che possiede la risorsa. In caso contrario, la politica è non interrompente, e una transazione può essere uccisa solo all'atto di fare una nuova richiesta.

Una politica può essere quella di uccidere le transazioni che hanno fatto meno lavoro. La situazione che si presenta è quella di un sistema senza blocchi critici, ma in cui vi sono delle transazioni in *blocco individuale (starvation)*. Per risolvere questo problema è necessario garantire che ogni transazione non possa essere uccisa un numero illimitato di volte. Una soluzione che si adotta è quella di mantenere lo stesso timestamp quando una transazione viene fatta abortire e ripartire.

Rilevamento dei blocchi critici

Questa tecnica prevede di non porre vincoli al comportamento del sistema, ma di controllare il contenuto delle tabelle di lock per rilevare eventuali situazioni di blocco. Il rilevamento di un blocco critico richiede di analizzare le relazioni di attesa tra le varie transazioni e di determinare se esiste un ciclo. La ricerca di cicli in un grafo, specie se effettuata periodicamente, risulta abbastanza efficiente; per questo motivo, alcuni DBMS commerciali utilizzando questa tecnica.

Gestione del buffer

La gestione ottimale dei buffer in memoria centrale è un aspetto essenziale del funzionamento delle basi di dati. Il buffer è una vasta zona della memoria centrale preallocata al DBMS e condivisa fra le varie transazioni; gli ultimi anni hanno visto un abbattimento dei costi delle memorie e quindi la allocazione di buffer di memoria sempre più vasti ai DBMS, fino ad arrivare alla condizione estrema in cui l'intero DBMS può essere copiato e gestito in memoria centrale.

Architettura del buffer manager

Il buffer manager si occupa del caricamento e scaricamento delle pagine dalla memoria centrale alla memoria di massa. Esso fornisce primitive per accedere alle pagine presenti nel buffer, denominate *fix*, *use*, *unfix*, *flush* e *force*; realizza poi talvolta le operazioni di ingresso/uscita in risposta a queste primitive, purché l'accesso condiviso ai dati sia consentito dallo scheduler.

Il buffer è organizzato in pagine, che hanno la stessa dimensione del blocco di ingresso-uscita utilizzato dal sistema operativo per leggere e scrivere blocchi dai dispositivi di memoria di massa. Quando una pagina della memoria di massa è presente nel buffer, il DBMS può svolgere le sue operazioni di lettura e scrittura direttamente su di essa; dato che i tempi di accesso alla memoria di massa sono dell'ordine dei millesimi di secolo e quelli di accesso alla memoria centrale sono dell'ordine dei miliardesimi di secondo è chiaro che accedere alle pagine del buffer comporta un vantaggio prestazionale significativo.

Il gestore del buffer gestisce un direttorio, che descrive il contenuto corrente del buffer indicando per ciascuna pagina caricata qual è il file fisico e il numero di blocco ad essa corrispondente.

Primitive per la gestione del buffer

Le operazioni messe a disposizione dal buffer manager per gestire il caricamento e lo scaricamento delle pagine sono le seguenti:

- la primitiva *fix* viene usata per richiedere l'accesso ad una pagina e caricarla nel buffer; L'esecuzione comporta una operazione di lettura da memoria di massa solo quando la pagina prescelta non è già residente nel buffer.
- La primitiva *use* viene usata dalla transazione per accedere alla pagina caricata precedentemente in memoria, confermando la sua allocazione nel buffer e il suo stato di pagina valida.
- La primitiva *unfix* indica al buffer manager che la transazione ha terminato di usare la pagina, la quale non è più valida.
- La primitiva *force* trasferisce in modo sincrono una pagina del buffer manager in memoria di massa. La transazione richiedente rimane sospesa fino al termine della esecuzione della primitiva, che comporta una operazione fisica di scrittura sulla memoria di massa.

In pratica, le primitive *fix* e *use* consentono di caricare nel buffer e leggere dati, la primitiva *force* è usata dalle transazioni per scrivere dati nella memoria di massa.

Inoltre la primitiva *flush*, di uso interno, è usata dal buffer manager stesso per trasferire in memoria di massa, in modo asincrono e indipendente dalle transazioni attive, le pagine non più

valide e rimaste inattive da più tempo; La primitiva *flush* rende disponibili pagine del buffer, che divengono *libere* e possono essere immediatamente utilizzate da successive operazioni di *fix*.
La primitiva di *fix* opera come segue:

1. Dapprima cerca la pagina vuota tra quelle già presenti in memoria, dopo l'*unifix* di altre transazioni. Se la ricerca ha esito positivo, l'operazione si conclude, e l'indirizzo della pagina viene restituito alla transazione richiedente.
2. Altrimenti, viene scelta una pagina nel buffer, cercando inizialmente di scegliere una pagina libera, e successivamente una pagina da scaricare in memoria di massa, detta *vittima*. Se viene scelta una pagina non libera, essa deve comunque essere riscritta in memoria di massa, invocando l'operazione di *flush*.
3. Infine, vengono operate le opportune conversioni in indirizzi in modo da identificare la pagina da caricare nel buffer, e avviene l'operazione di lettura.

Politiche di gestione del buffer

Descriviamo ora due coppie di politiche alternative nella gestione del buffer.

- la politica *steal* consente al buffer manager di selezionare come vittima una pagina attiva allocata ad un'altra transazione, mentre una politica *no-steal* esclude questa possibilità.
- La politica *force* richiede che tutte le pagine attive di una transazione siano trascritte in memoria di massa quando la transazione effettua il commit, mentre la politica *no-force* affida la scrittura delle pagine di una transazione ai meccanismi asincroni del buffer manager, consentendo che le scritture avvengano al di fuori dei confini transazionali.

La coppia di politiche *no-steal/no-force* è preferita nei DBMS, in quanto la politica *no-steal* è la più facile da realizzare e la politica *no-force* garantisce maggior efficienza.
Esiste poi la possibilità di "anticipare" i tempi di caricamento e scaricamento delle pagine, tramite *pre-fetching* e *pre-flushing*.

Relazione tra buffer manager e file system

Il file system è un modulo messo a disposizione dal sistema operativo; i DBMS ne utilizzano le funzionalità. Le funzioni tipicamente offerte dal file system e sfruttate dal DBMS sono:

- la creazione (**`create`**) e cancellazione (**`delete`**) di un file.
- La apertura (**`open`**) e chiusura (**`close`**) di un file, necessarie per caricare le informazioni che descrivono il file in opportune strutture di memoria centrale. La apertura di un file normalmente associa un identificatore numerico al nome del file.
- La primitiva **`read`**(**`fileid`**, **`block`**, **`buffer`**) per l'accesso diretto ad un blocco di un file, individuata tramite i primi due parametri, che viene trascritta nella pagina del buffer indicata.
- La primitiva **`read_seq`**(**`fileid`**, **`f-block`**, **`count`**, **`f-buffer`**) per l'accesso sequenziale a un numero fisso (count) di blocchi di un file.
- Le primitive duali di **`write`** e **`write-seq`**, caratterizzate esattamente dagli stessi parametri.

Inoltre, altre primitive consentono di strutturare la memoria di massa introducendo i *direttori(directories)*, in cui vengono allocati i file. Il file system è responsabile corrente di uso della memoria di massa, individuando quali blocchi sono liberi e quali sono allocati ai file, in modo da poter rispondere alle primitive.

Controllo di affidabilità

Il controllo della affidabilità garantisce due proprietà fondamentali delle transazioni: la atomicità e la persistenza. Il controllore dell'affidabilità è responsabile della scrittura del log, u archivio persistente che registra le varie azioni svolte dal DBMS. 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 della affidabilità

Il controllore della affidabilità è responsabile di realizzare i comandi transazionali begin transaction, commit work, rollback work e di realizzare le primitive di ripristino dopo i malfunzionamenti, detti rispettivamente *ripresa a caldo e ripresa a freddo*. Inoltre, il controllore di affidabilità riceve richieste di accessi a pagine in lettura e scrittura, che passa al buffer manager, e genera altre richieste di letture e scritture di pagine necessarie a garantire la robustezza e la resistenza ai guasti. Infine il controllore dell'affidabilità predispone i dati necessari per eseguire i meccanismi di ripristino dai guasti, in particolare realizzando azioni di *checkpoint* e di *dump*.

Memoria stabile

Per poter operare, il controllore della affidabilità deve disporre di *memoria stabile*, cioè di memoria che risulti resistente ai guasti. La memoria stabile è una astrazione, in quanto nessuna memoria può essere associata ad una probabilità di fallimento nullo; nonostante ciò, i meccanismi di controllo di affidabilità vengono definiti come se la memoria stabile fosse effettivamente esente da guasti. A seconda dei casi, la memoria stabile viene realizzata in modi diversi, con nastri, combinazioni nastro – disco, due unità a disco a "specchio".

Organizzazione del log

Il log è un file sequenziale di cui è responsabile il controllore della affidabilità scritto in maniera stabile. Sul log vengono registrate le azioni svolte dalle varie transazioni, nell'ordine temporale di esecuzione delle azioni stesse. 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. Ogni transazione inserisce nel log un record di begin, vari record relativi alle azioni effettuate (**`insert`**, **`delete`**, **`update`**) e un record di **`commit`** oppure di **`abort`**.

I record di sistema indicano l'effettuazione di operazioni di *dump* e di *checkpoint*, che illustreremo in dettaglio più avanti.

Struttura dei record nel log

I record di log che vengono scritti per descrivere l'azione di una transazione t_i sono:

- i record di **`begin`**, **`commit`** e **`abort`**, che contengono, oltre all'indicazione del tipo di record, anche l'identificativo T della transazione.
- 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, detto *before state,* e successivamente alla modifica, detto *after state.*
- I record di insert e di delete sono analoghi a quelli di update, da cui si differenziano per l'assenza nei primi del bifore state e nei secondi dell'after state.

Nel seguito, useremo i simboli B(T), A(T), C(T) per denotare i record di begin, abort e commit e U(T,O,BS,AS), I(T,O,AS) e D(T,O,BS) per denotare i record di update, insert e delete.

Questi record consentono di disfare e rifare le rispettive azioni sulla base di dati.

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

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

Checkpoint e dump

Un checkpoint è una operazione che viene svolta periodicamente, con l'obiettivo di registrare quali transazioni sono attive. L'operazione viene coordinata col buffer manager; si registrano gli identificatori delle transazioni in corso, e contemporaneamente si trasferiscono tutte le pagine nel buffer relative a transazioni che hanno già effettuato il commit o l'abort in memoria di massa tramite opportune operazioni di flush. Dopo aver iniziato un checkpoint non vengono accettate operazioni di commit da parte delle transazioni attive. L'operazione termina, dopo il completamento

delle operazioni di flush, scrivendo in modo sincrono (force) nel log un record di checkpoint che contiene gli identificatori delle transazioni attive.
Un *dump* è una copia completa della base di dati, che viene normalmente effettuata in mutua esclusione con tutte le altre transazioni quando il sistema non è operativo. La copia viene memorizzata su memoria stabile, tipicamente su nastro, ed è anche chiamata backup. Dopo la conclusione dell'operazione di dmp viene scritto nel log un record di dump, che segnala appunto la presenza di una copia fatta in un determinato istante; dopodiché il sistema riprende a funzionare normalmente.
Nel seguito useremo DUMP per denotare il record di dump e CK(T1,T2,…,Tn) per denotare un record di checkpoint.

Gestione delle transazioni

Durante il funzionamento normale delle transazioni, il controllore della affidabilità deve garantire che siano seguite due regole, che definiscono i requisiti minimi che consentono di ripristinare la correttezza della base a fronte di guasti.
La regola WAL (dall'inglese Write Ahead Log) impone che la parte bifore state dei record di un log venga scritta nel log prima di effettuare la corrispondente operazione sulla base di dati. Questa regola consente di disfare le scritture già effettuate in memoria di massa da parte di una transazione che non ho ancora effettuato un commit.
La regola di Commit-Precedenza impone che la parte after state dei record di un log venga scritta nel log prima di effettuare il 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.
Scrittura congiunta di log e base di dati.
Le due regole WAL e Commit-Precedenza impongono i seguenti protocolli per la scrittura del log e della base di dati:

- nel primo schema la transazione scrive inizialmente il record B(T), quindi svolge le sue azioni di update scrivendo prima il log U(T,O,BS,AS) e successivamente la pagina della base di dati. Tutte queste pagine devono essere effettivamente scritte (con la primitiva flush o force). In questo modo, al commit tutte le pagine della base di dati modificate sono certamente scritte in memoria di massa; questo schema non richiede operazioni di redo.
- Nel secondo schema, la scrittura dei record di log precede quella della azioni sulla base di dati, che però avvengono dopo la decisione di commit e la conseguentente scrittura sincrona del record di commit sul log; questo schema non richiede operazioni di Undo.
- nel terzo schema, più generale e comunemente usato, 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 commi 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.

Le azioni di scrittura del log da parte del controllore della affidabilità hanno un costo, paragonabile al costo di aggiornare la base di dati; in effetti, l'uso di protocolli transazionali robusti rappresenta un sensibile sovraccarico per il sistema, ma è irrinunciabile per garantire le proprietà "acide" delle transazioni.

Gestione dei guasti

Dal punto di vista di un DBMS, i guasti si suddividono in due casi:

- guasti di sistema: sono guasti indotti da "bachi software", ad esempio del sistema operativo, oppure da interruzioni del funzionamento dei dispositivi ad esempio dovuti a cali di tensione. Si traducono in una perdita del contenuto della memoria centrale (buffer), mantenendo invece valido il contenuto della memoria di massa (base di dati e log).

- Guasti di dispositivo: sono guasti relativi ai dispositivi di gestione della memoria di massa. Data la nostra assunzione che il log venga scritto sulla memoria stabile, i guasti di dispositivo 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: quanto il sistema individua un guasto, sia di sistema che di dispositivo, esso forza un arresto completo delle transazioni e il successivo ripristino del corretto funzionamento del sistema operativo (boot). Quindi viene attivata una procedura di ripresa, detta ripresa a caldo nel caso di guasto di sistema e ripresa a freddo nel caso di guasto di dispositivo.
Con questo modello, il guasto è un evento istantaneo che accade ad un certo istante dell'evoluzione della base di dati. Esisteranno transazioni potenzialmente attive all'atto del guasto, cioè delle quali no si conosce se abbiano ultimato le loro azioni sulla base di dati, e queste si classificano in due categorie in base all'informazione presente nel logo. Alcune di esse hanno effettuato il commit, e per loro è necessario rifare le azioni al fine di garantire la persistenza. Altre non hanno effettuato il commit, e per loro è necessario disfare le azioni. Si noti che alcuni sistemi aggiungono al log un altro record, detto record di end, quando le operazioni di trascrizione (flush) delle pagine ad opera del buffer manager sono terminate.

Ripresa a caldo

La ripresa a caldo è articolata in quattro fasi successive:

1. si accede all'ultimo blocco di log, che era corrente all'istante del guasto, e si ripercorre all'indietro il log fino al record di checkpoint.
2. si decidono le transazioni da rifare o disfare. 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 aventi, aggiungendo all'insieme di UNDO tutte le transazioni di cui è presente un record di begin, e spostando dall'insieme UNDO all'insieme di REDO tutti gli identificativi delle transazioni di cui è presente un record di commit.
3. si ripercorre all'indietro il log disfacendo le transazioni nel seti di UNDO, risalendo fino alla prima azione della transazione più "vecchia" nei due insiemi di UNDO e REDO; (procedendo eventualmente a precedere il record di checkpoint).
4. Si applicano le azioni di redo nell'ordine in cui sono registrate nel log.

Questo meccanismo garantisce atomicità e persistenza delle transazioni.
Ripresa a freddo
La ripresa a freddo risponde ad un guasto che provoca il deterioramento di una parte della base di dati; è articolata in tre fasi successive:

1. Durante la prima 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 aventi 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. si svolge una ripresa a caldo.

Strutture fisiche di accesso

Le strutture fisiche di accesso descrivono il modo in cui vengono organizzati i dati per garantire operazioni di ricerca e di modifica efficienti da parte dei programmi applicativi. In genere, ciascun DBMS ha a disposizione un numero abbastanza limitato di tipi di strutture di accesso.

Architettura del gestore degli accessi

Il gestore degli accessi è responsabile di trasformare un piano di accesso, prodotto dall'ottimizzatore, in opportune sequenza di accessi alle pagine della base di dati. Esso utilizza i cosiddetti metodi d'accesso, cioè opportuni moduli software che contengono primitive per l'accesso e la manipolazione dei dati specifici di ciascuna organizzazione fisica.
Il metodo di accesso conosce inoltre la disposizione delle tuple nelle pagine, ed è quindi in grado di individuare specifici valori all'interno di una pagina.

Gestione delle tuple nelle pagine

Sebbene ciascun metodo di accesso possa avere una propria organizzazione delle pagine, alcuni metodi di accesso hanno caratteristiche comuni, descritte nel seguito. In ciascuna pagina, sono presenti sia informazioni utili sia informazioni di controllo; l'informazione utile coincide con i dati veri e propri, l'informazione di controllo consente di accedere all'informazione utile.

- Ogni pagina, in quanto coincidente con un blocco di memoria di massa, ha un aparte iniziale (block header) e finale (block trailer) contenente informazione di controllo utilizzata dal file system.
- Ogni pagina, in quato appartenente ad una struttura di accesso, ha poi una parte iniziale (page header) e finale (page trailer) contenente informazione di controllo relativa al metodo di accesso.
- Ogni pagina ha poi un suo dizionario di pagina, che contiene puntatori a ciascun dato utile elementare contenuto nella pagina, e una parte utile che contiene i dati.
- Infine ogni pagina contiene bit di parità, per verificare che l'informazione in essa contenuta sia valida.

Alcuni gestori delle pagine non consentono la separazione di una tupla su più pagine, e in tal caso la massima dimensione di una tupla è limitata dal massimo spazio utile disponibile in una pagina. In alcuni casi inoltre tutte le tuple hanno sa stessa dimensione. In altri casi le tuple possono avere lunghezze diverse; nel caso di tuple a lunghezza variabile, il dizionario di pagina contiene una indicazione degli offset di ciascuna tupla rispetto all'inizio della parte utile e di ciascun valore dei campi.

Le primitive offerte dal gestore delle pagine sono :

- Inserimento e aggiornamento
- Cancellazione
- Accesso ad una particolare tupla
- Accesso ad un campo di una particolare tupla

Strutture sequenziali

Passiamo ora ad analizzare il modo in cui le pagine vengono collegate fra di loro in strutture dati, partendo dalle organizzazioni sequenziali. Le strutture sequenziali sono caratterizzate da una disposizione sequenziale delle tuple in memoria di massa; il file è costituito da vari blocchi di memoria consecutivi, e le tuple vengono inserite nei blocchi rispettando una sequenza. In particolare, tale sequenza può essere di vari tipi:

- In una organizzazione entry-sequenced, la sequenza delle tuple è indotta dal loro ordine di immissione;
- In una organizzazione ad array, le tuple sono disposte come in un array, e la loro posizione dipende dal valore assunto in ciascuna tupla da un campo indice;
- In una organizzazione sequenziale ordinata, la sequenza delle tuple dipende dal valore assunto in ciascuna tupla da un campo di ordinamento, detto campo chiave (key).

Struttura sequenziale entry-sequenced

Una struttura sequenziale entry-sequenced è ottimale per svolgere operazioni di lettura e scrittura sequenziali. Il modo tipico di accedere al loro contenuto è tramite una scansione (scan) sequenziale. Le operazioni di caricamento iniziale dei dati i di inserimento avvengono alla fine del file e in modo sequenziale, e sono perciò assai efficienti; è sufficiente gestire un puntatore all'ultima tupla per poter eseguire l'operazione. Il principale problema è posto dalle operazioni di modifica e cancellazione.

Struttura sequenziale ad array

Una organizzazione sequenziale ad array è possibile quando le tuple di una tabella sono di dimensione fissa; in tal caso, al file viene associato un numero *n* di blocchi contigui e ciascun blocco è dotato di un numero *m* di slot disponibili per le tuple. Per il caricamento iniziale del file, gli indici vengono semplicemente ottenuti incrementando un contatore; tuttavia gli indici vengono semplicemente ottenuti incrementando un contatore; tuttavia, sono possibili inserimenti e

cancellazioni. Le cancellazioni creano slot liberi, gli inserimenti devono essere svolti negli slot liberi o al termine del file.

Struttura sequenziale ordinata

L'organizzazione sequenziale ordinata assegna a ciascuna tupla una posizione in base al valore del campo chiave. Questa struttura, sebbene classica e molto studiata, non è più stata usata in tempi recenti, perché ha costi di gestione assai pesanti. Essa è basata sul dare alle tuple un ordinamento fisico che riflette l'ordinamento lessicografico dei valori presenti nel campo chiave. Le strutture sequenziali ordinate venivano usate su dispositivi sequenziali; esse erano costruite da processi di batch, i quali dovevano ordinare le registrazioni in base alla chiave e a caricarle sequenzialmente in un file. Le modifiche venivano collezionate in *file differenziali*, anch'essi ordinati in base al valore della chiave, e periodicamente elaborate da processi batch, i quali procedevano a fondere il file principale e il file differenziato.

Il principale problema si ha quando si devono inserire nuove tuple in quanto queste modifiche comportano un riordino delle tuple già presenti in memoria. Per evitare i riordini sono possibili le seguenti tecniche:

- Prevede a priori un certo numero di slot liberi all'atto di effettuare il caricamento iniziale, in modo da consentire il mantenimento della organizzazione sequenziale con operazioni di "riordino locale".
- Integrare il file sequenziale ordinato con un *file di overflow*, dedicato a gestire ogni nuova tupla che richieda maggiori spazi. I blocchi del file di overflow sono collegati fra loro in una *catena di overflow*.

Strutture con accesso calcolato

Una struttura con accesso calcolato garantisce, al pari della struttura sequenziale ordinata, un accesso *associativo* ai dati, in cui cioè la locazione fisica dei dati dipende dal valore assunto da un campo chiave; La struttura viene realizzata allocando al file un numero B di blocchi, spesso contigui; Il gestore di questo metodo di acceso dispone di un *algoritmo di calcolo (hash)* che, una volta applicato alla chiave, restituisce un numero compreso tra 0 e B –1; tale numero viene interpretato come numero del blocco nell'ambito dei blocchi allocati al file.

Questa struttura è ideale quando l'applicazione vuole accedere alla tupla che contiene uno specifico valore della chiave, in quanto l'indirizzo prodotto dall'algoritmo di calcolo viene passato al gestore dei buffer per accedere direttamente al blocco così identificato. In modo analogo, al scrittura di una tupla che contiene un determinato valore di chiave viene effettuata in quel blocco.

La primitiva che consente di trasformare un valore di chiave in un numero di blocco riceve come parametri il nome del file e il valore di chiave, e restituisce un numero di blocco: **`hash`**(**`Fileid`**, **`Key`**):**`Blockid`**. La corrispondente funzione offerta dal sistema consta di due parti:

- Una prima operazione, detta folding, opera sui valori di chiave così che essi si trasformino in valori interi positivi, possibilmente distribuiti in modo uniforme.
- La successiva operazione di *hashing* trasforma il numero positivo binario individuato precedentemente in un numero intero tra 0 e B-1.

Questa tecnica funziona bene se viene previsto un baso coefficiente di riempimento, cioè se sis sovradimensiona il file.

Infatti il problema principale delle strutture gestite con hashing è quello delle collisioni, cioè situazioni in cui lo stesso numero di blocco viene ritornato dall'algoritmo a partire da due valori diversi della chiave. In genere, questo fenomeno non è negativo: infatti ciascuna pagina può contenere al massimo F tuple. Tuttavia, quando si eccede il valore F la situazione è critica, perché si deve ricorrere ad una tecnica differente per allocare e ritrovare le tuple con non trovano posto nella loro destinazione naturale. Quando si verifica un numero eccessivo di collisioni e la capacità della pagina viene esaurita, al soluzione proposta prevede la costruzione di catene di overflow. Ovviamente la presenza di catene di overflow rallenta il tempo di ricerca, in quanto sarà talvolta necessario richiedere una operazione di i/o per ogni blocco della catena.

In conclusione notiamo che l'hashing è la tecnica più efficiente per accedere a dati in base a predicati di uguaglianza con valori definiti nella interrogazione, ma risulta assai inefficiente per interrogazioni che richiedono l'accesso ad intervalli di valori.

Strutture ad albero

Le strutture ad albero, di tipo B (B-tree) oppure B+ (B+-tree), sono le più frequentemente usate nei DBMS di tipo relazionale. Esse consentono accessi associativi, in base ad un valore di uno o più attributi chiave (key). La chiave può essere costituita da molti attributi, ma per semplicità considereremo nel seguito chiavi formate da un solo attributo.
Quando un utente specifica a livello DDL la definizione di un indice relativo a un attributo o una lista di attributi di una tabella, il sistema genera una opportuna struttura ad albero per la gestione dei dati. Ogni albero è caratterizzato da un nodo radice, vari nodi intermedi, e vari nodi foglia; I legami tra nodi vengono stabiliti da puntatori che collegano fra loro le pagine. Un altro requisito importante per il buon funzionamento di queste strutture è che gli alberi siano bilanciati, cioè la lunghezza di un cammino che collega il nodo radice da un qualunque nodo foglia sia costante.
Contenuto dei nodi e tecnica di ricerca
Ciascun nodo presenta una sequenza di F valori ordinati di chiave. Ogni chiave Ki <=i<=F è seguita da un puntatore Pi; K1 è preceduta da un P0. Ciascun puntatore indirizza un sotto-albero così caratterizzato:

- Il puntatore P0 indirizza al sotto-albero che contiene l'informazione relativa alle chiavi strettamente inferiori a K1;
- Il puntatore Pf indirizza al sotto-albero che contiene l'informazione relativa alle chiavi superiori o uguali a Kf;
- Ciascun puntatore intermedio Pi, 0 < i < F, indirizza un sotto-albero che contiene tutta la informazione relativa alle chiavi Kj comprese nell'intervallo Ki <= Kj < Ki+1.

In sintesi, ciascun nodo contiene F valori di chiave e F + 1 puntatori; il valore F + 1 viene detto fan-out dell'albero.
La tipica primitiva di ricerca che viene messa a disposizione dal gestore degli alberi consente un accesso associato alla tupla o alle tuple che contengono un certo valore di chiave V. Il meccanismo di ricerca consiste nel seguire i puntatori, già opportunamente predisposti, partendo dalla radice. Ad ogni nodo intermedio:

- Se V < K1 si segue il puntatore P0;
- Se V >= Kf si segue il puntatore Pf;
- Altrimenti, si segue il puntatore Pj tale che Kj <= V < Kj+1.

La ricerca prosegue in questo modo fino ai nodi foglia dell'albero, che possono essere organizzati in due modi diversi:

- Nel primo caso, i nodi foglia contengono l'intera tupla. La struttura dati che si ottiene in questo caso è detta *key-sequenced*; in essa la posizione di una tupla è vincolata dal valore assunto dal suo campo chiave.
- Nel secondo caso, ciascun nodo foglia contiene puntatori ai blocchi della base di dati che contengono tuple con il valore di chiave specificato. La struttura dati che si ottiene in questo caso è detta *indiretta*, il posizionamento delle tuple del file può essere qualsiasi, quindi in particolare questo meccanismo consente di indirizzare tuple allocate tramite un qualunque altro meccanismo "primario".

In alcuni casi, la struttura ad indice non è completa; non vengono cioè compresi nell'indice tutti i valori della chiave, ma solo alcuni; il tal caso si dice che l'indice è sparso. Un indice sparso viene normalmente costruito su una struttura sequenziale ordinata, in modo da consentire di localizzare via indice un valore di chiave prossimo al valore ricercato, e svolgere successivamente una ricerca di tipo sequenziale.
La struttura sequenziale key-sequenced è generalmente preferita per realizzare il cosiddetto *indice primario*, cioè quello che viene normalmente definito di tipo inique sulla chiave primaria.

La struttura indiretta è viceversa preferita per realizzare i cosiddetti *indici secondari*, che possono essere di tipo inique p multiple;
Gli inserimenti e le cancellazioni di tuple provocano anche aggiornamenti degli indici, che devono riflettere la situazione generata da una variazione dei valori del campo-chiave. Un inserimento non provoca problemi quando è possibile inserire il nuovo valore della chiave in una foglia dell'albero, la cui pagina ha uno slot libero; quando invece la pagina della foglia non ha spazio disponibile, si rende necessaria una operazione di split, che suddivide l'informazione già presente nella foglia e la nuova informazione in due, in modo equilibrato, allocando due foglie al posto di una.
Una cancellazione può essere sempre fatta in loco, marcando uno slot precedentemente allocato ad una tupla come invalido. Vi sono però due problemi:

- Quando la cancellazione coinvolge uno dei valori di chiave presenti nell'albero, è opportuno recuperare il successivo valore dalla base di dati e introdurlo al posto del valore cancellato. In questo modo, tutti i valori presenti nell'albero B+ appartengono anche alla base di dati.
- Quando la cancellazione lascia due pagine contigue al livello foglia così inutilizzate da consentire che tutta l'informazione in esse presente venga concentrata in una unica pagina, si può svolgere l'operazione di merge, duale alla operazione di split, che colleziona tutta l'informazione di due pagine in una sola pagina.

La modifica del valore di un campo chiave viene trattata come una cancellazione del suo valore iniziale seguito da un inserimento del suo valore finale.
L'uso attento delle operazioni di split e merge consente di mantenere un riempimento medio di ciascun nodo superiore al 50%.
Inoltre anche se l'albero è inizialmente bilanciato, possono nascere differenze fra le lunghezze dei cammini, che possono indicare la opportunità di ri-bilanciare l'albero. Il ri-bilanciamento dell'albero è una operazione che viene tipicamente decisa dall'amministratore della base di dati di fronte ad un degradare delle prestazioni di accesso ai dati; infatti, un albero perfettamente bilanciato garantisce le migliori prestazioni medie.

Distinzione fra alberi B e B+

Negli alberi B+ i nodi foglia sono collegati da una catena che li connette in base all'ordine imposto dalla chiave. Tale catena consente di svolgere in modo efficiente anche interrogazioni il cui predicato di selezione definisce un *intervallo* di valori ammissibili.
In particolare, questa struttura dati consente anche una scansione ordinata in base ai valori di chiave dell'intero file, che risulta abbastanza efficiente. Per questa sua versatilità, nei DBMS è maggiormente usata la struttura B+-tree.
Nelle strutture B-tree non viene previsto di collegare sequenzialmente i nodi foglia. In tal caso, una ottimizzazione possibile prevede di usare nei nodi intermedi due puntatori per ogni valore di chiave Ki; uno dei due puntatori viene utilizzato per puntare direttamente al blocco che contiene la tupla corrispondente a Ki, interrompendo la ricerca; l'altro puntatore serve per proseguire la ricerca nel sottoalbero che comprende i valori di chiave strettamente compresi fra Ki e Ki+1. Questa tecnica fa risparmiare spazio nelle pagine dell'indice e consente talvolta di terminare la ricerca senza dover percorrere tutti i livelli.
L'efficienza di un albero B o B+ è normalmente elevata in quanto spesso le pagine che memorizzano i primi livelli dell'albero risiedono nel buffer per effetto di altre transazioni.
Una ottimizzazione dello spazio occupato avviene tramite la compressione dei valori di chiave, ad esempio mantenendo solo i loro prefissi nei livelli alti dell'albero, ove si svolge la fase iniziale della ricerca, e solo i loro suffissi, a pari prefisso, nei livelli bassi dell'albero, ove si svolge la parte finale della ricerca.

10. Architetture distribuite

La quasi totalità delle applicazioni delle basi di dati si colloca naturalmente in un contesto distribuito, cioè caratterizzato dalla presenza di una rete.
L'architettura distribuita più semplice ma anche più diffusa utilizza il paradigma *client-server*; questa architettura è basata sulla tecnologia dei server per la gestione dei dati, descritta nel capitolo precedente, ma rientra nel panorama vastissimo delle architetture distribuite per basi di dati.
Problemi completamente diversi vengono posti dalle *basi di dati distribuite*; questo contesto è caratterizzato dalla presenza di almeno due server che possono anche svolgere transazioni autonomamente, però in qualche caso devono interagire.
Un'altra tipologia di architetture per basi di dati utilizza il parallelismo per migliorare le prestazioni. Si hanno pertanto le cosiddette *basi di dati parallele*, caratterizzate dall'uso di macchine multiprocessore e di molteplici dispositivi per la gestione dei dati.
Le architetture più recenti per basi di dati sono via via specializzate in base alle applicazioni dominanti; in particolare, sta sempre più emergendo una separazione dei DBMS in due componenti, uno dedicato alla gestione in linea dei dati e uno responsabile del "supporto delle decisioni". Sono quindi emerse due tipologie di architetture hardware/software:

- Alcuni sistemi sono finalizzati alla gestione ottimizzata ed affidabile di transazioni, gestendo il cosiddetto *On-Line Transaction Processing*, OLPT. Questi sistemi vengono dimensionati per gestire centinaia o addirittura migliaia di transazioni al secondo.
- Altri sistemi sono finalizzati alla analisi dei dati gestendo il cosiddetto *On-Line Analitical Processing*, OLAP. Per far questo è necessario esportare i dati dai sistemi OLTP, ove essi vengono generati, e importarli nelle cosiddette *data warehouse*.

I server tipicamente contengono al loro interno sia funzioni OLTP per gestire molteplici transazioni al secondo, sia funzioni OLAP per ottimizzare interrogazioni complesse. Tuttavia la separazione fra OLAP e OLTP consente di meglio organizzare e dimensionare i server, specializzandoli per funzione.
Recentemente è emersa una tecnologia di servizio per la realizzazione di applicazioni distribuite, specie in presenza della separazione tra ILTP e OLAP; quella della *replicazione dei dati*, intesa come capacità di costruire copie di collezioni di dati, esportandole da un nodo a un altro di un sistema distribuito, in modo da massimizzare la disponibilità dei dati e anche aumentarne la affidabilità. Prodotti specifici sono in grado di garantire la gestione di copie in modo trasparente rispetto ai programmi che aggiornano gli originali.
I un quadro tecnologico caratterizzato dalla necessità di far interagire fra loro sistemi e prodotti diversi, assumono inoltre grande importanza i problemi di portabilità e di interoperabilità.

- La portabilità denota la possibilità di trasportare programmi da un ambiente a un altro
- La interoperabilità denota la possibilità di far interagire sistemi eterogenei.

Per ottenere portabilità e introperabilità assumono grande importanza gli standard; in particolare, la portabilità dipende dagli standard relativi ai linguaggi, mentre la interoperabilità dipende dagli standard relativi ai protocolli di accesso ai dati.

Architettura client-server

Il paradigma client-server è un modello di interazione tra processi software, ove i processi interagenti si suddividono tra client, che richiedono servizi, e server, che offrono servizi. L'interazione client-server richiede perciò una precisa definizione di una interfaccia di servizi, che elenca i servizi messi a disposizione dal server.
Non è necessario che i processi server e client siano allocati su macchine diverse: la distribuzione fra processi è un ottimo paradigma per la costruzione del software indipendentemente dalla allocazione dei processi.
Vari motivi spingono verso l'uso di architetture client-server per basi di dati:

- Le funzioni di client e server sono ben identificate nel contesto delle basi di dati.

- Oltre alla decomposizione funzionale dei processi e dei compiti, nella basi di dati l'utilizzo di elaboratori diversi per client e server è particolarmente conveniente.
- Il linguaggio SQL, diffuso in tutte le basi di dati relazionali, offre un paradigma di programmazione ideale per identificare la "frontiera dei servizi". Le interrogazioni SQL vengono infatti formulate dal client e inviate al server; i risultati della interrogazione vengono calcolati dal server e restituiti al client. Sulla rete viaggia così una informazione compatta.

La architettura client-server si adatta sia a interrogazioni compilate staticamente sia a interrogazioni con SQL dinamico. Con un processo statico le interrogazioni vengono sottomesse al server una volta e poi richiamate molte volte; meccanismi quali i cursori, consentono di richiamare le interrogazioni dai processi client, con chiamate a procedure e/o servizi remoti. Con un processo dinamico le interrogazioni vengono trasmesse sotto forma di stringhe di caratteri che vengono compilate ed eseguite dal serve. In entrambi i casi, l'ottimizzatore e i metodi di acceso risiedono sul server.
In genere, un server può ricevere una interrogazione parametrica, in cui cioè il processo client assegna valori ad alcuni parametri di ingresso e poi richiama l'esecuzione di una interrogazione o procedura. Spesso il server che gestisce tali richieste è multi-threaded: si comporta cioè come un unico processo che opera dinamicamente per conto di differenti transazioni; ciascuna unità di esecuzione del processo server per conto di una transazione è detto thread. I server possono gestire le code o direttamente o tramite altri processi denominati dispatcher. L'architettura illustrata fin'ora viene denominata architettura a due componenti in quanto in essa sono presenti un client, con funzioni sia di interfaccia utente sia di gestione della applicazione, e un server dedicato alla gestione dei dati. Recentemente di gestire la logica applicativa comune a più client.

Basi di dati distribuite

Abbiamo visto che in una architettura client-server una transazione coinvolge al più un server; quando viceversa le transazioni coinvolgono più server, parliamo di *base di dati distribuita.* In questo paragrafo intraprendiamo lo studio delle basi di dati distribuite da un punto di vista funzionale, studiando cioè il modo in cui un utente può specificare interrogazioni distribuite.

Applicazioni delle basi di dati distribuite

La gestione distribuita dei dati si contrappone a una gestione centralizzata tipica dei grandi centri di calcolo, dominante fino alla metà degli anni Ottanta, e consente la elaborazione e il controllo dei dati negli ambienti ove essi sono generati e maggiormente usati. I sistemi distribuiti possono essere configurati aggiungendo e modificando progressivamente gli elaboratori che li compongono, con una flessibilità e modularità ben maggiore dei sistemi basati sull'uso di "mainframe": per essendo maggiormente esposti ai guasti per la loro maggior complessità strutturale, sono infatti capaci di rispondere ai guasti con un degradamento delle prestazioni ma senza un blocco completo.
Una prima classificazione delle basi di dati distribuite considera il tipo di DBMS e di rete utilizzato. Quando tutti i server utilizzano lo stess DBMS, la base di dati si dice *omogenea*, altrimenti si dice *eterogenea*. Una base di dati distribuita può utilizzare una *rete locale* (LAN) oppure una *rete geografica* (WAN).
I sistemi omogenei su LAN corrispondono a soluzioni tecnologicamente più semplici e più diffuse, presenti in un gran numero di applicazioni. La soluzione eterogenea è molto diffusa: molti sistemi informativi intersettoriali integrati presenti nelle aziende sono eterogenei.

Autonomia locale e cooperazione

Da un punto di vista astratto, una base di dati distribuita può essere considerata come una unica raccolta di dati. Tuttavia è importante notare che ciascun server è in grado di gestire in modo autonomo applicazione e garantire autonomia dei server è uno dei principali obiettivi nella gestione di questi sistemi. La motivazione che spinge a costruire le basi di dati distribuite non è perciò quello di massimizzare l'interazione e la necessità di trasmettere in rete.

Frammentazione e allocazione dei dati

La frammentazione dei dati consente di organizzare i dati stessi in modo tale da garantire una loro distribuzione efficiente e ben organizzata. Consideriamo una relazione R; la sua frammentazione consiste nel determinare un certo numero di frammenti Ri, ottenuti applicando ad R operazioni algebriche. La frammentazione può essere di due tipi, orizzontale e verticale.

- Nella frammentazione orizzontale, i frammenti Ri sono insiemi di tuple con lo stessa schema della relazione R; ciascun frammento orizzontale può essere interpretato come il risultato di una selezione applicata alla relazione R.
- Nella frammentazione verticale, i frammenti Ri hanno uno schema ottenuto come sottoinsieme dello schema di R; ciascun frammento verticale può essere interpretato come il risultato di una proiezione applicata alla relazione R.

La frammentazione è corretta se valgono le seguenti proprietà:

- Completezza: ogni dato della relazione R deve essere presente in un quale suo frammento Ri;
- Ricostruibilità: la relazione deve essere interamente ricostruibile a partire dai suoi frammenti.

Normalmente i frammenti orizzontali sono disgiunti, cioè non hanno tuple in comune; viceversa i frammenti verticali includono la chiave primaria definita per Ri in modo da garantire la ricostruibilità.

Ogni frammento Ri viene implementato tramite un file fisico e installato presso un server; si dice che i frammenti vengono allocati sui server. Quindi, la relazione è presente in modo virtuale, mentre i frammenti sono effettivamente memorizzati.

Lo schema di allocazione contiene il mapping dei frammenti o da intere relazioni ai server che li memorizzano, consentendo il passaggio da una descrizione logica a una descrizione fisica dei dati. Tale mapping può essere:

- *Non ridondante*, quando ciascun frammento o relazione viene allocato esattamente su un server;
- *Ridondante*, quando qualche frammento o relazione viene allocato su più di un server.

Livelli di trasparenza

La distinzione tra frammentazione e allocazione consente di individuare vari livelli di trasparenza nelle applicazioni, cioè vari modi di scrivere applicazioni. I livelli di trasparenza più significativi sono tre: trasparenza di frammentazione, di allocazione, e di linguaggio. Vi è poi la possibilità di assenza di trasparenza, quando non esiste un linguaggio comune per accedere ai dati presenti nel sistema, e i programmatori devono quindi rivolgersi a ciascun server con uno specifico “dialetto”.

Trasparenza di frammentazione: a questo livello, il programmatore non si deve preoccupare del fatto che la base di dati sia o meno distribuita e rammentata;

Trasparenza di allocazione: a questo livello, il programmatore conosce la struttura dei frammenti, ma non deve indicarne la allocazione. In particolare, se il sistema ammette frammenti replicati, il programmatore non deve neppure indicare quale copia viene scelta per l’accesso.

Trasparenza di linguaggio: a questo livello, il programmatore deve indicare nella sua interrogazione sia la struttura dei frammenti sia la loro allocazione.

Assenza di trasparenza: in questo caso, ciascun DBMS accetta un proprio “dialetto” SQL, in quanto il sistema è eterogeneo e i DBMS non supportano uno standard di interoperabilità comune.

Classificazione delle transazioni

Uno schema di classificazione delle transazioni basato sulla composizione degli statement SQL che compongono una transazione è stato proposto per i DBMS della IBM e successivamente adottato da vari altri costruttori. Il client può ovviamente indirizzare transazioni locali al DBMS, cioè transazioni composte da interrogazioni le cui tabelle sono tutte allocate su quel DBMS; può poi indirizzare transazioni non locali, che cioè coinvolgono dati memorizzati su altri DBMS.

Queste transazioni, delimitate al solito dai comandi begin transaction ed end transaction, possono essere più o meno complesse:

- Le richieste remote sono transazioni di sola lettura indirizzate a un solo DBMS remoto.
- Le transazioni remote sono transazioni costituite da un numero qualsiasi di comandi SQL dirette a un solo DBMS remoto.
- Le transazioni distribuite sono transazioni rivolte a un numero generico di DBMS ma in cui ciascun comando SQL fa riferimento a dati memorizzati su di un solo DBMS.
- Le richieste distribuite sono transazioni arbitrarie, costituite da un numero arbitrario di conadi SQL, in cui ciascuna query può far riferimento a dati distribuiti su qualunque DBMS.

Questa classificazione è importante perché individua livelli progressivi di complessità nell'interazione fra DBMS. Nel primo caso il DBMS remoto può essere solo interrogato; nel secondo caso è possibile operare scritture; nel terzo caso è possibile includere in una transazione scritture su più nodi ma ciascuna interrogazione SQL deve essere distribuita su più nodi.

Tecnologia delle basi di dati distribuite

La distribuzione dei dati non influisce su due proprietà acide delle transazioni: consistenza e persistenza.

- La consistenza delle transazioni non dipende dalla distribuzione dei dati, in quanto i vincoli di integrità descrivono solo proprietà locali a un DBMS.
- In modo del tutto analogo, la persistenza non è un problema che dipende dalla distribuzione dei dati, perché ciascun sistema garantisce la persistenza anche in presenza di guasti dei dispositivi tramite meccanismi di dump e backup locali.

Ottimizzazione di interrogazioni distribuite

La ottimizzazione delle interrogazioni è richiesta solo quando un utente può sottomettere a un DBMS una richiesta distribuita. Il DBMS cui viene sottomessa l'interrogazione è responsabile della cosiddetta "ottimizzazione globale"; decide cioè la decomposizione della interrogazione in tante sotto-interrogazioni, ciascuna rivolta a uno specifico DBMS. Una strategia distribuita consiste nell'esecuzione coordinata di vari programmi sui valori DBMS e nello scambio di dati fra di essi; tra i fattori di costo di una interrogazione distribuita vi è anche la quantità di dati trasmessi in rete.
L'ottimizzatore globale decide il piano più conveniente esaminando un albero delle alternative. Come nei sistemi centralizzati, è necessario scegliere l'ordine delle operazioni e il loro metodo di esecuzione; in aggiunta, si deve anche definire la strategia di esecuzione delle operazioni i cui operandi sono allocati su nodi distinti, definendo quindi una strategia di trasmissione dei dati e di allocazione dei risultati. E' possibile associare a un nodo foglia un costo, che questa volta prevede il contributo di tre componenti:

$C_{total} = C_{I/O} \, x \, n_{I/O} + C_{cpu} \, x \, n_{cpu} + C_{tr} \, x \, n_{tr}$

I due nuovi elementi n_{tr} e C_{tr} misurano, rispettivamente, la quantità di dati trasmessi in rete e il costo di trasmissione unitario, e si aggiungono ai costi di elaborazione e di ingresso/uscita.

Controllo di concorrenza

Il controllo della concorrenza in ambito distribuito potrebbe riservare gravi difficoltà, ma la teoria del controllo di concorrenza ci riserva una sorpresa gradevole, sotto forma di proprietà che confermano la validità dei metodi di locking a due fasi e di timestamping.
In un sistema distribuito, una transazione t_i può eseguire varie sotto-transazioni t_{ij}, ove il secondo indice denota il nodo del sistema su cui la sotto-transazione opera. La serializzabilità locale preso gli scheduler non è garanzia sufficiente per la serializzabilità.

Serializzabilità globale

La serializzabilità globale di due transazioni distribuite richiede l'esistenza di un unico schedule seriale S, che coinvolga tutte le transazioni del sistema, e che sia equivalente a tutti gli schedule locali Si: per ogni nodo i deve accadere che la proiezione S[i] di S contenente le sole azioni che

avvengono sul nodo i sia equivalente allo schedule S. Questa proprietà è difficile da verificare nel caso si usino scheduler che applicano direttamente la view-equivalenza o la conflict-equivalenza, ma risulta immediatamente verificata quando si usino il locking a due fasi o il metodo di timestamping. Valgono infatti le seguenti proprietà:

- Se ciascuno scheduler della base di dati distribuita usa su ciascun nodo il metodo di locking a due fasi e svolge l'azione di commit in modo atomico in un istante in cui tutte le sotto-transazioni ai vari nodi detengono tutte le risorse, gli schedule risultanti sono globalmente serializzabili in base ai conflitti.
- Se un insieme di sotto-transazioni distribuite acquisisce un unico timestamp e usa tale timestamp nelle sue richieste a tutti gli scheduler che usano il controllo di concorrenza basato su timestamp, gli schedule risultanti sono globalmente seriali in base all'ordinamento indotto dai timestamp.

Metodo di Lamport per assegnare i timestamp

Per garantire il buon funzionamento del controllo di concorrenza basato su timestamp, è necessario che ciascuna transazione possa acquisire un timestamp che corrisponda all'istante di tempo in cui la transazione distribuita deve sincronizzarsi con le altre applicazioni; per questo si può usare il metodo di Lamport per assegnare timestamp che riflettano le relazioni di precedenza fra eventi in un sistema distribuito. Con questo metodo, un timestamp è caratterizzato da due gruppi di cifre; le meno significative identificano un nodo, le più significative identificano gli eventi che si succedono a ciascun nodo.

Rilevazione distribuita dei deadlock.

Quando i deadlock coinvolgono transazioni distribuite, la loro rilevazione è ovviamente più complessa, in quanto i deadlock possono essere dovuti a situazioni di attesa circolare che si verificano tra due o più nodi; ovviamente la soluzione semplice di usare timeout vale indipendentemente dal contesto in cui si creano i deadlock.

Atomicità di transazioni distribuite

Per garantire l'atomicità delle transazioni distribuite è necessario che tutti i nodi che partecipano ad una transazione giungano alla stessa decisione circa la transazione (commit o abort); è perciò necessario eseguire protocolli particolare, detti *protocolli di commit*, che consentano a una transazione di raggiungere correttamente la decisione di commit e abort.

Un sistema distribuito è soggetto alle *cadute di un nodo*, che possono avvenire su ogni elaboratore; il guasto su un nodo può essere soft oppure hard, come discusso nel capitolo predente. In aggiunta, è anche possibile che vengano *persi dei messaggi*, che lasciano l'esecuzione di un protocollo in situazione di incertezza. In genere, proprio per garantire l'avanzamento di un protocollo, ciascun messaggio del protocollo è seguito da un messaggio di risposta (detto "ack"); tuttavia, la perdita di uno dei due messaggi lascia il mittente nell'incertezza se il messaggio primario sia stato ricevuto o no.

Dato che i messaggi possono essere persi, i protocolli di commit pongono un tempo limite alla ricezione del messaggio di ack, trascorso il quale il mittente decide di procedere comunque con il protocollo. Infine è possibile che qualche collegamento della rete si interrompa; in tal caso, oltre alla perdita di messaggi, si può verificare un altro fenomeno: il partizionamento della rete in due sotto-reti che non comunicano fra loro. Questo guasto può causare ulteriori problemi, in quanto è possibile che una transazione risulti contemporaneamente attiva in più di una sotto-rete-

Protocollo di commit a due fasi

Il protocollo di commit a due fasi ricorda, nelle sue linee essenziali, un matrimonio. La decisione di due persone viene raccolta e ratificata da una terza persona. Il ruolo di promesso sposo è assegnato a ciascun server che partecipa a una transazione; in questo contesto, i server vengono denominati *resource manager (RM).* Il celebrante è viceversa associato a un processo, detto *transaction manager (TM);* Il protocollo di commit a due fasi si svolge tramite un rapido scambio di messaggi

fra TM e RM; per rendere il protocollo resistente ai guasti, RM e TM scrivono alcuni nuovi record nei loro log.

Nuovi record nel log

I nuovi record nel log scritti durante il protocollo di commit a due fasi estendono i tipi di record visti; Sia TM e RM sono dotati di un proprio log. Il TM scrive:

- Il record di **`prepare`**, che contiene l'identità dei processi RM;
- Il record di **`global commit`** o di **`global abort`**. Anticipando il protocollo, notiamo che l'istante di tempo in cui il TM scrive nel suo record di log il record di global commit esprime la decisione, atomica e persistente, di portare a termine con successo l'intera transazione, rendendo visibile il suo stato finale;
- Il record di **`complete`**, che è scritto alla conclusione del protocollo di commit a due fasi.

Il processo RM rappresenta una sotto-transazione; come nel contesto centralizzato, ogni RM scrive un record di begin all'inizio, seguito da vari record di **`insert`**, **`delete`** e **`update`** che rappresentano le operazioni svolte dalla sotto-transazione.
Per quanto concerne il protocollo di commit a due fasi, sull'RM vi è un solo nuovo record:

- Il record di **`ready`**, che indica la irrevocabile disponibilità di partecipare al protocollo di commit a due fasi contribuendo a una decisione di commit. Su tale record viene scritto anche l'identificativo del TM.

Protocollo in assenza di guasti

In assenza di guasti, il protocollo di commit a due fasi consiste in una rapida sequenza di scritture sul log e di scambi di messaggi tra TM e RM; nella comunicazione con gli RM, il TM può utilizzare meccanismi di *broadcast*, cioè che trasmettono lo stesso messaggio a molti nodi.
La *prima fase* del protocollo si articola nel modo seguente:

1. il TM scrive il record di **`prepare`** nel suo log e invia un messaggio di **`prepare`** per informare tutti gli RM dell'inizio del protocollo;
2. gli RM che sono in uno stato affidabile attendono il messaggio di **`prepare`**; non appena il messaggio arriva, scrivono sul proprio log il record di **`ready`** e trasmettono al TM un messaggio di **`ready`**, che indica la scelta favorevole di partecipare al commit. Se invece un RM non è in uno stato affidabile, invia un messaggio di **`not-ready`** e termina il protocollo;
3. il TM colleziona i messaggi di risposta dagli RM. Se da tutti gli RM riceve un messaggio positivo, scrive sul log un record di **`global commit`**; se invece uno o più messaggi sono negativi oppure il timeout scatta senza che il TM abbia ricevuto tutte le risposte, il TM scrive sul suo log un record di global abort.

La *seconda fase* del protocollo si articola nel modo seguente:

1. il TM trasmette la sua decisione globale agli RM; imposta poi un timeout, che scatterà in presenza di un eccessivo ritardo nella ricezione di messaggi di risposta dagli RM.
2. Gli RM che sono in uno stato **`ready`** attendono il messaggio di decision; non appena il messaggio arriva, scrivono sul proprio log il record di commit oppure di abort, che vanno interpretati localmente; inviano poi al TM un messaggio di **`acknowledgement`** (*ack*).
3. Il TM colleziona tutti i messaggi di ack dagli RM coinvolti nella seconda fase. Se tutti gli ack arrivano regolarmente, scrive sul suo log il record di complete; se invece il timeout scatta senza che il TM abbia ricevuto tutti gli ack, il TM imposta un altro timeout e ripete la trasmissione verso tutti gli RM da cui non ha ricevuto un ack. Questa sequenza si ripete fintantoché tutti gli RM non rispondono inviando il loro ack.

Quindi, l'assenza di comunicazione fra TM e RM durante la prima fase provoca un abort globale, mentre l'assenza di comunicazione fra TM e RM durante la seconda fase provoca una ripetizione delle trasmissioni.
Un RM in stato ready perde la propria autonomia e si affida alla decisione del TM; l'intervallo di tempo che intercorre tra la scrittura sul log del record ready e la scrittura del record commit o abort

è detto intervallo di incertezza; il protocollo è progettato in modo da ridurre tale intervallo al minimo.

Protocolli di ripristino

Affrontiamo ora le possibili cause di errore che possono compromettere la correttezza del protocollo di commit a due fasi.

Caduta di un partecipante.

La caduta di un partecipante comporta la perdita del contenuto dei buffer e può quindi lasciare la base di dati in uno stato inconsistente. Il protocollo di ripresa a caldo ci indica come comportarci in due casi, nei quali il fatto che una transazione sia distribuita oppure centralizzata è del tutto irrilevante:

- Quando l'ultimo record scritto nel log è un record che descrive una azione oppure un record di abort, le azioni vanno disfatte;
- Quando l'ultimo record scritto nel log è un commit, le azioni vanno rifatte.

Quindi, l'unico caso critico aggiuntivo, dovuto alla presenza di transazioni distribuite, si verifica per quelle transazioni in cui l'ultimo record scritto nel log è di ready. In tal caso, il partecipante è in dubbio circa l'esito della transazione. Durante il protocollo di recovery a caldo, vengono collezionati in un insieme gli identificatori delle transazioni in dubbio; per ciascuna di queste transazioni è necessario richiedere l'esito finale della transazione medesima al processo master;

Caduta del coordinatore

La caduta di un coordinatore avviene durante la trasmissione dei messaggi e comporta la loro eventuale perdita; Lo stato del TM è caratterizzato dai tre seguenti casi:

- Quando l'ultimo record nel log è un prepare, la caduta del TM può effettivamente aver posto alcuni RM in situazione di blocco. Il loro recovery, guidato dal TM, avviene normalmente decidendo per un global abort, scrivendo tale record nel log, e poi svolgendo la seconda fase del protocollo. Alternativamente, il TM può ripetere anche la prima fase, sperando che tutti gli RM siano ancora in attesa in stato di ready, per poter così decidere un global commit.
- Quando l'ultimo record nel log è una global decision, la caduta del TM può aver causato una situazione in cui alcuni RM sono stati correttamente avvertiti dalla decisione e altri sono viceversa rimasti in stato di blocco.
- Quando l'ultimo record nel log è una complete, la caduta del coordinatore non ha effetto sulla transazione.

Perdita di messaggi e partizionamento della rete

Infine analizziamo le perdite di messaggi e partizionamento della rete:

- Le perdite di un messaggio prepare o del successivo messaggio ready non sono distinguibili dal TM; in entrambi i casi scatta un timeout sulla prima fase, e viene presa una decisine di global abort;
- Le perdite di un messaggio decision o del successivo messaggio ack non sono anch'esse distinguibili; in entrambi i casi scatta un timeout nella seconda fase, e quest'ultima viene ripetuta;
- Un *partizionamento della rete* non provoca problemi ulteriori, in quanto la transazione avrà successo solo se il TM e tutti gli RM appartengono, durante le fasi critiche del protocollo, alla stessa partizione.

Ottimizzazioni del commit a due fasi

Il protocollo assume che tutte le scritture siano sincrone in modo da garantirne la persistenza. In effetti, alcune varianti del protocollo consentono di evitare la scrittura sincrona di alcuni record nel log. E' possibile cioè che il TM, in assenza di informazione circa alcuni partecipanti, indichi per default che questi partecipanti hanno preso una decisione di commit o abort.

Si costruiscono così due varianti del protocollo, dette di *commit presunto* oppure di *abort presunto.*

Protocollo di abort presunto

Il protocollo di abort presunto si basa sulla seguente regola:

A ogni richiesta di remote recovery da parte di un partecipante in dubbio sulla cui transazione il TM non abbia informazione viene restituita la decisione di abort.
Come conseguenza è possibile evitare scritture alcune sincrone di record nel log del TM; in particolare, si può evitare di scrivere con la primitiva force i record di prepare e global abort. Inoltre, anche il record di complete non è critico per l'algoritmo.

Ottimizzazione "sola lettura"

Un'ulteriore ottimizzazione del protocollo di commit a due fasi si verifica quando un partecipante, pur essendo coinvolto nel protocollo di commit a due fasi, scopre durante l'esecuzione di aver svolto solo operazioni di lettura ma nessuna operazione di scrittura. L'ottimizzazione del protocollo di un partecipante che scopra di essere di "sola lettura" consiste nel rispondere un messaggio *read-only* al messaggio di prepare, proveniente dal coordinatore. Dopo aver ricevuto la risposta di *read-only*, il coordinatore ignora il partecipante nella seconda fase del protocollo.

Interoperabilità

L'interoperabilità è il principale problema nella realizzazione di applicazioni eterogenee delle basi di dati distribuite. Il termine denota la capacità di interagire a livello di sistema, e richiede la disponibilità di funzioni di adattamento e conversione che renda possibile lo scambio di informazioni fra sistemi, reti e applicazioni, anche eterogenee. L'interoperabilità è realizzata attraverso protocolli standard, quali quelli per lo scambio di file (ftp), la posta elettronica e così via.
In questo programma vediamo due proposte: ODBC, uno standard per garantire accessi remoti senza commit a due fasi, e X-Open, uno standard relativo al protocollo di commit.

Open Database Connectivity: ODBC

Lo standard Open Database Connectivity è una interfaccia applicative Microsoft per costruire applicazioni eterogenee, supportate dall maggior parte dei prodotti relazionali. Tramite una interfaccia ODBC, le applicazioni scritte in SQL possono accedere a dati remoti.
Nella architettura ODBC, il collegamento tra una applicazione e un server richiede l'uso di un driver, una libreria che viene collegata dinamicamente alle applicazioni e da esse invocata. Il driver maschera le differenze di interazione legate non solo al DBMS, ma anche al sistema operativo e al protocollo di rete utilizzato. Il driver così maschera tutti i problemi di interoperabilità e facilità al massimo la scrittura delle applicazioni.
L'accesso a una base di dati remota tramite ODBC richiede al cooperazione di quattro componenti di sistema:

- L'applicazione richiama le funzioni SQL per eseguire interrogazioni e per acquisirne i risultati. L'applicazione è trasparente rispetto alla scelta del protocollo di comunicazione;
- Il driver manager è responsabile di caricare i driver a richiesta dell'applicazione;
- I driver sono responsabili di eseguire funzioni ODBC; pertanto, sono in grado di eseguire interrogazioni in SQL. I driver sono anche responsabili di restituire i risultati alle applicazioni, tramite meccanismi di buffering.
- La fonte di informazione è il sistema remoto, che esegue le funzioni trasmesse dal client.

Le interrogazioni SQL possono essere specificate in modo statico oppure essere incluse in stringhe che vengono generate ed eseguite dinamicamente.

Commit standard: X-Open DTP

Il protocollo X-Open DTP (Distribuited Transaction Processing) è un protocollo che garantisce l'interoperabilità di computazioni transazionali su DBMS di venditori differenti. L'architettura di X-Open DTP assume la presenza di un processo client, vari processi RM e un processo TM che interagiscono. Il protocollo consta di due interfacce:

- L'interfaccia fra client e TM, chiamata TM-interface
- L'interfaccia fra TM e RM, chiamata XA-interface

L'aspetto rilevante dello standard è che i costruttori di DBMS, per garantire che i loro server siano accessibili dai TM, devono garantire la disponibilità della XA-interface. Per questo motivo vari

sistemi relazionali hanno, in aggiunta a una realizzazione proprietaria del protocollo di commit a due fasi utilizzata per realizzare applicazioni relazionali omogenee, anche una implementazione della interfaccia XA, utilizzata per realizzare applicazione transazionali eterogenee. Le caratteristiche principali dello standard X-Open DTP sono le seguenti:

- Lo standard prevede che gli RM siano totalmente passivi;
- Il protocollo realizza un commit a due fasi con le ottimizzazioni di abort presunto e sola lettura
- Il protocollo prevede decisioni euristiche, che in presenza di guasti consentono la evoluzione della transazione sotto il controllo dell'operatore.

Cooperazione con sistemi preesistenti

Il grande sviluppo delle reti degli ultimi anni ha portato in molti casi a opportunità di integrazione di sistemi informativi esistenti. In questo contesto, è necessario distinguere i concetti di interoperabilità e cooperazione, che consiste nella capacità della applicazioni di un sistema di fare uso dei servizi applicativi messi a disposizione da altri sistemi, anche gestiti da soggetti diversi. La cooperazione è talvolta centrata sui processi. Noi centriamo l'attenzione sulla cooperazione basata sui dati, in cui i dati di un sistema sono visibiliad altri sistemi.
Autonomia, eterogeneità e distribuzione portano spesso a difficoltà significative nello sviluppo dei progetti di cooperazione, e vengono viste come ostacoli al processo stesso, da rimuovere attraverso opportune iniziative di standardizzazione o di razionalizzazione.
In alcuni casi le iniziative di cooperazione possono costituire una importante occasione per razionalizzare e riorganizzare questi sistemi, modificandone anche il grado di eterogeneità, di distribuzione e di autonomia. Il tutto richiede una valutazione dei costi e dei benefici, nonché dei requisiti normativi e della molteplicità delle iniziative di cooperazione.
Le forme di cooperazione centrate sui dati possono essere molteplici; esse differiscono per livello di trasparenza, complessità delle operazioni gestite e livello dell'attualità dei dati.

- Il livello di trasparenza, misura quanto la distribuzione e l'eterogeneità dei dati siano mascherate
- La complessità delle operazioni distribuite è una misura del grado di coordinamento necessario per effettuare operazioni sulle basi di dati cooperanti
- Il livello di attualità dei dati è una misura di quanto sia necessario accedere a dati recenti.

Sulla base dei criteri sopra enunciati, possiamo individuare tre architetture, che rappresentano le tre principali alternative per garantire la cooperazione basata sui dati.

Una prima categoria è quella dei sistemi multidatabase; in questi sistemi le singole basi di dati partecipanti continuano a essere utilizzate dai rispettivi utenti. Ai singoli sistemi accedono anche moduli, chiamati mediatori, che trasformano e filtrano gli accessi mostrando solo la porzione di base di dati che si desidera esportare, e la mettono a disposizione di un gestore globale, che realizza l'integrazione e mette a disposizione degli utenti della cooperazione una visione integrata.

Una seconda categoria di sistemi utilizza collezioni di dati replicati per garantire l'accesso in sola lettura a copie secondarie e derivate, ed elaborazioni fuori linea.; l'unica differenza è la presenza della data warehouse, che contiene sistemi estratti da vari sistemi eterogenei distribuiti e ofre una visione globale dei dati.

Una terza architettura è quella dei sistemi informativi locali con dati esterni. La differenza sostanziale rispetto ai casi precedenti è costituita dal fatto che in questa architettura non c'è alcun gestore locale e le integrazioni sono realizzate esplicitament dall'applicazione che integra.

Parallelismo

Il parallelismo è una dimensione importante della tecnologia delle basi di dati. Il parallelismo si è imposto all'inizio degli anni Novanta assieme alla diffusione di architetture multiprocessore a memoria condivida o a memorie separate.
Il motivo del successo del parallelismo nelle basi di dati è che le computazioni svolte da una base di dati si prestano a essere eseguite in parallelo con grande efficienza. In generale la gestione dei dati avviene tramite operazioni ripetitive, che si prestano a essere parallelizzate con prestazioni ideali.

Parallelismo intrer-query e intra-query

Il parallelismo viene introdotto nelle basi di dati con uno scopo preciso: garantire prstazioni migliori. Ci sono due tipologie di parallelismo:

- Il parallelismo si dice inter-query quando si eseguono interrogazioni diverse in parallelo. Il carico cui il DBMS è sottoposto è tipicamente caratterizzato da molteplici transazioni molto semplici;
- Il parallelismo si dice inra-query quando si eseguono parti della stessa interrogazione in parallelo; il carico cui il DBMS sottoposto è caratterizzato da poche interrogazioni assai complesse.

In entrambi i casi il parallelismo consente che a ciascun processore venga indirizzata una frazione del carico. Nel parallelismo inter-query il parallelismo viene introdotto moltiplicando il numero di processi server e allocando su ciascun processore un numero ideali di questi processi. Le interrogazioni sono raccolte da un processo dispatcher che ridirige ciascuna interrogazione verso uno dei processi server.
Il parallelismo intra-query è tipicamente caratterizzato da interrogazioni complesse, che coinvolgono cioè molti operatori e si valutano su basi di dati di grosse dimensioni;
in genere si dedica ad una determinata interrogazione un insieme ben definito di processi. Per sfruttare il parallelismo intra-query, l'ottimizzatore deve individuare una decomposizione dell'interrogazione in sotto-interrogazioni e prevedere le modalità di coordinamento e sincronizzazione fra di esse.

Parallelismo e frammentazione dei dati

Parallelismo è normalmente associato alla frammentazione dei dati: i frammenti vengono distribuiti su più processori e allocati su dischi distinti. Questa frammentazione può essere statica, cioè permanente, oppure dinamica, realizzata cioè poco prima di rispondere a una specifica interrogazione.

Speed-up e scale-up

Gli effetti del parallelismo vengono tipicamente descritti da due curve, dette di speed-up e scale-up. La curva di Speed-up caratterizza solo il parallelismo inter-query e misura il crescere delle prestazioni, misurate in tps al crescere del numero di processori.
La curva di scale-up caratterizza sia il parallelismo inter-query sia il parallelismo intra-query e misura il costo di una singola transazione al crescere del numero di processori. Nei sistemi OLTP l'aumento dei processori consente di gestire un numero maggiore di transazioni al secondo e risponde quindi a un accentuato carico transazionale, mentre nei sistemi OLAP l'aumento dei processori è spesso associato a un argomento della numerosità dei dati, che fa crescere la complessità delle interrogazioni.

11. Basi di dati a oggetti

Le basi di dati a oggetti, sviluppate a partire dalla seconda metà degli anni Ottanta, integrano la tecnologia delle basi di dati con il paradigma a oggetti, sviluppato nell'ambito dei linguaggi di programmazione e utilizzato, sul piano metodologico, nell'ambito dell'ingegneria del software.
Nelle basi di dati a oggetti ogni entità del mondo reale è rappresentata da un oggetto. Nel modello relazione ogni oggetto si trova distribuito su di un alto numero di tabelle; una visione unitaria dell'oggetto richiede query complesse che lo ricostruiscano estraendone i componenti dalle varie tabelle, tramite join. Le basi di dati a oggetti vengono incontro a questa esigenza applicativa mediante sistemi che consentono:

- La specifica di strutture dati complesse con relazioni semantiche fra i dati
- La descrizione integrata dei dati e delle operazioni disponibili
- Una integrazione stretta con i linguaggi di programmazione a oggetti

E' possibile riconoscere due approcci nella introduzione degli oggetti alle basi di dati. Le basi di dati orientate agli oggetti (*Object-Oriented Database Management Systems*, OODBMS) hanno assunto un approccio più rivoluzionario, estendendo i DBMS. I sistemi "relazionali e a oggetti" (*Object-Relational Database Management Systems*, ORDBMS) hanno viceversa assunto un approccio più evolutivo, integrando il concetto di oggetto all'interno del mondo relazionale.

Basi di dati a oggetti

Tipi

I tipi, in un linguaggio di programmazione a oggetti, consentono di definire le proprietà degli oggetti; in particolare, distinguiamo proprietà statiche e dinamiche. La parte statica di un tipo è costituita usando i cosiddetti costruttori di tipo e un insieme abbastanza esteso di tipi di dati atomici.
Alcuni sistemi consentono la definizione di tipi enumerativi, i cui valori vengono elencati dall'utente. I tipi atomici includono gli identificatori di oggetto. Ogni definizione di tipo associa un nome a un tipo.

Tipi di dati complessi

I costruttori di tipo consentono di costruire tipi, detti *tipi di dati complessi*, che descrivono la struttura delle istanze. Abbiamo costruttori di record, di insieme, multi-insieme, e lista.
Dato un tipo complesso T, un oggetto che ha per tipo T si dice istanza di T. I costruttori di tipo sono ortogonali, possono cioè essere applicati in modo arbitrario, risultando in oggetti di arbitraria complessità.
L'uso dei costruttori di tipo garantisce la cosiddetta complessità strutturale degli oggetti; in particolare, se un oggetto del mondo reale è complesso, i costruttori di tipo ci consentono di modellarlo in modo accurato.
Data un valore di tipo, è possibile accedere ai suoi componenti tramite la classica notazione punto, che può essere applicata in modo ricorsivo.

Oggetti e valori

La possibilità di introdurre un'arbitraria complessità strutturale, soddisfa l'esigenza di associare a un unico oggetto una struttura arbitrariamente complessa; quindi, un automezzo viene descritto in modo più articolato e unitario che non ad esempio, usando il modello relazionale e separandone la descrizione in tante tabelle.
Per ovviare a questo problema, si usano i riferimenti fra oggetti. La parte strutturale di un oggetto è in realtà costituita da una coppia; il valore è una istanza del tipo dell'oggetto; lo chiamano lo "stato" dell'oggetto.

Identità e uguaglianza

L'uso di OID consente anche di garantire la possibilità che due oggetti distinti abbiano lo stesso stato e differiscano solo per l'OID; questa possibilità non è consentita dal modello relazionale. Diremo che due oggetti O1 e O2 sono identici quando cannolo stesso identificatore, e utilizzeremo la nozione di uguaglianza per confrontare lo stato di due oggetti.

Nel modello a oggetti esistono due nozioni di uguaglianza:

- L'*uguaglianza superficiale (==)* richiede che due oggetti abbiano lo stesso stato
- L'*uguaglianza profonda (===)* richiede che due oggetti abbiano identici valori ottenuti sostituendo ricorsivamente gli oggetti raggiungibili tramite OID agli OID stessi.

Classi

Una classe raccoglie oggetti dello stesso tipo; essa funge cioè da contenitore di oggetti, che possono essere dinamicamente aggiunti e tolti alla classe. Gli oggetti che appartengono alla sessa classe sono omogenei. In genere la definizione di una classe è separata in due parti.

- L'interfaccia descrive il tipo statico e dinamico degli oggetti appartenenti alla classe; il tipo dinamico include la segnatura di tutti i metodi della classe.
- L'implementazione descrive il codice dei metodi e, talvolta, le strutture dati per memorizzare gli oggetti.

Il principio di incapsulamento, una importante astrazione dei linguaggi a oggetti, deriva dal più generale concetto di tipo di dato astratto che attribuisce a ciascun oggetto un'interfaccia e una implementazione. L'interfaccia descrive solo le operazioni applicabili sull'oggetto, mentre l'implementazione nasconde la struttura dati e l'effettiva costruzione delle operazioni.
Modelli dei dati più complessi riservano al concetto di classe il solo ruolo di definire la implementazione dei metodi, introducendo poi un terzo concetto, quello di estensione, che consente di inserire oggetti dello stesso tipo in collezioni differenti e dare nomi distinti a queste collezioni.

Metodi

I metodi vengono usati per manipolare gli oggetti di un OODBMS; la loro presenza è il principale elemento di innovazione di un OODBMS rispetto a una base di dati relazionale. Un metodo ha una segnatura, che ne descrive l'interfaccia e comprende tutte le informazioni che consentono di invocarlo, e una implementazione, che contiene il codice del metodo.
In genere ciascun metodo è associato con una specifica classe di oggetti. Esistono però sistemi che ammettono metodi multitarget, cioè si applicano a un numero arbitrario di oggetti senza privilegiarne uno in modo specifico. Assumeremo inoltre che ogni metodo abbia un numero arbitrario di parametri di ingresso e un unico parametro in uscita.
I metodi presenti in un OODBMS possono essere classificati in :

- Costruttori, utilizzati per costruire gli oggetti
- Distruttori, servono per cancellare gli oggetti
- Accessori, restituiscono informazioni sugli oggetti
- Trasformatori, cambiano il contenuto dello stato degli oggetti.

Altri metodi non possono esser classificati in base a questo schema e rispondono a specifiche esigenze applicative.
In molti sistemi i metodi sono distinti in pubblici e privati, quelli pubblici sono richiamabili da qualunque programma applicativo, quelli privati solo all'interno degli altri metodi.

Mismatch di impedenza

Gli esempi di metodi appena visti consentono di introdurre una discussine relativa a una caratteristica importante delle basi di dati a oggetti, nelle quali il programmatore può manipolare oggetti persistenti tramite le istruzioni del linguaggio di programmazione. Si dice che i database a oggetti risolvono il cosiddetto *mismatch d'impedenza* che caratterizza invece i sistemi relazionali; esso consiste nella necessità di far dialogare un linguaggio di programmazione, contenente variabili scalari, con il linguaggio di SQL che estrae insiemi di tuple.
Criteri di progettazione dei metodi
Uno dei principali vantaggi che la programmazione a oggetti offre è la possibilità di *riusare* le varie componenti di un sistema; infatti, se i metodi vengono progettati con attenzione, prevedendo a priori tutti i modi sensati di interagire con gli oggetti della classe, una gran parte del codice applicativo viene definito una volta per tutte, incluso nei metodi, e riusato dalle varie applicazioni.

Alcuni criteri vengono in aiuto al progettista nell'impostare il progetto dei metodi, così da garantirne la massima ricusabilità. La metodologia OMT propone i seguenti suggerimenti:

1. i metodi devono essere brevi
2. i metodi devono essere coerenti
3. i metodi non dovrebbero confondere al loro interno le politiche con le implementazioni
4. i metodi dovrebbero anticipare i requisiti di applicazioni future
5. i metodi devono essere autonomi
6. si deve sfruttare al massimo l'ereditarietà.

Gerarchie di generalizzazione

La possibilità di stabilire gerarchie di generalizzazione fra classi è probabilmente l'astrazione più importante nei linguaggi e nelle basi di dati a oggetti. Le generalizzazioni consentono la definizione di una sotto-classe a partire da una super-classe; la sotto-classe eredita lo stato e il comportamento della super-classe, e può in aggiunta rendere il proprio stato e comportamento più specifici tramite l'aggiunta di attributi e metodi. In una gerarchia di generalizzazione:

- tutti gli oggetti delle sotto-classi appartengono automaticamente alle superclassi;
- tutte le proprietà e i metodi delle super-classi vengono ereditati dalle sottoclassi;
- è possibile introdurre nella descrizione delle sotto-classi delle nuove proprietà e dei nuovi metodi.

In virtù dell'ereditarietà, le proprietà e i metodi definiti nell'ambito di tutte le super-classi sono automaticamente ereditati dalle sotto-classi.

Migrazioni fra le classi

In presenza di gerarchie di generalizzazione, in alcuni OODBMS gli oggetti possono migrare da un livello di gerarchia a un altro; in altri invece gli oggetti rimangono nella classe i cui sono creati durante tutta la loro esistenza. Si chiama specializzazione l'operazione tramite la quale un oggetto migra da una super-classe a una sotto-classe, è detta generalizzazione la migrazione inversa e consente a un oggetto di migrare da una sotto-classe a ua superclasse.

Distinguiamo tra istanze e membri di una classe; un oggetto è istanza di una classe solo se essa è la classe più specializzata per l'oggetto nell'ambito di una gerarchia di generalizzazione; le istanze di una classe sono automaticamente membri delle sue super-classi.

In alcuni sistemi è consentito a una classe di ereditare da più super-classi; questa situazione si dice ereditarietà multipla.

Conflitti

Le ereditarietà multipla oppure la presenza di oggetti che sono istanze di molteplici classi possono essere la fonte di *conflitti di nome,* qualora due o più super-classi abbiano proprietà o metodi con lo stesso nome. Alcune delle soluzioni possibili sono:

- rilevare il conflitto all'atto della definizione delle classi e non accettare come corrette le definizioni.
- Definire dei meccanismi per togliere ogni ambiguità della scelta.
- Ridefinire le proprietà e i metodi localmente.

Persistenza

Gli oggetti definiti in un programma possono essere persistenti oppure temporanei; gli oggetti temporanei cessano di esistere al termine dell'esecuzione del programma, mentre gli oggetti persistenti vengono inseriti nell'OODB.

Un oggetto diviene persistente tramite i seguenti meccanismi:

- Tramite l'inserimento in una classe che è definita come persistente. Si adopera in tal caso la primitiva new.
- Tramite la raggiungibilità a partire da un altro oggetto persistente.
- Tramite la denominazione, cioè dando a un oggetto un nome che possa essere utilizzato pe ritrovarlo a una successiva invocazione del programma.

La persistenza tramite raggiungibilità garantisce la cosiddetta “integrità referenziale” dell’OODB, cioè il mantenimento automatico di vincoli di integrità referenziale fra le classi che sono simili ai vincoli di riferimento fra tabelle.
Un oggetto può essere cancellato dal sistema solo quando esso non è più raggiungibile tramite denominazione o tramite riferimenti da altri oggetti persistenti; questo processo si chiama garbage collection.

Ridefinizioni dei metodi

Una volta introdotta una gerarchia è possibile ridefinire i metodi delle sotto-classi; questa tecnica, detta overriding dei metodi è estremamente utile per garantire una interfaccia uniforme dei metodi.
Per effetto della ridefinizione, è possibile avere quindi varie versioni dello stesso metodo con identica interfaccia; questo fenomeno è chiamato overloading dei nomi dei metodi.

13. Architetture e paradigmi per l'analisi dei dati

Per molti decenni lo sviluppo della tecnologia ha trascurato l'analisi dei dati. Si è pensato che i linguaggi di interrogazione fossero sufficienti sia per la gestione operativa sia per l'analisi. In effetti, SQL consente di costruire interrogazioni in modo arbitrario e offre quindi alcune caratteristiche utili per l'analisi, ma i linguaggi per l'analisi dei dati devono essere adatti a utenti che non sono necessariamente esperti informatici.
Con l'inizio degli anni Novanta, parallelamente allo sviluppo delle reti e dei prodotti per la distribuzione dei dati, si sono imposte nuove architetture, caratterizzate dalla *separazione degli ambienti*: a fianco dei sistemi per OLTP si sono sviluppati i sistemi OLAP. L'elemento principale della architettura OLAP, che svolge il ruolo di server è la *data warehouse*.
Mentre i sistemi OLTP sono normalmente condivisi da un elevatissimo numero di utenti finali, i sistemi OLAP sono caratterizzati dalla presenza di pochi utenti, che però occupano posizioni di alto livello nell'impresa e svolgono attività di *supporto alle decisioni*.
Mentre normalmente nei sistemi OLTP viene descritto solo lo "stato corrente" di una applicazione, i dati presenti nella warehouse possono essere di tipo *storico-temporale*.
Un altro problema importante nella gestione di una warehouse è quello della *qualità dei dati:* spesso la semplice raccolta di dati sulla warehouse non consente analisi significative, in quanto i dati contengono molte inesattezze, errori e omissioni. Infine, le data warehouse consentono l'uso di tecniche algoritmiche particolari, dette di *data minino*, utilizzate per la ricerca di informazioni nascoste nei dati.

Architettura della data warehouse

Una data warehouse (DW) contiene dati che vengono estratti da uno o più sistemi, detti *data source*, le data source includono una vasta tipologia di sistemi. Per l'estrazione dei dati da fonti eterogenee e legaci si utilizzano le tecniche già discusse.
Nell'architettura di una DW si possono definire i seguenti componenti, dei quali i primi due operano nella data source, gli altri 5 nella data warehouse:

- Un componente di *filtraggio dei dati (data filter)* che ne controlli la correttezza prima dell'inserimento nella warehosue. In tal modo, viene fatta la pulizia dei dati (data cleaning), che è essenzialmente per assicurare un sufficiente livello di qualità.
- Un componente per l'esportazione dei dati che consente di estrarre i dati dalla data source.
- Un compoente di *acquisizione dei dati (loader)*, responsabile di caricare inizialmente i dati nella DW, svolgendo anche operazioni di ordinamento e aggregazione e costruendo le strutture dati della DW. Più spesso i dati vengono allineati in modo incrementale, utilizzando il modulo di allineamento dei dati.
- Un componente di *allineamento dei dati (refresh)* che propaga incrementalmente le modifiche della data source in modo da aggiornare il contenuto della DW. Si possono usare due tecniche: *invio dei dati (data shipping)* e *invio delle transazioni (transaction shipping).* In entrambi i casi, gli archivi transazionali vengono poi utilizzati per *rinfrescare*(*refresh*) la DW.
- Un componente per *data minino*, che consente di svolgere ricerche sofisticate sulle informazioni "nascoste" nei dati.
- Un componente per l'*esportazione dei dati*, che consente di esportare i dati presenti in una warehouse ad altre DW, realizzando così una architettura gerarchica.

In aggiunta, una DW è speso dotata di moduli di ausilio alla sua progettazione e gestione:

- Un componente per l'*assistenza allo sviluppo* della warehouse
- Un componente, detto *dizionario dei dati*, che descrive il contenuto della DW

La qualità dei dati è un elemento essenziale per il successo di una DW; infatti, se i dati memorizzati contengono imprecisioni o errori, l'analisi risultante sarà necessariamente fuorviante, e l'uso della DW potrà risultare addirittura controproducente. Purtroppo, vari fattori pregiudicano la qualità dei dati:

- In basi di dati prive di vincoli di integrità, ad esempio, perché gestite con tecnologie pre-relazionali, il *tasso di errori (dirty data)* è assai elevato;
- In DW costruite assemblando dati estratti da plurime fonti si aggiungono problemi di disallineamento dei dati relativi allo stesso fenomeno osservati in basi dati diverse.

Schema della data warehouse

La costruzione di una *DW aziendale*, che descriva tutti i dati presenti in una impresa, è un obiettivo ambizioso ma anche assai difficile da realizzare. Per questo motivo, è prevalente l'approccio di costruire la DW concentrandosi separatamente su sottoinsiemi molto semplici dei dati aziendali. Ciascun schema semplificato dei dati dipartimentali prende il nome di *data mart*. I dati di un data mart sono organizzati secondo una semplice struttura, detta *schema multidimensionale* oppure più semplicemente *schema a stella*; il primo termine mette in luce la presenza di molteplici dimensioni di analisi, il secondo la struttura stellare dello schema una volta che esso venga interpretato con il modello E-R classico.

Schema a stella

Lo schema a stella ha una struttura semplicissima a stella utilizzando il modello E-R. Una entità centrale rappresenta i fatti; varie unità disposte a raggiera intorno a essa rappresentano le *dimensioni* dell'analisi; varie relazioni uno-molti collegano ciascuna occorrenza di fatto a una e una sola occorrenza di ciascuna delle dimensioni.
La principale caratteristica dello schema a stella è la sua struttura regolare e indipendente dal problema considerato; ovviamente, il numero di dimensioni di ciascun problema può essere diverso, ma occorrono almeno due dimensioni; anche un numero elevato di dimensioni è sconsigliabile, perché la gestione dei fatti e l'analisi si complicano.

Schema a fiocco di neve

Lo *schema a fiocco di neve* è una evoluzione del semplice schema a stella in cui le dimensioni vengono strutturate gerarchicamente, introdotta per tener conto della presenza di dimensioni non normalizzate. Il principale vantaggio dello schema a stella è la sua semplicità. Lo schema a fiocco di neve rappresenta in modo esplicito le gerarchie, quindi si presta sia alla riduzione di ridondanze e anomalie, sia alla costruzione di interfacce anch'esse abbastanza semplici.

Operazioni per l'analisi dei dati

Interfacce per la formulazione delle query

L'analisi dei dati di un data mart organizzato a stella richiede innanzitutto l'estrazione di un sottoinsieme dei fatti e delle dimensioni, in base agli interessi degli analisti. Tale estrazione dei dati segue un paradigma standard: le dimensioni vengono usate per selezionare i dati e per raggrupparli, mentre i fatti vengono tipicamente aggregati.
La query viene presentata all'utente in forma gabellare oppure in forma grafica, ad esempio tramite diagrammi a barre con l'uso di differenti colori per rappresentare diversi tipi di prodotti. In una forma matriciale, le dimensioni corrispondono alle righe e alle colonne, mentre i dati corrispondono alle celle della matrice, come in un *foglio elettronico*.

Drill down e roll up

L'analogia con I fogli elettronici non si limita alla prestazione dei dati; infatti, le nuove operazioni di *drill down* e *roll up* si ispirano ad analoghe operazioni presenti nei fogli elettronici.
Il drill down consente di aggiunger euna dimensione di analisi disaggregando i dati.
Dualmente il roll up consente di eliminare una dimensione di analisi, riaggregando i dati. Alternando operazioni di roll up e drill down, l'analista può evidenziare la dipendenza dei fenomeni rappresentati nei fatti dalle varie dimensioni, e così evidenziare le loro proprietà.

Data cube

La ricorrenza nell'uso delle aggregazioni ha suggerito l'introduzione di un operatore molto potente, detto *data cube*, per svolgere tutte le possibili aggregazioni presenti in una tabella estratta per l'analisi.
Il data cube si costruisce aggiungendo la clausola **`with cube`** ad una query che contenga una clausola di raggruppamento (**`group by`**).

Realizzazione della data warehouse

Per la realizzazione delle DW si contrappongono due soluzioni.

- La prima soluzione consiste nell'uso della tecnologia relazionale, opportunamente adattata ed estesa; i dati vengono memorizzati tramite tabelle, ma le operazioni di analisi vengono realizzate efficientemente tramite strutture dati speciali. Sistemi di questo tipo si dicono ROLAP (Relational OLAP).
- La seconda soluzione, più radicale, memorizza i dati direttamente in forma multidimensionale, tramite strutture dati vettoriali. Sistemi di questo tipo si dicono MOLAP (Multidimensional OLAP).

La soluzione MOLAP è praticata da un gran numero di piccole ditte informatiche che hanno realizzato prodotti specializzati nella gestione di data mart. La soluzione ROLAP è praticata dai grandi costruttori relazionali.
In ogni caso, le due tecnologie utilizzano soluzioni innovative per l'accesso ai dati, in particolare per quanto concerne l'uso di indici e la materializzazione delle viste. Queste soluzioni tengono conto del fatto che la DW viene usata essenzialmente per operazioni di lettura o caricamento iniziale dei dati, mentre modifiche e cancellazione sono assai rare.

Indici bitmap e indici di join

Gli indici bitmap consentono una realizzazione efficiente delle congiunzioni o disgiunzioni nel predicato di selezione, oppure operazioni algebriche di unione e intersezione. Ovviamente un indice bitmap è difficile da gestire se la tabella subisce modifiche: è conveniente costruirlo solo a valle di una operazione di caricamento dei dati, per una data cardinalità della tabella.
Gli indici di join consentono una realizzazione efficiente delle operazioni di join fra le tabelle delle dimensioni e la tabella dei fatti. Tali operazioni di join, estraggono quei fatti che soddisfano condizioni poste sulle dimensioni.

Materializzazione delle viste

Molte interrogazioni sulla DW richiedono ripetutamente aggregazioni e sintesi assai laboriose; in tal caso può essere conveniente valutare viste che esprimano i dati aggregati una volta per tutte, e memorizzarle; questa tecnica, introdotta nel capitolo 3, prende il nome di materializzazione delle viste. La scelta delle viste da materializzare è un problema abbastanza complesso, che richiede la conoscenza delle tipiche interrogazioni usate in un data mart e della loro frequenza di esecuzione.